```
struct MonotonicAllocatorPartialImp : ManagedAllocator
{
    inline void deallocate(void* address) final { /* empty */ }
        // Deallocate the block at the specified address.
};
```

Notice that we have specified the empty `inline` `deallocate` member function of `MonotonicAllocatorPartialImp` to be **`final`**. A concrete monotonic allocator — e.g., a `BufferedSequentialAllocator` — can then derive from this partial implementation:

```
struct BufferedSequentialAllocator : MonotonicAllocatorPartialImp
{
    BufferedSequentialAllocator();
        // Create a default version of a buffered-sequential allocator.

    void* allocate(std::size_t numBytes);
        // Allocate a block of memory of at least the specified numBytes.

    void release();
        // Reclaim all memory currently allocated from this allocator.
};
```

Now consider two **allocator-aware** types, `TypeA` and `TypeB`, each of which is always constructed with some flavor of managed allocator:

```
struct TypeA
{
    TypeA(ManagedAllocator* a);
    // ...
};

struct TypeB
{
    TypeB(MonotonicAllocatorPartialImp* a);
    // ...
};
```

We now construct each of the types using the same concrete allocator object:

```
void client()
{
    BufferedSequentialAllocator a;  // Concrete monotonic allocator object

    TypeA ta(&a);  // deallocate is a virtual call to an empty function.
    TypeB tb(&a);  // deallocate is an inline call to an empty function.
}
```

When `TypeA` invokes `deallocate`, it goes through the nonfinal, virtual function interface of `ManagedAllocator` and is subject to the runtime overhead of dynamic dispatch. Note