particular, notice that `g(y)` is not **accessible** at `level 1`, can be used only with non-negative values at `level 2` (i.e., `g(y)` has a **narrow contract**), and is usable with all syntactically legal values at `level 3` (i.e., `g(y)` has a **wide contract**). Note that this same sort of interface widening can apply in the absence of virtual functions through judicious use of hiding nonvirtual functions; see *Potential Pitfalls — Systemic lost opportunities for reuse* on page 1023.

Concrete leaf nodes can then be derived from the protocol hierarchy to implement the desired level of service as efficiently as practical. In cases where multiple concrete nodes need to share the same implementation of one or more functions, we can derive an intermediate node from the appropriate *protocol* that doesn't widen the interface at all but does implement one or more of the pure abstract functions; such an *impure* abstract node is known as a **partial implementation**. When the implementation of one of these functions is trivial, declaring that `virtual` function to also be `inline` might be sensible. Since, by design, there will be no need to further override that function, we can declare it to be `final` as well.

For performance-critical clients that would otherwise consume the concrete object via the pure abstract interface from which this partial implementation derives, we might decide to instead take the partial implementation itself as the **reference type**. Because one or more functions are both `inline` and `final`, the client can dispense with runtime dispatch and inline the virtual functions directly as discussed in *Restoring performance lost to mocking* on page 1017.

As a real-world example, consider a simplified protocol hierarchy for memory allocation:

```
#include <cstddef>  // std::size_t

struct Allocator
{
    virtual void* allocate(std::size_t numBytes) = 0;
        // Allocate a block of memory of at least the specified numBytes.

    virtual void deallocate(void* address) = 0;
        // Deallocate the block at the specified address.
};

struct ManagedAllocator : Allocator
{
    virtual void release() = 0;
        // Reclaim all memory currently allocated from this allocator.
};
```

A **monotonic allocator** is a kind of **managed allocator** that allocates memory sequentially in a buffer subject to **alignment requirements**. In this class of allocators, the `deallocate` method is always a no-op; memory is reclaimed only when the managed allocator is destroyed or its `release` method is invoked: