active member by means of **placement operator new**. The copy-assignment operator must destroy the current active member and construct the new one — tasks that it delegates to the `UShape`'s destructor and copy constructor, respectively. If the **union** will have the same active member after the assignment, we could avoid this destroy/construct dance, at the cost of significantly more logic, including another **switch** statement.

Our revised `UShape` **union** has the same compact representation as the original `UShape` described in the previous use case but is now a fully abstract data type having private data members, enforced invariants, and the ability to manage resources, but at the cost of significantly more code and complexity. Moreover, this type is no longer source-code compatible with C; however, there are techniques for making such types available in C; see *Translating a C++-only type to C (standard layout)*[19] below.

Translating a C++-only type to C (standard layout)

Well-designed C++ classes typically encapsulate private data behind an easy-to-understand, easy-to-use public interface. Many C++ classes naturally happen to be of standard-layout type. To be otherwise, the class would need to have nonstatic data members with different access levels, partake in **implementation inheritance** that involves nonstatic data in a base class, or have one or more virtual functions or virtual base classes.

Suppose that we have a C++ date and time library that includes a **value-semantic type**, `Date`, representing a calendar date in which a date value is encoded directly within the value representation of the footprint of an object. This particular type (for didactic purpose only) happens to inherit publicly from a **struct** `DateEncoder` that provides a **static** function for encoding a date as an integer:

```
// datetime.h:

struct DateEncoder  // empty base class
{
    static int encode(int year, int month, int day);
        // Return an encoding of a year, month, and day as a single integer
        // such that, if two encoded dates compare equal, they represent the
        // same date and, if one encoded date compares less than another encoded
        // date, then the first date comes before the second date in the
        // calendar.
};
```

The `Date` class encodes its date value within a single **int** data member. Having a single chronologically ordered integer to represent the encoding provides good performance for certain operations such as equality comparison and is particularly well suited for deter-

[19] The C++17 library template `std::variant` provides similar functionality without using an intrusive tag type, but doesn't allow straightforward access to additional fields that are common across variant members — e.g., the `d_orig` field in `UShape2` from *Vertical encoding for non-trivial types (standard layout)* on page 448.