Importantly, knowing that the semantics of *move* operations can be fully defined in terms of their corresponding *copy* operations makes them easy to understand. What's more, these newly added move-semantic operations can be tested using **unit tests** that are modified only slightly from their existing copy-semantic counterparts; see *Potential Pitfalls — Making a noncopyable type movable without just cause* on page 788.

Repurposing internal resources from an object that no longer needs them can lead to faster *copy*-like operations, especially when dynamic allocation and deallocation of memory is involved. Countervailing considerations, such as **locality of reference**, can, however, suggest that preferring *move* operations to *copy* operations might, in some circumstances, be contraindicated for overall optimal runtime performance, especially at scale.[17]

Properly implementing *move* operations that, for expiring objects, act like optimized copy operations will depend on the specifics of how we chose to implement a type, e.g., an object that (1) is written to manage its own resources explicitly (see *Creating a low-level value-semantic type (VST)* below) or (2) delegates resource management to its subobjects (see *Description — Special member function generation* on page 732).

Creating a low-level value-semantic type (VST) Often, we want to create a user-defined type (UDT) that represents what we'll call a **platonic value**, i.e., one whose meaning is independent of its representation within the current process. When implemented properly, we refer to such a type as a **value-semantic type (VST)**. Although there are some cases where a VST might be implemented as a simple **aggregate type** (see *Description — Special member function generation* on page 732), there are other cases where a VST might, instead, manage its internal resources directly. The latter explicit implementation of a VST is the subject of this subsection.

For illustration purposes, consider a simple VST, **class** `String`, that maintains, as an **object invariant**, a null-*terminated* string value; i.e., this string class explicitly does *not* support having a *null* pointer value. In addition to a **value constructor** and a single **const** member function to access the value of the object, each of the four C++03 special member functions — **default constructor**, copy constructor, assignment operator, and destructor — are **user provided**, i.e., defined explicitly by the programmer. To keep this example focused, however, we will *not* store separately the length of the string, and we'll omit the notion of *excess* capacity altogether, leaving just a single nonstatic **const char*** data member, `d_str_p`, to hold the address of the dynamically allocated memory:

```
class String { const char* d_str_p; /*...*/ };  // null-terminated-string manager
```

One practical aspect that we preserve is that default-constructed container types — going back to C++03 — are well advised, on purely performance grounds, never to pre-allocate resources, lest creating a large array of such empty containers be impracticably runtime-intensive to construct:

[17] **halpern21c**