Unleashing the Power of Allocator-Aware Software Infrastructure

Document #:	P2126R0
Date:	2020-08-12
Project:	Programming Language C++

Reply-to:	Pablo Halpern (on behalf of Bloomberg) <phalpern@halpernwightsoftware.com> John Lakos <jlakos@bloomberg.net>

NOTE: This white paper (i.e., this is not a proposal) is intended to motivate continued investment in developing and maturing better memory allocators in the C++ Standard as well as to counter misinformation about allocators, their costs and benefits, and whether they should have a continuing role in the C++ library and language.

Abstract

Local (arena) memory allocators have been demonstrated to be effective at improving runtime performance both empirically in repeated controlled experiments and anecdotally in a variety of real-world applications. The initial development and subsequent maintenance effort of implementing bespoke data structures using custom memory allocation, however, are typically substantial and often untenable, especially in the limited timeframes that urgent business needs impose. To address such recurring performance needs effectively across the enterprise, Bloomberg has adopted a consistent, ubiquitous allocator-aware software infrastructure (AASI) based on the PMR-style[1] plug-in memory allocator protocol pioneered at Bloomberg and adopted into the C++17 Standard Library.

In this paper, we highlight the essential mechanics and subtle nuances of programing on top of Bloomberg's AASI platform. In addition to delineating how to obtain performance gains safely at minimal development cost, we explore many of the inherent collateral benefits — such as object placement, metrics gathering, testing, debugging, and so on — that an AASI naturally affords. After reading this paper and surveying the available concrete allocators (provided in Appendix A), a competent C++ developer should be adequately equipped to extract substantial performance and other collateral benefits immediately using our AASI.

[1] polymorphic memory resource model

Contents

1 Introduction

Bloomberg's allocator-aware software infrastructure (AASI) allows clients to readily customize memory allocation strategies to improve runtime performance – often substantially and sometimes dramatically. Let's explore how to exploit this infrastructure through a couple of examples before diving into the technical details.

1.1 A Simple Example

Consider a simple unique_chars() function, which returns the number of unique bytes in its string argument:

```
bsl::size_t unique_chars(const bsl::string& s)
{
     bsl::set<char> uniq;
     uniq.insert(s.begin(), s.end());
     return uniq.size();
}
```

All of the temporary memory used by the set container is allocated from the general-purpose heap. We observe, however, that the set is built up monotonically and destroyed all at once, which is the ideal use pattern for a sequential allocator (as described later in the "bdlma::SequentialAllocator and Its Variants" section). Using a sequential allocator is as simple as passing one to the set constructor:

```
bdlma::SequentialAllocator alloc;
bsl::set<char> uniq(&alloc);
```

This one-line change yields a 75–78sample input data set. Yet instrumentation might point us to a way to do even better. Let's count the number of bytes allocated by interposing a counting allocator between the set and the sequential allocator:

```
bdlma::SequentialAllocator alloc;
bdlma::CountingAllocator countingAlloc(&alloc);
bsl::set<char> uniq(&countingAlloc);
```

For our data set, we observe that a total of less than 2KB is typically allocated by the set. This amount of memory can be allocated painlessly from the stack using a LocalSequentialAllocator instead of a plain SequentialAllocator:

```
bdlma::LocalSequentialAllocator<2048> alloc;
bsl::set<char> uniq(&alloc);
```

This trivial change cuts another 3–4reduction compared to the SequentialAllocator version). In total, we've cut execution time by over 80small, local change.

1.2 A More Realistic Example

The previous example is artificially simple for illustrative purposes. Let's now turn our attention to a more realistic example abstracted from large-scale data processing software that Bloomberg's DataLayer team developed. We begin by declaring a function that must allocate memory in its implementation:

```
bsl::string_view findInstrument(bsl::string_view wholeTopic,
bslma::Allocator *transient = 0);
```

In the Bloomberg's DataLayer software, the `transient` argument name implies that the memory allocated from this allocator is transient, i.e., will be returned to the allocator before the function returns. Exposing this function's use of memory through this implicitly documented interface can reasonably be called a leaky abstraction, but its pervasive use in DataLayer has been shown to preserve modularity in all other respects and is thus a reasonable engineering compromise. The `findInstrument` function is called within a loop in the `run` function:

```
void processInstrument(bsl::string_view instrument);
void run(const bsl::vector<bsl::string>& topics)
{
    bdlma::SequentialAllocator loopAlloc;
    for (topic_iter topic = topics.begin();
         topic != topics.end(); ++topic) {
         loopAlloc.rewind();
        bsl::string_view instrument = findInstrument(*topic, &loopAlloc);
        processInstrument(instrument);
    }
}
```

The `SequentialAllocator` defined in the first line of the `run` function is well suited for allocating transient memory, especially when repeated allocate-deallocate-reallocate cycles are rare (i.e., when allocation is done in the function body and most deallocation occurs only at the end). The sequential allocator uses the global heap sparingly, allocating ever larger pools of memory from which client blocks are carved, thus preserving locality among the allocated blocks. A final, critical attribute of `SequentialAllocator` (and all BDE[2] managed allocators, as described in "The bslma Allocator Infrastructure," the next major section) is the `rewind` method, which logically puts the allocator in its freshly created state, except that any memory blocks already retrieved from the global heap remain associated with the allocator. Additional use of the allocator after calling `rewind()` will reuse these blocks of memory, reducing global heap interaction even further and keeping allocated blocks hot in the cache.

Completing our example, the `findInstrument` function uses the transient allocator to build a vector:

```
void loadInstrumentPattern(bdlpcre::RegEx *regex);

bsl::string_view findInstrument(bsl::string_view wholeTopic,
                                bslma::Allocator *transient)
```

[2]Bloomberg Development Environment

```
{
     static bdlpcre::RegEx pattern(bslma::Default::globalAllocator());
     BSLMT_ONCE_DO { loadInstrumentPattern(&pattern); }
     bsl::vector<bslstl::StringRef> matches(transient);
     matches.reserve(pattern.numSubpatterns() + 1);
     pattern.matchRaw(&matches, wholeTopic.data(), wholeTopic.size());
     return matches.empty() ? wholeTopic : bsl::string_view(matches[0]);
}
```

The transient allocator is used to provide memory for the vector, which is destroyed (and therefore releases all of its memory) at the end of the function. The call to reserve minimizes or eliminates the allocate-deallocate-reallocate cycle that vectors are known for when they grow one element at a time. Note the use of the global allocator in the first line of `findInstrument`. The object being initialized has static storage duration and will thus outlive the function call. In a test scenario, it might also outlive the default allocator. The global allocator should be used for any allocator-aware object with static lifetime (see the "Choosing an Allocator Type for Optimal Performance" section later in this paper).

1.3 Lessons from the `findInstrument` Example

The top-to-bottom example in the previous section illustrates

1. the creation of an AA object (a vector) with a custom-selected allocator,
2. the use of sequential allocators for performance gain,
3. engineering tradeoffs (abstraction versus customization) sometimes needed to take advantage of an AASI,
4. the use of the global allocator for static objects, and
5. the overall simplicity of employing Bloomberg's AASI to use and experiment with allocators.

The fifth point deserves some elaboration. How do we know that `bdlma::SequentialAllocator` provides the best performance? We don't. The guidelines in the "Choosing an Allocator Type for Optimal Performance" section certainly help, and profiling showed excellent (if not optimal) performance benefits, but we might still do better. For example, we might increase performance further by replacing the `SequentialAllocator` with a `LocalSequentialAllocator`. Fortunately, choosing a different allocator is trivial: Simply replace the allocator type with a different allocator type and profile the code again to see the performance gain or loss.

The DataLayer team at Bloomberg has benchmarked its codebase using both the default (**`new`**/**`delete`**) allocator and ubiquitous use of sequential allocators and found that sequential allocators provided an order of magnitude speedup in practice. Microbenchmarks similarly achieved up to a 12x speedup for large data sets where elements are reused intensively.[3] The mechanics of using an allocator are simple; experimentation with different types of allocators is feasible because the cost of such experimentation is so low. The DataLayer code, from which this example is distilled, needs

[3] [**?**]; see Section 8, "Benchmark II: Variation in Locality (Long Running)," pp. 28–47.

this attention to detail because it is running near-real-time software in a resource-constrained environment, i.e., the client PC, where we have no control of the CPU or memory specifications. We cannot "throw hardware at the problem"; we must optimize. Higher up the stack, e.g., at points in the application that run at user interaction frequency rather than streaming data frequency, custom allocators would make little difference in performance; the programmer can pretend that allocators don't exist and use the infrastructure defaults. The advantages of having an AASI go beyond performance gains,[4] delivering important collateral benefits such as the ability to

1. place entire objects within an arbitrary memory arena,
2. instrument (and gather metrics for) individual regions of arbitrarily large systems,
3. compose with other allocators that implement triggers and alarms (e.g., when a fixed-size buffer overflows), and
4. ensure that memory allocation is properly implemented (e.g., using our `bslma::TestAllocator` facility[5]).

1.4 An Overview of This Paper

In this paper, we aim to demonstrate how an application-level developer can fully exploit the many benefits afforded by a pre-existing AASI platform without necessarily creating AA types themselves. Specifically, this paper treats allocator pointers as if they were opaque tokens passed from client code into AASI classes; we defer the description of how to create an AA class that actually allocates and deallocates memory using an allocator to a companion paper.[6]

We begin by introducing the pure abstract `bslma::Allocator` interface (also called a protocol) and, in particular, the concept of a *managed allocator*, i.e., one having an additional `release` method (a useful but dangerous tool that we will discuss near the end of the paper). We then present a short description of each of the allocators that BDE has provided and opine on how best to choose an appropriate allocator in various situations. Next, we describe how to associate a particular allocator with an AA object — including a Standard-Library container — at construction. Finally, we discuss some advanced topics, including how to use `release` to extract every last cycle out of the deallocation process. Along the way, we will alert the reader to common allocator-related errors and how to avoid them. In Appendix A, we go into additional detail about the off-the-shelf BDE allocators.

2 The `bslma Allocator` Infrastructure

The pure abstract class `bslma::Allocator` defines a protocol for allocating and deallocating memory. It stands at the root of a hierarchy of allocator classes, with concrete classes at the leaves, as shown in the figure below.

[4] See "Collateral Benefits" in [**?**].
[5] A version of the test allocator has been proposed for the C++ Standard Library [**?**].
[6] [**?**] provides a detailed description of how to create reusable AA library components.

A *managed allocator* manages a pool of memory and returns it to the general heap (or to the upstream allocator, as described later) upon destruction or upon a call to the `release` method. This pooling improves performance by providing locality for objects that are used together. Typically, a managed allocator is created for a limited scope (which nevertheless can be long lived) to construct objects used only within that scope. When the memory pool is released on destruction, any blocks that have not been returned to the allocator are automatically freed. This bulk deallocation can be useful for the advanced techniques of *winking out* and *localized garbage* collection, both of which are described near the end of this paper. A managed allocator should be derived from the `bdlma::ManagedAllocator` pure abstract class.

Allocators are typically designed such that an instance can be accessed safely by only one thread at a time (i.e., they are not *thread aware*) since having multiple threads allocate and deallocate from the same pool would destroy locality. That does not mean that they cannot be used in multithreaded programs. On the contrary, single-threaded access is ideal for avoiding synchronization overhead when objects are created, used, and destroyed all within a single thread. Moreover, an allocator that is not thread aware can still be effective if a set of objects moves to a new thread, as long as the allocator moves with them and is never used in an interleaved fashion by more than one thread.

There are two allocators that are known globally throughout the program: the *default allocator* and the *global allocator*. As its name implies, the default allocator is used to allocate memory when no other allocator is specified. The global allocator is reserved for constructing objects of static duration, including objects at global scope. In rare cases when creating an AA object of static duration is absolutely necessary, the developer must explicitly pass `bslma::Default::globalAllocator()` to its constructor to prevent improperly using the default allocator. Typically, both default and global allocators refer to the `bslma::NewDeleteAllocator` singleton. Changing either of them to refer to a different allocator is a task deferred to the developer in charge of the program as a whole (i.e., the owner of `main()`) and is described in the "Advanced Topics" section of this paper.

With rare exceptions, the allocators in the BDE collection of package groups are designed to allow chaining, whereby an allocator may be constructed with a pointer to another upstream allocator. When the first allocator runs out of its own private storage, it obtains memory from its upstream allocator. If an upstream allocator is not provided, the current default allocator is used.

Chaining allows the features of multiple allocators to be combined. For example, a multipool allocator can be made to use a preallocated buffer by choosing a buffered sequential allocator as its upstream allocator. Similarly, memory usage by multiple locally managed allocators can be monitored for efficiency and correctness by choosing a test allocator as their upstream allocator.

2.1 Three Off-the-Shelf Allocators

In this section, we introduce three BDE-provided allocators that every user should be aware of when tuning the memory-allocation performance of their program. Also see the "Allocator Types that Provide Nonperformance Collateral Benefits" section later in this paper and in Appendix A, where we list briefly other allocators included in the BDE library.

2.1.1 `bslma::NewDeleteAllocator`

The new-delete allocator simply forwards allocation requests to global **`operator new`** and deallocation requests to global **`operator delete`**. It is the default default allocator, i.e., a singleton of type `bslma::NewDeleteAllocator` is the default allocator unless some other default is set. To explicitly obtain this singleton, call `bslma::NewDeleteAllocator::singleton()`. The new-delete allocator is thread aware and always available. When the performance of the global **`new`** and **`delete`** is adequate, `bslma::NewDeleteAllocator` is an effective way to insulate code from changes to the default allocator.

2.1.2 `bdlma::MultipoolAllocator`

A `bdlma::MultipoolAllocator` object consists of an array of pools, one for each geometrically increasing request size in a range up to some specified maximum. Each time a block is requested, it is provided from the most appropriately sized pool, and is returned to that pool when that block is freed. When a pool is exhausted, the allocator replenishes it using chunks obtained from the upstream allocator (typically the default allocator), with such chunks having increasingly larger size up to a built-in limit. Requests that exceed the maximum pool size pass directly through to the upstream allocator. The design of `bdlma::MultipoolAllocator` makes allocations fast (because finding the best-fit free block is so efficient and because there is no thread synchronization), eliminates fragmentation, and maximizes locality (i.e., minimizes diffusion).

A `bdlma::MultipoolAllocator` is ideal for node-based containers with frequent insertions and deletions. When using this allocator within a loop, create the `bdlma::MultipoolAllocator` in the scope outside of the loop, so that the blocks obtained from the upstream allocator can be re-used efficiently.

2.1.3 `bdlma::SequentialAllocator` and Its Variants

A `bdlma::SequentialAllocator` supplies memory from a contiguous block sequentially until the block is exhausted and then dynamically allocates new blocks of geometrically increasing size from the upstream allocator (usually the default allocator). Returning memory to a `bdlma::SequentialAllocator` is a no-op; any deallocated memory remains unavailable for reuse until it is explicitly released (via the `release` or `rewind` methods) or the allocator object itself is destroyed. No allocator is faster for allocating memory than a `bdlma::SequentialAllocator`, nor does any other allocator provide better locality for allocated blocks of disparate sizes.

A `bdlma::SequentialAllocator` is ideal for data structures that get built up monotonically (elements are added but not removed), used, and then destroyed. Since a no-op `deallocate` means that object destructors do not return memory to the allocator, the developer, when using a sequential allocator in a loop, must take special care to prevent the allocator from consuming memory blocks of ever-increasing size, without bound, from its upstream allocator. The `bdlma::SequentialAllocator` provides a `rewind` method that logically frees all allocated memory but, unlike `release`, retains the pool of blocks obtained from the upstream allocator for reuse in subsequent allocations. A well-constructed loop using a `bdlma::SequentialAllocator` would call rewind in every iteration:

```
bdlma::SequentialAllocator alloc;
for (int i = 0; i < N; ++i) {
    alloc.rewind(); // Don't forget to do this!
    // ... use alloc ...
}
alloc.release(); // optional (if alloc won't be destroyed soon)
```

In the loop above, memory is allocated from the upstream allocator (the default allocator, in this case) only if an iteration requests more memory from `alloc` than any prior iteration. Eventually, memory consumption will reach a highwater mark and subsequent uses of alloc will be extremely efficient.

A `bdlma::BufferedSequentialAllocator` is a variation of `bdlma::SequentialAllocator` that is constructed with an initial buffer, avoiding allocation from the upstream allocator unless the initial buffer is exhausted. Another variant is `bdlma::LocalSequentialAllocator<SIZE>`, which has an initial buffer of specified SIZE built into the allocator's footprint, making it ideal for allocating memory directly from a fixed-sized buffer on the program stack:

```
// No allocation from the heap unless tempStr grows larger than 128 bytes
bdlma::LocalSequentialAllocator<128> stackAlloc;
bsl::string tempStr(&stackAlloc);
// Code that builds up tempStr
```

3 Choosing an Allocator Type for Optimal Performance

Before choosing an allocator for constructing a set of AA objects, consider whether custom allocation is truly needed. The main reasons for needing custom allocation are listed here.

To reduce time spent on allocation and deallocation: Profilers such as Quantify, gprof, and the Callgrind plugin for Valgrind will reveal where a program is consuming CPU resources. If allocation and deallocation are consuming significant CPU time, sequential and multipool allocators can drastically improve performance.

To reduce memory diffusion: Diffusion occurs when memory blocks in the working set[7] are spread throughout physical memory, causing cache misses and page faults. Vtune and Valgrind can help diagnose this problem. Local (managed) allocators can significantly improve locality, thus improving cache and virtual-memory effectiveness.

To exploit non-performance-related collateral benefits: To instrument code to track memory use or to place objects into specific parts of memory, an allocator is the best option.

The remainder of this section focuses on performance (reasons 1 and 2). If allocators are likely to improve the application's performance, the following guidelines will help determine which allocator (or combination of allocators) to use. The developer should profile the application before and after altering the code and be willing to experiment; easy experimentation is one of the main benefits of pluggable allocators.

[7] The working set of a process is the collection of information referenced by the process in a specific period of time [**?**].

- *When constructing an AA object of static duration* (i.e., a static variable, static member variable, global variable, or namespace-scoped variable) or thread duration (a thread-local variable), use the global allocator (obtained by calling `bslma::Default::globalAllocator()`). Developers can use a custom allocator if they can ensure that it will exist for the entire lifetime of the object.[8] Do not use the default allocator for static objects (e.g., by not specifying an allocator) because such use causes the default allocator to be set to the "default default" for the duration of the program (i.e., *frozen*, explained more in the "Changing the Default and Global Allocators" section) and thus interferes with the owner of `main` being able to set the default allocator after static objects are constructed.

- *When constructing an AA object that is a member of another AA object* or which is logically part of another AA object (i.e., when creating an AA class), use the same allocator as the containing object. We cover creating AA classes in detail in a subsequent paper.[9]

- *When constructing an AA object that will be swapped with or moved into another AA object*, use the allocator retrieved from the object into which the result will be moved or swapped. To retrieve the allocator from object X, call `X.allocator()` or (for types that come from the Standard Library) `X.get_allocator().mechanism()`. Matching the other object's allocator ensures that the move or swap operation will take constant time and will not throw an exception.

- *Do not use the allocator retrieved from an object to construct a nonowned variable* unless that local variable is intended to be swapped or moved into the object or one of its members, as described in the previous bulleted item. The allocator imbued into an object is intended for use only by members of that object; other uses could exhaust or fragment the object's memory pool.[10]

- *When constructing an object that will be modified by multiple threads* in an interleaved fashion, choose a thread-aware allocator, such as the global allocator or `bslma::NewDeleteAllocator::singleton()`.

- *When constructing a short-lived AA object*, such as a temporary string on the stack, consider using `bdlma::LocalSequentialAllocator`, which can often avoid accessing the global heap entirely.

- *When constructing a large data structure that will be built up monotonically* (elements added, but rarely, if ever, individually removed), consider using `bdlma::SequentialAllocator` or `bdlma::BufferedSequentialAllocator`. If the object being constructed is a vector, string, or unordered set/map, we strongly recommend calling `reserve` with the expected or maximum size of the container or string (before inserting the first element) to avoid allocating memory blocks that become unused during a container resize.

- *When constructing a data structure that will experience numerous insertions and deletions*, consider using `bdlma::MultipoolAllocator` to enable efficient memory reuse with robust locality. Chaining a `bdlma::SequentialAllocator` upstream from a `bdlma::MultipoolAllocator`

[8] Defining objects with global, namespace, class, or file scope that require initialization is a design violation, irrespective of the allocator.

[9] [**?**]

[10] For the same reason, a local allocator should never be chained to the allocator retrieved from an object, even within that object's constructors or member functions.

sometimes yields a noticeable performance benefit over either one alone. As always, measure and experiment; do not simply trust intuition or some rule of thumb.

- *When creating local AA objects within a deep (possibly recursive) call hierarchy*, consider creating a top-level `bdlma::MultipoolAllocator` and passing it down the call chain. Local AA objects in each function can use the passed-in allocator such that, as each function frame is popped from the call stack, local-variable destructors return memory blocks to the pool. Those newly freed blocks, while still hot in cache, can be immediately reused by AA variables in the next function call.

4 Using Allocators with the BDE AASI

If allocators provide the *service* of allocating memory, then instances of AA types are the clients of that service. The BDE library provides a large number of general-purpose classes as well as classes that have specific applicability to Bloomberg's business. Not all of those classes allocate memory, but those that do allocate give the programmer the option of customizing that allocation by providing an allocator on object construction.

The first step as a programmer using a BDE AA type is to determine whether custom allocation is desirable at all. Just because a type allows choosing a custom allocator doesn't mean that one is necessary; unless the default allocation strategy is deemed unacceptable (e.g., as indicated by profiling), the programmer can simply construct the object without supplying an allocator:

```
bdlc::BitArray ba(96); // bit array of length 96 using default allocation
```

To customize the bit array's allocation strategy requires creating an instance (i.e., defining a variable) of the chosen allocator type and passing its address as an additional argument to the AA type's constructor. Using `LocalSequentialAllocator` as an example, let's create a bit array that allocates the first 128 bytes of memory from a local stack buffer:

```
bdlma::LocalSequentialAllocator<128> alloc;
bdlc::BitArray ba(96, &alloc); // Bit array using custom allocation
```

In this example, the bit array will fit entirely within the stack buffer and, in fact, can grow quite a bit larger without going to the global heap for memory.

Note that the constructed AA object (ba in this case) retains a pointer to the allocator. Returning an AA object by value (usually a bad idea; see the "Copying, Moving, Inserting, and Returning AA Objects" section) or returning a pointer to a dynamically allocated AA object will result in a dangling pointer if the returned object was constructed using a local allocator.

Once constructed, the allocator pointer associated with an object is stable throughout the AA object's lifetime. This behavior is similar to that of a polymorphic object's *vtbl* pointer; once the constructor completes execution, it never changes until the destructor is invoked, even if the object is assigned to.

4.1 Copying, Moving, Inserting, and Returning AA Objects

The copy constructor of a BDE-style AA type never uses the allocator of the copied-from object. Instead, the newly constructed copy is imbued with the address of the currently installed default allocator. In contrast, the move constructor (C++11 and later or simulated in C++03 with `bslmf::MovableRef`) of an AA type does imbue the new object with the allocator of the moved-from object, achieving the same end result as if the moved-from object had been initially constructed directly at the new location.

An AA type provides an *extended copy constructor* and an *extended move constructor* that generally do the same job as the copy and move constructors, respectively, but allow the programmer to provide the address of an allocator via a trailing argument:

```
bsl::vector<int> vec(10, 8); // original (w/default allocator)
bdlma::SequentialAllocator alloc2;
bsl::vector<int> vec1(vec, &alloc2);          // extended copy ctor
bsl::vector<int> vec2(std::move(vec),
                      &alloc2);               // extended move ctor (C++11)
bsl::vector<int> vec3(bslmf::MovableRefUtil::move(vec),
&alloc2);               // extended move ctor (C++03)
```

In the case of the extended move constructor, if the supplied allocator is the same as the one used by the moved-from object, the behavior is identical to the (nonextended) move constructor; otherwise, the behavior is identical to the extended copy constructor. Thus, the extended move constructor automatically optimizes the move when possible while giving the programmer control over the constructed object's allocator.

The bsl package group contains (BDE-style) AA versions of most of the standard containers (`bsl::vector`, `bsl::set`, `bsl::unordered_map`, etc.). Include the bsl variants of standard headers (e.g., `#include <bsl_vector.h>` instead of `#include <vector>`) to get the versions that follow all of the AA rules described here.

When an AA object is inserted into an AA container, the container uses the object's extended copy or move constructor to construct the new element, passing its own allocator as the trailing argument, which ensures that all of the elements have the same allocator as the container itself. The performance and collateral benefits of the allocator are thus seamlessly extended to the entire container and its contained elements.

Try to structure self-contained subsystems such that items being move-assigned or swapped with each other are constructed with the same allocator. As mentioned in the previous section, the allocator pointer for an object remains constant throughout the object's lifetime. An operation such as move assignment or swap, which would normally simply move pointers, "degenerates" to a copy operation if the objects being assigned or swapped use different allocators. (In fact, the standard requires that standard containers have the same allocator when calling member `swap`. The BDE library currently relaxes this requirement but might not do so forever.) A move assignment manifesting as a copy is actually quite rare in practice because the most common uses of move assignment and swap are within container operations such as `insert` or `erase` or when calling sorting or shuffling algorithms on sequences within a container. Since all of the elements in a container have the same allocator as the container (and therefore as one another), the issue of

assigning or swapping between elements with different allocators is moot for the vast majority of oft-repeated (i.e., performance-critical) operations.

Returning AA types by value is contraindicated; a better practice is passing these types by address so that the caller can configure the allocator appropriately for the *caller*'s purposes. Consider an application function, makeIntSequence, that fills a vector with the integers 1 to n where n is a function argument. The vector result is passed to the function by address, allowing the caller to benefit from an optimized allocation strategy:

```
void makeIntSequence(bsl::vector<int> *v, int n) {
    v->reserve(n);
    for (int x = 1; x <= n; ++x) {
        v->push_back(x);
    }
}

bdlma::LocalSequentialAllocator<300> mySeqAllocator;
bsl::vector<int> seq(&mySeqAllocator);
makeIntSequence(&seq, 60);
```

"Returning" AA types via a pointer argument pays huge performance dividends when calling a function within a loop. If the object is returned by value, it needs to be constructed and destroyed each time through the loop. Conversely, if the object is passed in by address, it can be created (once) outside the loop and then reused every time through. For containers like `vector`, such construction/destruction avoidance can eliminate a lot of allocations and deallocations since the vector grows to some high-water mark and stays there until the loop terminates.[11]

Aside from performance concerns, returning an AA type by value can cause issues with correctness. Because of copy and move elision, the object received by the caller of a return-by-value function will end up capturing the allocator specified on construction of the return value rather than the (expected) default allocator. This behavior can be both surprising and dangerous as shown in the following rewrite of the `makeIntSequence` example in which the function author is attempting (incorrectly) to optimize the vector allocation.

```
bsl::vector<int> makeIntSequence(int n) {
    bdlma::LocalSequentialAllocator<512> mySeqAllocator;
    bsl::vector<int> ret(&mySeqAllocator);
    ret.reserve(n);
    for (int x = 1; x <= n; ++x) {
        ret.push_back(x);
    }
    return ret;
}

bsl::vector<int> seq = makeIntSequence(60); // Disaster waiting to happen
```

In the code above, the variable ret will be constructed with a buffered sequential allocator even though the caller never intended that. Worse than that, the allocator itself has gone out of scope.

[11] The benefit of returning a value via a pointer is not limited to AA types; any type that allocates memory or has an expensive constructor or destructor benefits from this treatment.

Worse still, the error is unlikely to be detected until much later, when reading seq returns random data from the stack or modifying seq corrupts the stack. If returning an AA type by value is absolutely necessary, then it is safest either to construct it using the default allocator or (in the case of some factory functions) to allow the caller to pass the result allocator as an (optional) argument.

5 Allocator Types that Provide Nonperformance Collateral Benefits

The benefits of pluggable allocators extend beyond performance. In this

section, we describe two important allocator types that provide the collateral benefits of instrumentation and object placement, respectively.

5.1 `bslma::TestAllocator`

As its name implies, the *test allocator* is used for testing. It gathers metrics and provides debugging facilities related to the program's use of memory. The most common use of a test allocator is in the test driver of an AA component. The use of a test allocator for validating the operation of an AA type and the use of a test allocator as an upstream allocator to test other allocators are covered in subsequent papers.[12]

For application programmers, the test allocator can help ensure that the program is using allocators correctly. If a program is experiencing mysterious crashes, a test allocator can be inserted into the allocator chain to detect memory leaks and logic errors in which the program writes beyond the start or end of an allocated block.

The test allocator provides the following features:

- keeps track of the amount of memory (in bytes and blocks) allocated and deallocated,
- detects memory leaks,
- detects attempts to deallocate the same block twice,
- detects certain overrun and underrun errors,
- dumps information about allocations and deallocations to the console under program control, and
- throws an exception after a configurable number of allocation operations.

This advanced feature is used to test the exception safety of AA types. Pass a test allocator to an AA object constructor to gather data on the number and size of allocations and high-water marks in memory usage. Pass a test allocator as the upstream allocator to another allocator to test whether that allocator is being used correctly. For example, creating a sequential allocator outside of a loop but using it inside the loop will cause unbounded growth in the use of the upstream

[12] [**?**]; [**?**]

allocator, which can be detected during testing by using a test allocator as the upstream allocator. In some cases, temporarily setting the default allocator to a test allocator may be appropriate to facilitate counting allocations and deallocations from the default allocator in a specific region of code.

5.2 Shared Memory Allocators

An allocator can also be used to place objects into special memory regions. An example would be an allocator that manages memory in memory-mapped pages.[13]

To work correctly, every part of the object, *including the footprint of the object itself*, must be allocated using the memory-mapping allocator; a programmer should never construct a static- or automatic-lifetime object with this type of allocator. One way to allocate an object from an allocator is to pass the allocator (not the allocator's address) to the placement-style operator new. Note that, when using **`operator new`** in this way, the programmer must also pass the allocator's address to the object's constructor. To delete an object allocated in this way, call the allocator's `deleteObject` method.

```
typedef bsl::vector<bsl::string> VecType;
bslma::Allocator& alloc = my_MemoryMappingAllocator::singleton();
VecType v1(&alloc); // BAD IDEA: object footprint not in mapped memory
VecType *v2_p = new(alloc) VecType(&alloc); // OK: mapped footprint
v2_p->push_back("hello"); // string element in mapped memory
// ...
alloc.deleteObject(v2_p); // 'Dont forget to free the memory.
```

5.3 Using Allocators with Shared and Managed Pointers

An allocator can occupy several different roles in a single instance of one of the smart pointer types, `bsl::shared_ptr<T>` or `bslma::ManagedPtr<T>`:

- as the source of memory for the managed object's footprint;
- as the *deleter* to destroy and deallocate the managed object. Although it can be specified independently, in a correct program the deleter must agree with (refer to the same allocator as) the source of the object's memory.
- as a constructor argument to the managed object, used by that object to allocate memory outside of its footprint;
- as the source of memory for the internal representation of the shared pointer itself (`bsl::shared_ptr` only).

[13] Bloomberg's Big environment is one of several very large, multifunction processes that run on Bloomberg's servers and execute terminal functions on behalf of users. The user's state is stored in a memory-mapped file, which is mapped into whichever Big process a user is temporarily assigned to. All objects created in that file are allocated using a Bloomberg internal allocator called `a_bdema_GmallocAllocator`.

In most cases, the same allocator should appear in all of these roles. To avoid inadvertent divergence, manually constructing these smart pointers and supplying allocators for each role is discouraged in favor of using the factory functions, `bsl::allocate_shared<T>` or `bslma::ManagedPtrUtil::allocateManaged<T>`, each of which take a single allocator argument and use it consistently:

```
bsl::shared_ptr<bsl::string> sharedStr =
    bsl::allocate_shared<bsl::string>(&alloc1, "hello");
bslma::ManagedPtr<bsl::string> managedStr =
    bslma::ManagedPtrUtil::allocateManaged<bsl::string>(&alloc1, "hello");
```

The load methods of both pointer types are similarly subject to inadvertent divergence in the use of allocators. Instead, call the factory functions above and assign the result:

```
sharedStr = bsl::allocate_shared<bsl::string>(&alloc2, "world");
managedStr =
bslma::ManagedPtrUtil::allocateManaged<bsl::string>(&alloc2, "world");
```

The shared internal representation of a `bsl::shared_ptr` and the deleter for both `bsl::shared_ptr` and `bslma::ManagedPtr` logically belong to the pointed-to object and have the same lifetime as the pointed-to object, i.e., they are destroyed (and the shared representation is deallocated) when the last smart pointer referring to the pointed-to object is destroyed. Smart pointers are often used when tracking the lifetime of the pointed-to object is difficult or impossible; choosing allocators having global scope when constructing them is advisable.

A common cause of confusion for smart pointer users is that, although the interface to `bsl::shared_ptr` and `bslma::ManagedPtr` have constructors that accept allocator arguments, smart pointers are not AA objects. The smartpointer templates do not have extended copy and extended move constructors, nor do they have an `allocator()` (or `get_allocator()`) method. Unlike AA types, copying or moving a smart pointer, either by copy construction or by copy/move assignment, will result in the target of the copy or move having the same allocator as the original. The allocator associated with a smart pointer can change over the lifetime of the pointer instance, i.e., through assignment, in contradiction of the rule for AA types that the allocator never changes:

```
bsl::shared_ptr<int> p1 = bsl::allocate_shared<int>(&alloc1, 1);
bsl::shared_ptr<int> p2 = bsl::allocate_shared<int>(&alloc2, 2);
bsl::shared_ptr<int> p3(p1); // copy-constructed p3 uses alloc1
p1 = p2; // p1 uses alloc2 after assignment
```

Finally, because the smart-pointer classes do not define the `bslma::UsesBslmaAllocator` type trait that identifies classes as being AA, a container of smart pointers does not imbue its allocator into its elements. Thus, most of the qualities of an AA type do not hold for either smart-pointer type.

6 Advanced Topics

Since this paper is about using the BDE AASI effectively, to be thorough we explain the features and techniques for getting every last ounce of value from it. The topics in this section are advanced but accessible to anyone with a moderate amount of experience using allocators.

6.1 Changing the Default and Global Allocators

The typical reason to explicitly configure the global or default allocator is for testing. In production code, the default and global allocators are almost never changed from their initial value of `bslma::NewDeleteAllocator::singleton()`. The direct mechanism for setting the default or global allocator is to call `bslma::Default::setDefaultAllocator` or `bslma::Default::setGlobalAllocator`, respectively. Only the owner of `main()` should call these functions. The arguments to these functions should have static lifetime. Once the default allocator has been used, it cannot be changed using `setDefaultAllocator`. In practice, this means that any variable of AA type that is constructed without an explicit allocator argument before `main` runs will freeze the default allocator.

Once frozen, the default allocator can still be changed for a limited scope, e.g., to install a counting or test allocator for a region of code. This temporary replacement can be accomplished by employing the `bslma::DefaultAllocatorGuard` class, which installs a new default allocator and then automatically reverts to the previously installed allocator when the guard goes out of scope. Note the default (or global) allocator should never be changed if more than one thread is running. The guard is primarily intended for validating correct behavior of AA classes in test drivers.

For more information about changing the default allocator, including full usage examples, see the component-level documentation for the bslma_default and `bslma_defaultallocatorguard` components.[14]

6.2 Winking Out

A managed allocator, such as `bdlma::SequentialAllocator` or `bdlma::MultipoolAllocator`, will reclaim all allocated memory on destruction or when the `release()` or `rewind()` method is called. Releasing memory in this way does not call any destructors and thus avoids accessing individual blocks unnecessarily. A specific optimization technique, called winking out, involves reclaiming memory for a container, especially one that holds other AA objects, without invoking the container's destructor and thus the destructors for its elements. To wink out a container, first create a managed allocator and then allocate the container itself from the allocator rather than creating it on the stack. That way, when the allocator goes out of scope, the container and all of its allocated memory are freed all at once:

```
{
    bdlma::MultipoolAllocator alloc;
    bsl::vector<bsl::list<bdlt::Calendar> >& data =
        *new(alloc) bsl::vector<bsl::list<bdlt::Calendar> >(&alloc);
    // ... Build up and use 'data' here ...
    // When 'alloc' goes out of scope, 'data' gets winked out;
    // no need to call 'alloc.deleteObject(&data)'.
}
```

In the example above, `data` is a reference to a vector allocated from the allocator (using the placement-style **`operator new`**, as described in the "Shared Memory Allocators" section). The

[14] [?]. [?]

vector's destructor is never called, but all allocated memory blocks are returned to the heap when `alloc` goes out of scope.[15]

Winking out is a powerful technique but is also dangerous. It can be used to successfully reclaim a data structure's memory only if (1) every subpart of the data structure that allocates memory uses the provided allocator and (2) no part of the data structure has a destructor with side effects other than releasing memory to the allocator. (Nonsemantic side effects in destructors, such as logging, will not happen, but that should cause no additional issues.) In particular, if a destructor releases a nonmemory resource, then that type is not a candidate for winking out because failing to run its destructor would result in a resource leak. For this reason, winking out is inappropriate in generic (template) code unless the template arguments are carefully constrained.

6.3 Local Garbage Collection

As a final example of the expressive design power that managed allocators provide, let's look at winking out as a technique not for improving performance (though it does that, too) but for simplifying the correct implementation of a data structure. Consider the problem of managing memory for an arbitrary graph (possibly with cycles). Representing a graph in such a way that nodes can be reclaimed without leaks or double deletions is notoriously difficult. Using a managed allocator, however, the nodes can be leaked without negative consequences because the entire graph is reclaimed when the allocator is destroyed:

```
struct GraphNode {
    bsl::string d_payload;
    bsl::vector<GraphNode *> d_outgoingEdges;
    GraphNode(const bsl::string& payload, bslma::Allocator *alloc);
    ~GraphNode() { }
};
GraphNode::GraphNode(const bsl::string& payload, bslma::Allocator *alloc)
: d_payload(payload, alloc), d_outgoingEdges(alloc) {
    d_outgoingEdges.reserve(2); // Typical fan-out is 2.
}
// ...

{
    bdlma::SequentialAllocator alloc;
    GraphNode *start = new(alloc) GraphNode("start", &alloc);
    // ...
    GraphNode *n = new(alloc) GraphNode(nodename, &alloc);
        start->d_outgoingEdges.push_back(n);
    n->d_outgoingEdges.push_back(start); // cycles are no problem
    // ...
    // 'alloc' destructor calls 'alloc.release()'
}
```

[15]The C++ Standard states that an object's lifetime ends when ". . . the storage which the object occupies is released, or is reused by an object that is not nested within [it]" [[**?**], section 6.8, paragraph 1.4, p. 74]. Thus, freeing an object without invoking its destructor is well-defined (valid) C++.

Note that no `GraphNode` object is ever individually destroyed or deallocated in the example above. Instead, the *entire* graph is deallocated when the allocator goes out of scope. Not only is this code simpler than using, e.g., reference-counted pointers (for which cycles are a huge problem), but it is likely more

efficient. Large graphs created with shared pointers have been known to overflow the call stack on destruction because each node's destructor recursively calls the destructor for the next node in the graph.

7 Conclusion

Customized local memory allocation can improve the runtime performance of most applications, sometimes dramatically. Writing custom data structures, however, is inherently costly and often impractical in real-world applications. Having a consistent and ubiquitous AASI based on the BDE/C++17-PMR style enables every application developer to realize essentially all of the benefits of custom memory allocation with minimal effort and much reduced time to product than would otherwise be possible.

The BDE AASI comes with a rich supply of allocators that can be used off-theshelf to effectively address a wide variety of application design patterns and scenarios. By following a few simple rules, these patterns can be identified and allocators can be applied safely and effectively to improve performance. What's more, with some additional effort, these allocators can be tuned to extract nearly every cycle that might be available.

The potential to improve runtime performance alone is compelling. Yet maximizing the value of our ubiquitously interoperable AASI will be achieved only by applying knowledge of the many convenient and productive ways — beyond mere performance — by which such a robust infrastructure can be exploited:

- placing objects at a particular location in memory, e.g., on the stack or in file-mapped memory,
- measuring and reporting per-object memory usage,
- testing correctness of allocation, and
- implementing efficient local garbage collection, e.g., in graph data structures.

By applying our recommendations, developers can benefit from the power of AASI.

A Other Allocators in the BDE library

Table 1 is a short list of other allocator types that the BDE infrastructure library provides. More information about each allocator type is found in its component-level documentation.[16]

[16] [**?**]

Table 1: Additional BDE allocators

Allocator Type	Purpose
`bslma::MallocFreeAllocator`	Uses the C library functions malloc and free to manage memory. Useful for bypassing a user-defined replacement for global operator new.
`bdlma::AligningAllocator`	Allows clients to specify both size and alignment of blocks to allocate. (Note that most containers cannot take advantage of this feature.)
`bdlma::ConcurrentMultipoolAllocator`	Similar to a `bdlma::MultipoolAllocator` but safe for concurrent allocation/deallocation.
`bdlma::ConcurrentPoolAllocator`	A concurrent allocator optimized for blocks of a single size.
`bdlma::CountingAllocator`	Counts the number of bytes allocated from the upstream allocator. (A mini version of the test allocator, with less overhead.)
`bdlma::GuardingAllocator`	Uses virtual memory faults to detect buffer overruns for debugging.
`bdlma::HeapBypassAllocator`	Allocates memory directly from the OS, without calling **`new`**/**`delete`** or `malloc`/`free`.

B Mapping BDE Names to C++17 Names

Through Bloomberg's efforts, much of the BDE allocator system was adopted as the PMR section of the C++17 Standard, using C++ standard naming conventions. Table 2 shows an approximate mapping of BDE types and free functions to their C++17 equivalents. All BDE names are in namespace BloombergLP, and all C++17 names are in namespace std.

[17] Although the `ManagedAllocator` base class has no equivalent, the standard pooling and monotonic resources adhere to a managed allocator concept in that they have a `release()` method.

Table 2: BDE to C++17 name mappings

BDE Name	Approximate C++17 Equivalent
`bslma::Allocator`	`pmr::memory_resource`
`bdlma::ManagedAllocator`	no equivalent[17]
`bsl::allocator<T>`	`pmr::polymorphic_allocator<T>`
`bslma::NewDeleteAllocator::singleton()`	`pmr::new_delete_resource()`
`bdlma::MultipoolAllocator`	`pmr::unsynchronized_pool_resource`
`bdlma::SequentialAllocator` or `bdlma::BufferedSequentialAllocator`	`pmr::monotonic_resource`
`bdlma::LocalSequentialAllocator`	no equivalent
`bslma::Default::defaultAllocator()`	`pmr::get_default_resource()`
`bslma::Default::setDefaultAllocator()`	`pmr::set_default_resource()`
`bslma::Default::globalAllocator()`	no equivalent
`bslma::Default::setGlobalAllocator()`	no equivalent
`bsl::string`	`std::pmr::string`
`bsl::vector<T>`	`std::pmr::vector<T>`
`bsl::list<T>`	`std::pmr::list<T>`
`bsl::set<T>`	`std::pmr::set<T>`
`bsl::map<K,V>`	`std::pmr::map<K,V>`
`bsl::unordered_set<T,H,E>`	`std::pmr::unordered_map<T,H,E>`
`bsl::unordered_map<K,V,H,E>`	`std::pmr::unordered_map<K,V,H,E>`

References