```
#include <algorithm>  // std::sort
#include <string>     // std::string
#include <vector>     // std::vector

struct Person { std::string d_name; };

void sortByName(std::vector<Person>& people)
{
    std::sort(people.begin(), people.end(),
              [](const Person& lhs, const Person& rhs)
              {
                  return lhs.d_name < rhs.d_name;
              });
}
```

In the example above, the lambda expression passed to the `std::sort` algorithm is a local unnamed type, and the algorithm itself is a function template.

Use Cases

Encapsulating a type within a function

Limiting the scope and visibility of an **entity** to the body of a function actively prevents its direct use, even when the function body is exposed widely, say, as an **`inline`** function or function template defined within a header file.

Consider, for instance, an implementation of Dijkstra's algorithm that uses a local type to keep track of metadata for each vertex in the input graph:

```
// dijkstra.h:

#include <vector>  // std::vector

inline int dijkstra(std::vector<Vertex>* path, const Graph& graph)
{
    struct VertexMetadata         // implementation-specific helper class
    {
        int  d_distanceFromSource;
        bool d_inShortestPath;
    };

    std::vector<VertexMetadata> vertexMetadata(graph.numNodes());
        // standard vector of local VertexMetadata objects --- one per vertex

    // ... (body of algorithm)
}
```