

Defining `VertexMetadata` outside of the body of `dijkstra`, e.g., to comply with C++03 restrictions, would make that implementation-specific helper class directly accessible to anyone including the `dijkstra.h` header file. As Hyrum’s law² suggests, if the implementation-specific `VertexMetadata` detail is defined outside the function body, it is to be expected that some user somewhere will depend on it in its current form, making it problematic, if not impossible, to change.³ Conversely, encapsulating the type within the function body avoids unintended use by clients, while improving human cognition by colocating the definition of the type with its sole purpose.⁴

Instantiating templates with local function objects as type arguments

Suppose that we have a program that makes wide use of an aggregate data type, `City`:

```
#include <algorithm> // std::copy
#include <iostream> // std::ostream
#include <iterator> // std::ostream_iterator
#include <set> // std::set
#include <string> // std::string
#include <vector> // std::vector

struct City
{
    int d_uniqueId;
    std::string d_name;
};

std::ostream& operator<<(std::ostream& stream,
                        const City& object);
```

Consider now the task of writing a function to print unique elements of an `std::vector<City>`, ordered by name:

²“With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody” (**wight**).

³The C++20 *modules* facility enables the encapsulation of helper types, such as `metadata` in the `dijkstra.h` example on the previous page, used in the implementation of other locally defined types or functions, even when the helper types appear at namespace scope within the module.

⁴For a detailed discussion, see **lakos20**, section 0.5, “Malleable vs. Stable Software,” pp. 29–43.