Making C++ Software *Allocator Aware*

Document #:	P2127R0
Date:	2021-01-08
Project:	Programming Language C++

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

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] 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] 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.

[1] Motivational background can be found in [**?**]. Information on using AASI can be found in [**?**].

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

Contents

1 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. 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

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. nadequate 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:

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.

Attribute classes: demonstration of how to identify missing constructors and add an optional allocator parameter to each existing constructor.

Classes that allocate memory: demonstration of how to use the allocator directly in the constructors, destructor, assignment operators, and swap function.

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.

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, bdlb_variant, or bslstl_sharedptr. This paper, however, provides sufficient information to render most common components consistently and interoperably with the BDE AASI.

2 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. This section describes the interface features common to all allocator-aware types consistent with the BDE infrastructure. 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. 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.

- Every constructor has a variant that can be invoked with an allocator to be used by the constructed object. 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. 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.