Avoiding boilerplate code when employing structural inheritance

A key indication for using inheriting constructors is that the derived class ~~addresses~~ only auxiliary or optional~~, rather than required or necessary,~~ functionality ~~to~~ its self-sufficient base class. As an interesting, albeit mostly pedagogical, example, suppose we want to provide a proxy for an `std::vector` that performs explicit checking of indices supplied to its index operator:

```
#include <cassert>  // standard C assert macro
#include <vector>   // std::vector

template <typename T>
struct CheckedVector : std::vector<T>
{
    using std::vector<T>::vector;      // Inherit std::vector's constructors.

    T& operator[](std::size_t index)   // Hide std::vector's index operator.
    {
         assert(index < std::vector<T>::size());
         return std::vector<T>::operator[](index);
    }

    const T& operator[](std::size_t index) const  // Hide const index operator.
    {
         assert(index < std::vector<T>::size());
         return std::vector<T>::operator[](index);
    }
};
```

In the example above, inheriting constructors allowed us to use public structural inheritance to readily create a distinct new type having all of the functionality of its base type except for a couple of functions where we chose to augment the original behavior.

Although this example might be compelling, it suffers from inherent deficiencies making it insufficient for general use in practice: Passing the derived class to a function — whether by value or reference — will strip it of its auxiliary functionality. When we have access to the source, an alternative solution would be to use conditional compilation to add explicit checks in certain build configurations (e.g., using C-style `assert` macros).[1]

Avoiding boilerplate code when employing implementation inheritance

Sometimes it can be cost effective to adapt a **concrete class** having virtual functions to a specialized purpose by using inheritance. Useful design patterns exist where a **partial implementation** class, derived from a **pure abstract interface** (a.k.a. a **protocol**), con-

[1] A more robust solution along these same lines is anticipated for a future release of the C++ Language Standard and will be addressed in **lakos23**.