Are Graphs Hard in Rust?

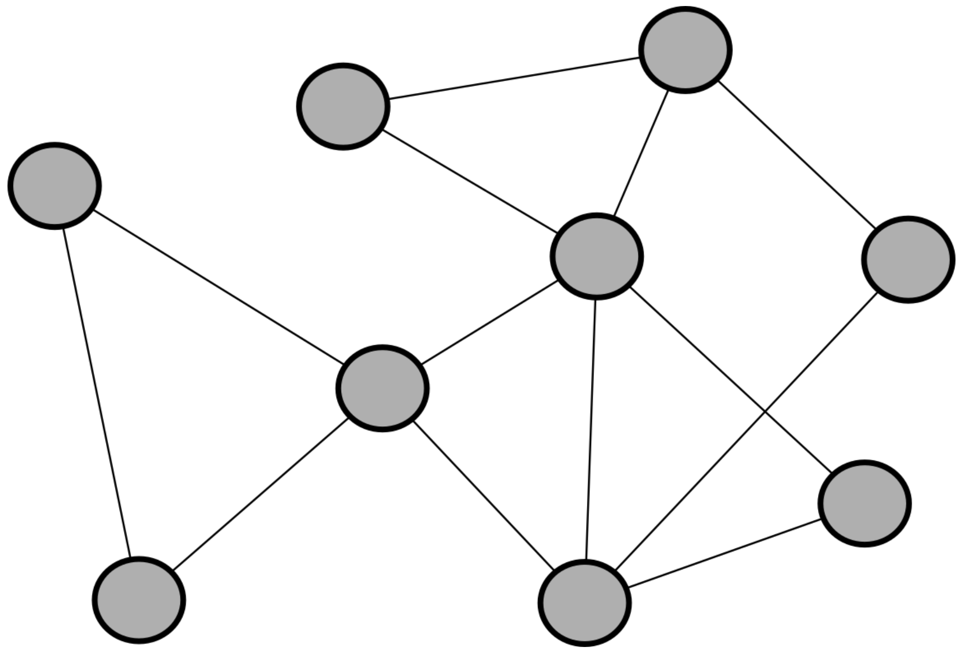
@PayasR

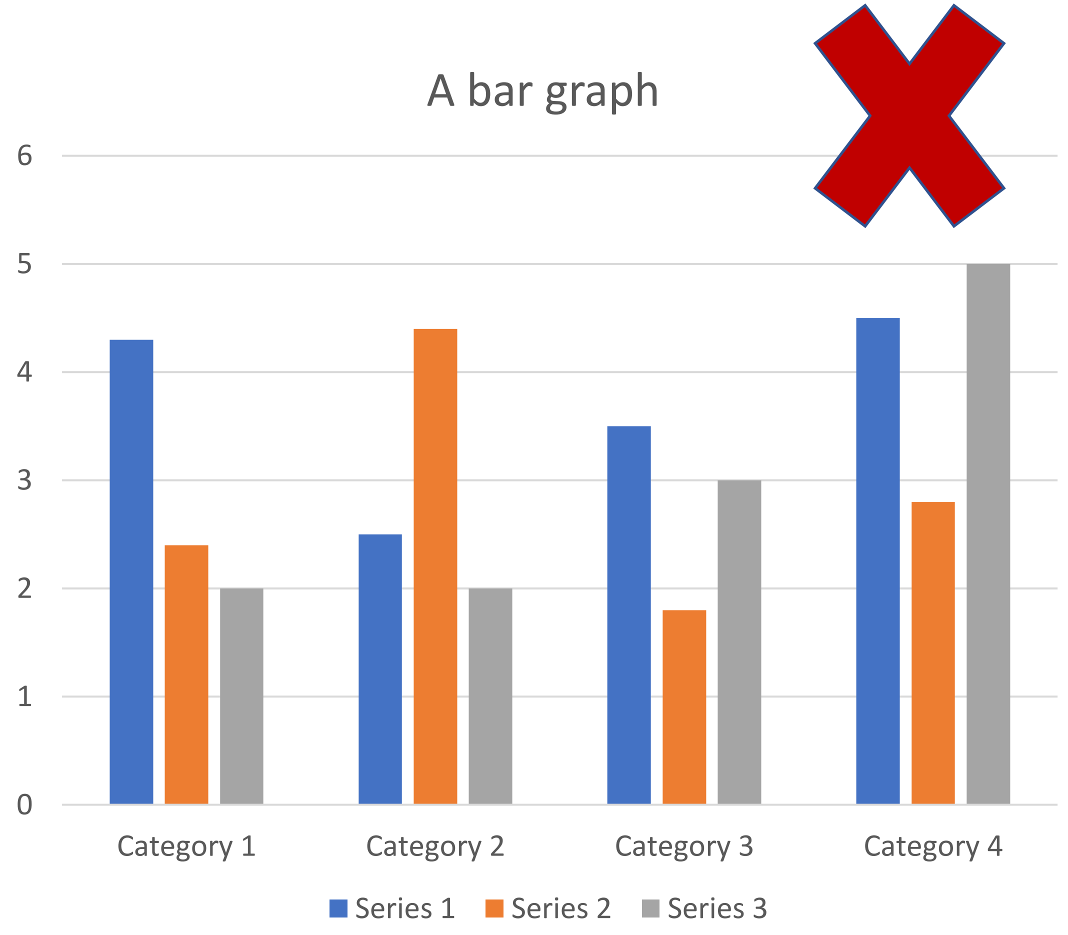
Are Graphs Hard in Rust?

Are Graphs Hard in Rust?

Graphs

Graph data structure





Questions…!

- What *are* graphs?

Graph representations:

- How do we represent graphs in our programs?
- Are all graph representations ‘hard’?
- Which ones are hard only in Rust but not in other languages (C++)? Why?

A few questions to start with…

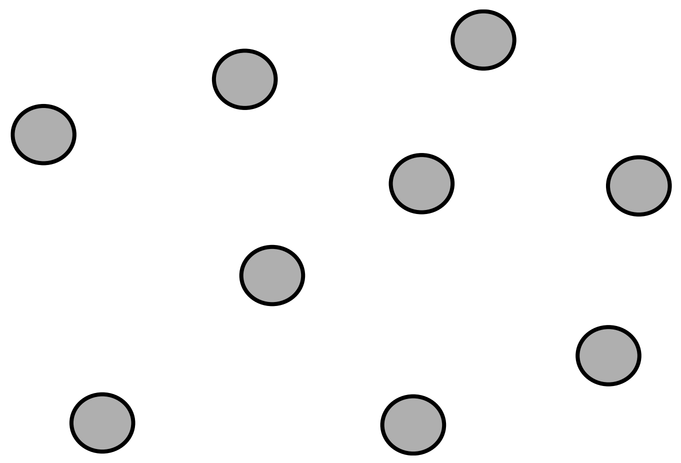
Graph Libraries:

- How do we design reusable graph libraries?
- Graph libraries = Graph representations + <*Other components?*>
  Are these other components hard too?
- If Rust makes a few things *harder*, does it make anything *easier*?

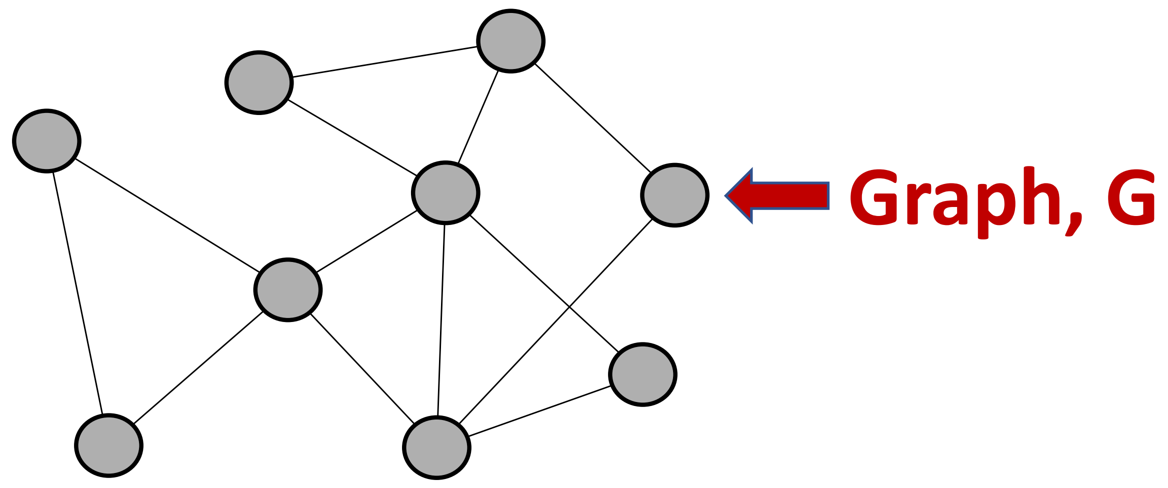
What are graphs?

A graph **G** consists of:

- A set of <u>*vertices*</u>, **V**.
- A set of <u>*edges*</u>, **E**, containing pairs **(x, y)** such that both **x** and **y** are in **V**.



Vertices, **V**

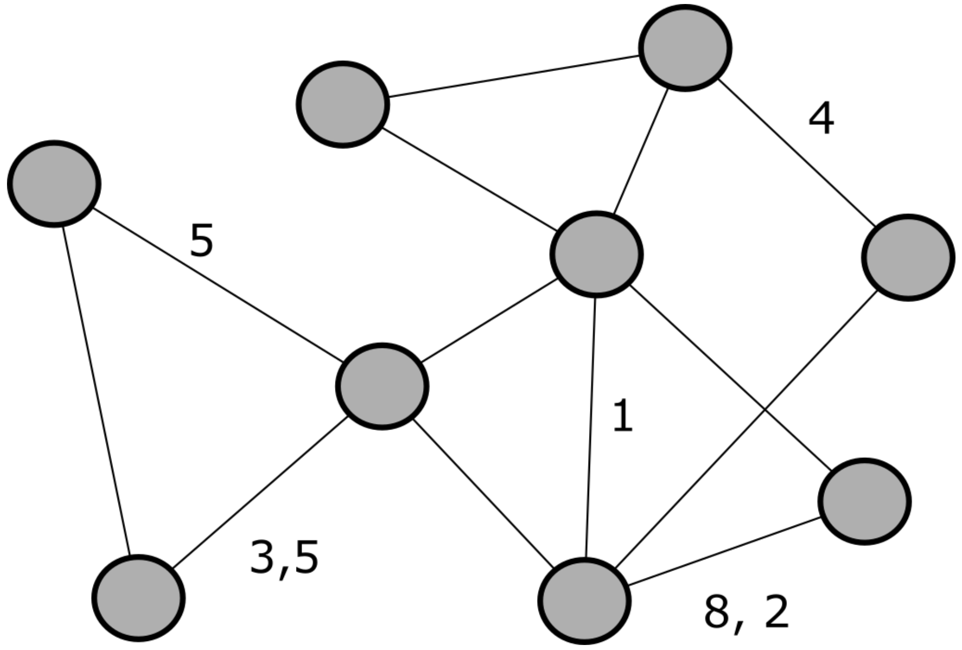


Vertices, **V** and Edges, **E**

What are graphs?

A graph **G** consists of:

- A set of <u>*vertices*</u>, **V**.
- A set of <u>*edges*</u>, **E**, containing pairs **(x, y)** such that both **x** and **y** are in **V**.
- Sometimes, some edges are associated with one or more edge weights.



Section 1:
Graph Representations

Graph Representations: Adjacency Matrix

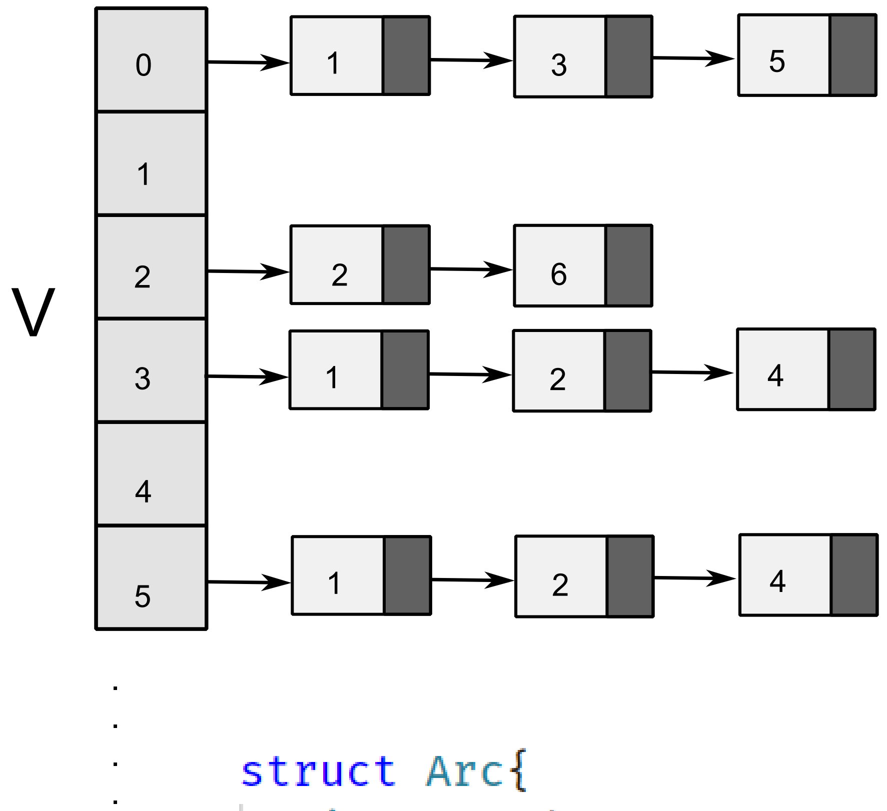
- Represent graph as a **V x V** matrix.
- An entry **[m,n]** in matrix is zero if there is no edge between vertices **m** and **n**.
- An entry **[m, n]** in matrix is non-zero if there is an edge between vertices **m** and **n**.

$$\begin{array}{cc} & V \\ V & \begin{pmatrix} 1 & 3 & 2 & 0 & 3 & 5 & 6 & 5 \\ 0 & 2 & 5 & \boxed{0} & 3 & 3 & 8 & 3 \\ 0 & 5 & 2 & 2 & 5 & 4 & 4 & 4 \\ 0 & 3 & 3 & 6 & 4 & 2 & 2 & 5 \\ 1 & 5 & 7 & 3 & 3 & 6 & 5 & 9 \\ 3 & 1 & 4 & 6 & \boxed{7} & 8 & 19 & 0 \\ 4 & 9 & 6 & 5 & 8 & 5 & 5 & 0 \\ 6 & 0 & 3 & 6 & 3 & 5 & 4 & 2 \end{pmatrix} \end{array}$$

```
const size_t NUM_VERTICES = 10;

size_t Graph[NUM_VERTICES][NUM_VERTICES];
```

Graph Representations: Adjacency List

- Array of **V** linked lists.
- **N**th linked list contains vertices adjacent to vertex **N**.
- In practice, this often takes the form of a vector-of-vectors.



```
struct Arc{
  size_t node;
  unsigned weight;
};
```

```
std::vector<std::vector<Arc>> Graph;
```

Graph Representations: Edge List (Arc List)

- Literally, a list of all edges in the graph.

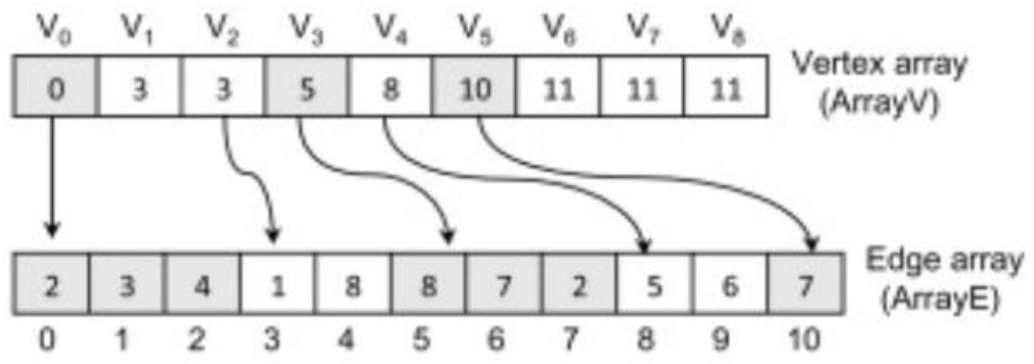
index	start node id	target node id
0	2	3
1	3	4
2	4	1
3	4	5
4	5	4
6	1	5

```
struct Graph {
  size_t node_count;
  std::vector<size_t> tail;
  std::vector<size_t> head;
  std::vector<unsigned> geo_distance;
  std::vector<unsigned> travel_time;
  std::vector<unsigned> speed;
}
```

https://github.com/Project-OSRM/osrm-backend/wiki/Processing-Flow
https://github.com/RoutingKit/RoutingKit/blob/master/doc/SupportFunctions.md

Graph Representations: Adjacency Array

- “A better representation than edge lists for graph traversal”
  - RoutingKit docs



```
struct Graph{
  std::vector<size_t> first_adjacent_vertex;
  std::vector<size_t> head;
  std::vector<unsigned> geo_distance;
  std::vector<unsigned> travel_time;
  std::vector<unsigned> speed;
};
```

```
for(unsigned xy=first_out[x]; xy<first_out[x+1]; ++xy){
  unsigned y = head[xy];
  // xy is the arc from node x to node y
}
```

Source:
https://github.com/RoutingKit/RoutingKit/blob/master/doc/SupportFunctions.md
https://www.sciencedirect.com/topics/computer-science/adjacency-list

Graph Representations: Doubly Connected Edge List (DCEL)

- Used for planar embeddings of graphs.
- Stores edges, vertices and ***faces*** of the graph.
- Provides *efficient manipulation* of edges, vertices and faces.
- Used in Computer Graphics.

Geometric Data Structures

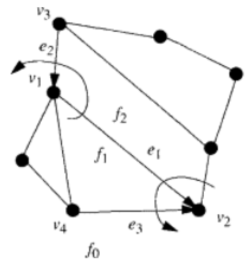
Michael T. Goodrich, Kumar Ramaiyer, in Handbook of Computational Geometry, 2000

2.1 The Doubly Connected Edge List (DCEL)

Muller and Preparata [50,57] designed a PSLG representation, which they called the doubly-connected edge list (or DCEL). The DCEL for a PSLG $G = (V, E, F)$ has a collection of edge nodes. This representation treats each edge as a directed edge; hence, it imposes an orientation on each edge. Each edge node $e = (v_a, v_b)$ is a structure consisting of six fields:

- V_o, representing the origin vertex (v_a),
- V_d, representing the destination vertex (v_b),
- F_l, representing the left face as we traverse on e from V_o, to V_d,
- F_r, representing the right face as we traverse on e from V_o, to V_d,
- CCW_o, representing the counter-clockwise successor of e around V_o, and
- CCW_d, representing the counter-clockwise successor of e around V_d.

The Figure 1 shows the DCEL representation of the edge e_1 in the subdivision given.

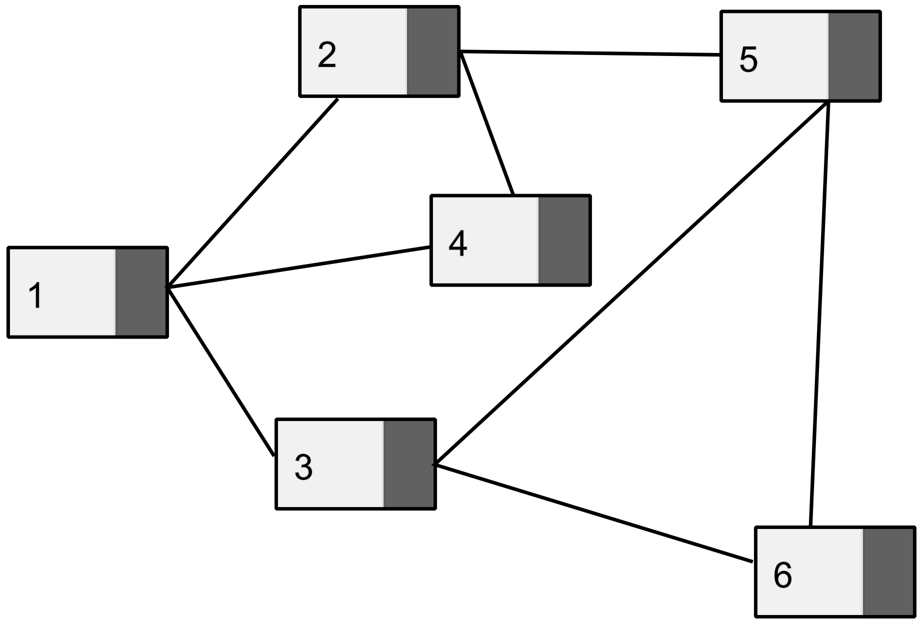


	V_o	V_d	F_l	F_r	CCW_o	CCW_d
e_1	v_1	v_2	f_2	f_1	e_2	e_3
e_2	v_3	v_1	f_2	f_0		
e_3	v_4	v_2	f_1	f_0		

Source:
https://www.sciencedirect.com/topics/computer-science/graph-representation

Graph Representations: Pointer-and-Struct

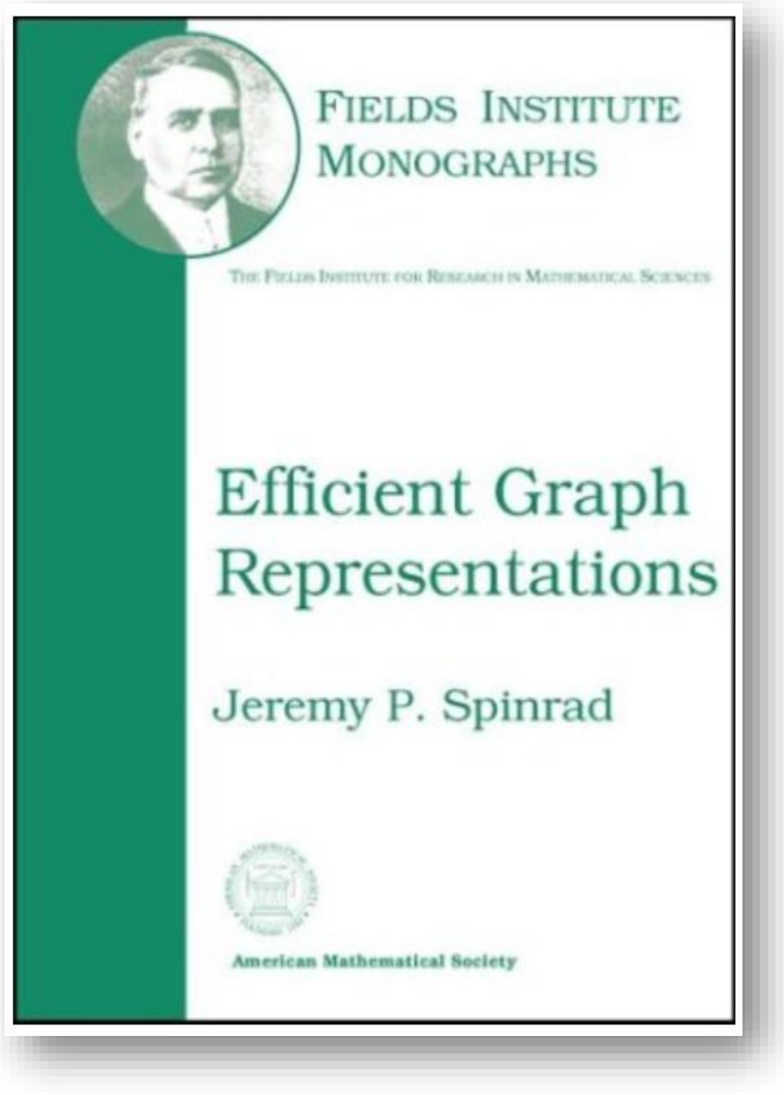
- Each vertex is a struct.
- Each vertex structure holds pointers to adjacent vertices in the graph.



```
struct Node {
  int node_ID;
  std::vector<Node*> out_edges;
};

std::vector<Node> Graph;
```

Source:
https://www.sciencedirect.com/topics/computer-science/graph-representation

Graph Representations: The Book



FIELDS INSTITUTE MONOGRAPHS

THE FIELDS INSTITUTE FOR RESEARCH IN MATHEMATICAL SCIENCES

Efficient Graph Representations

Jeremy P. Spinrad

American Mathematical Society

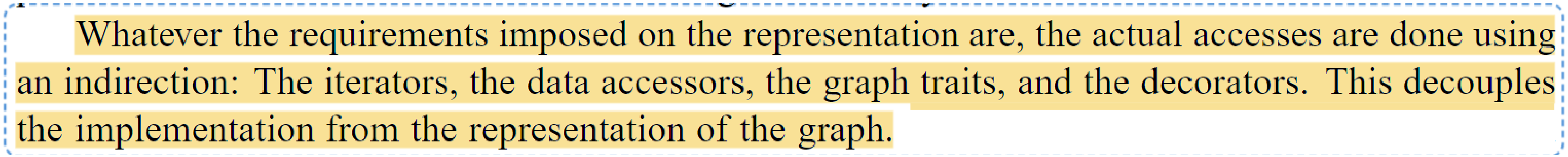
Source: https://bookstore.ams.org/fim-19

Questions?

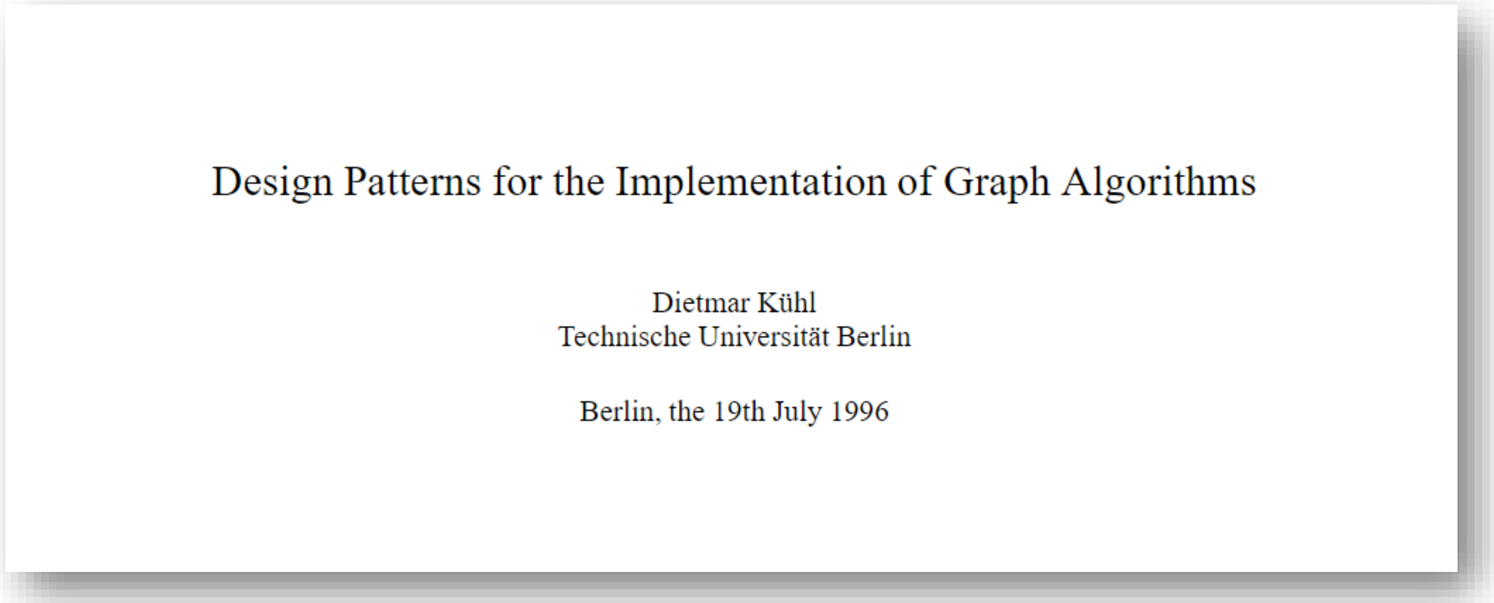
We’ll talk about graph libraries next.

Section 2:
Graph Libraries

The Design of Graph Libraries



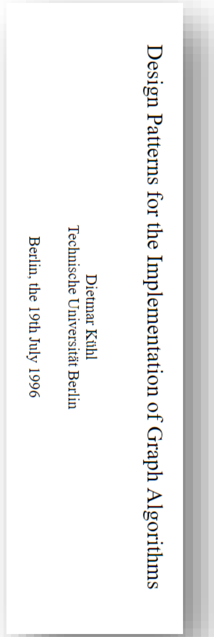
Whatever the requirements imposed on the representation are, the actual accesses are done using an indirection: The iterators, the data accessors, the graph traits, and the decorators. This decouples the implementation from the representation of the graph.



Design Patterns for the Implementation of Graph Algorithms

Dietmar Kühl
Technische Universität Berlin

Berlin, the 19th July 1996


Design Patterns for the Implementation of Graph Algorithms

Dietmar Kühl
Technische Universität Berlin

Berlin, the 19th July 1996

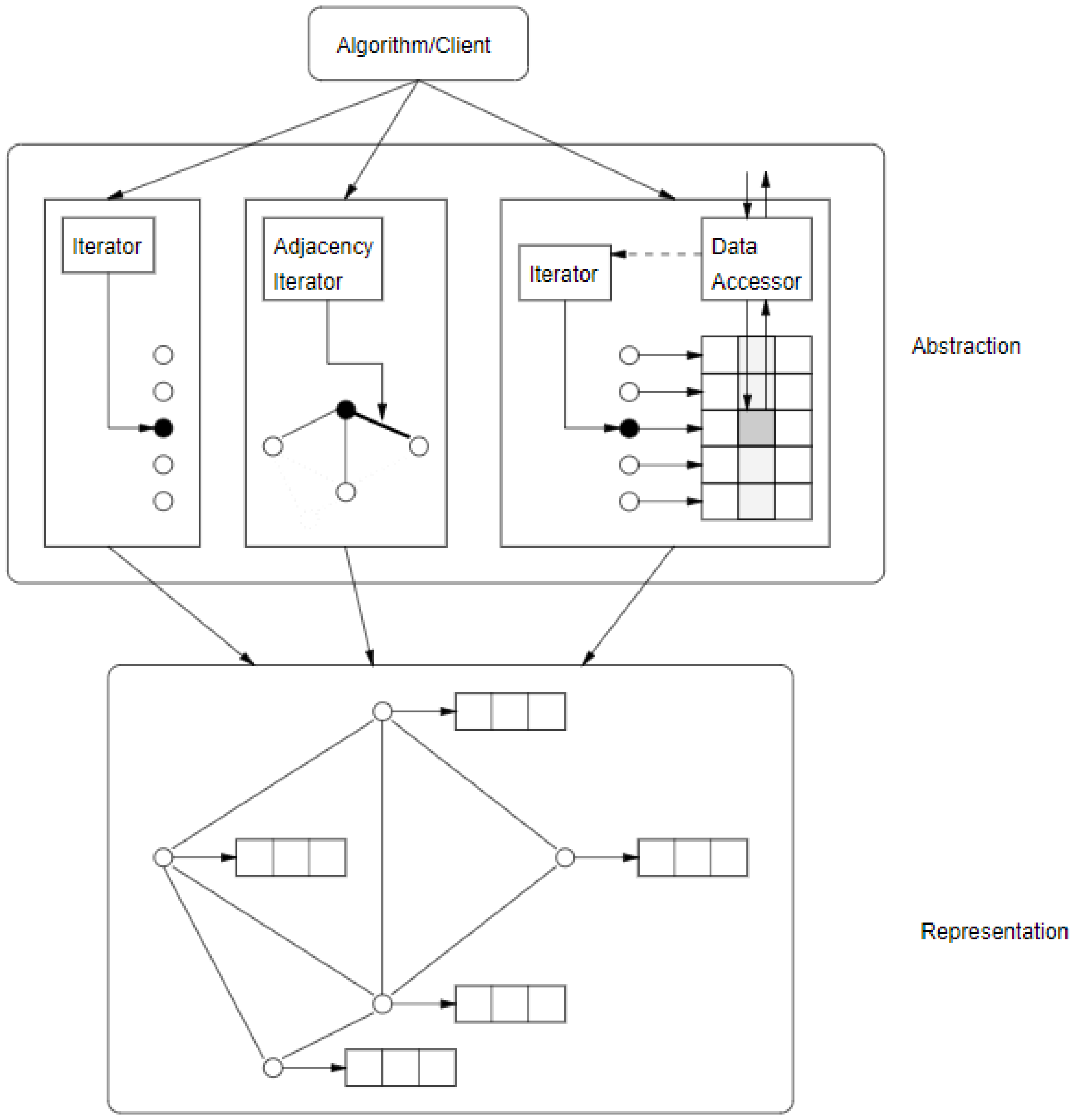
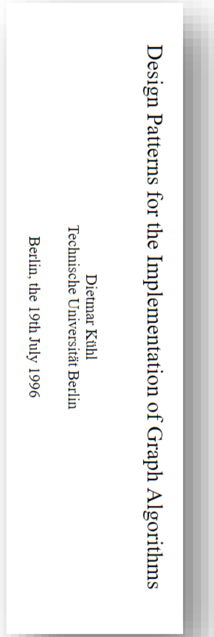


Figure 4.1: Components of the abstraction from the representation

A Graph Library


Design Patterns for the Implementation of Graph Algorithms

Dietmar Kühl
Technische Universität Berlin

Berlin, the 19th July 1996

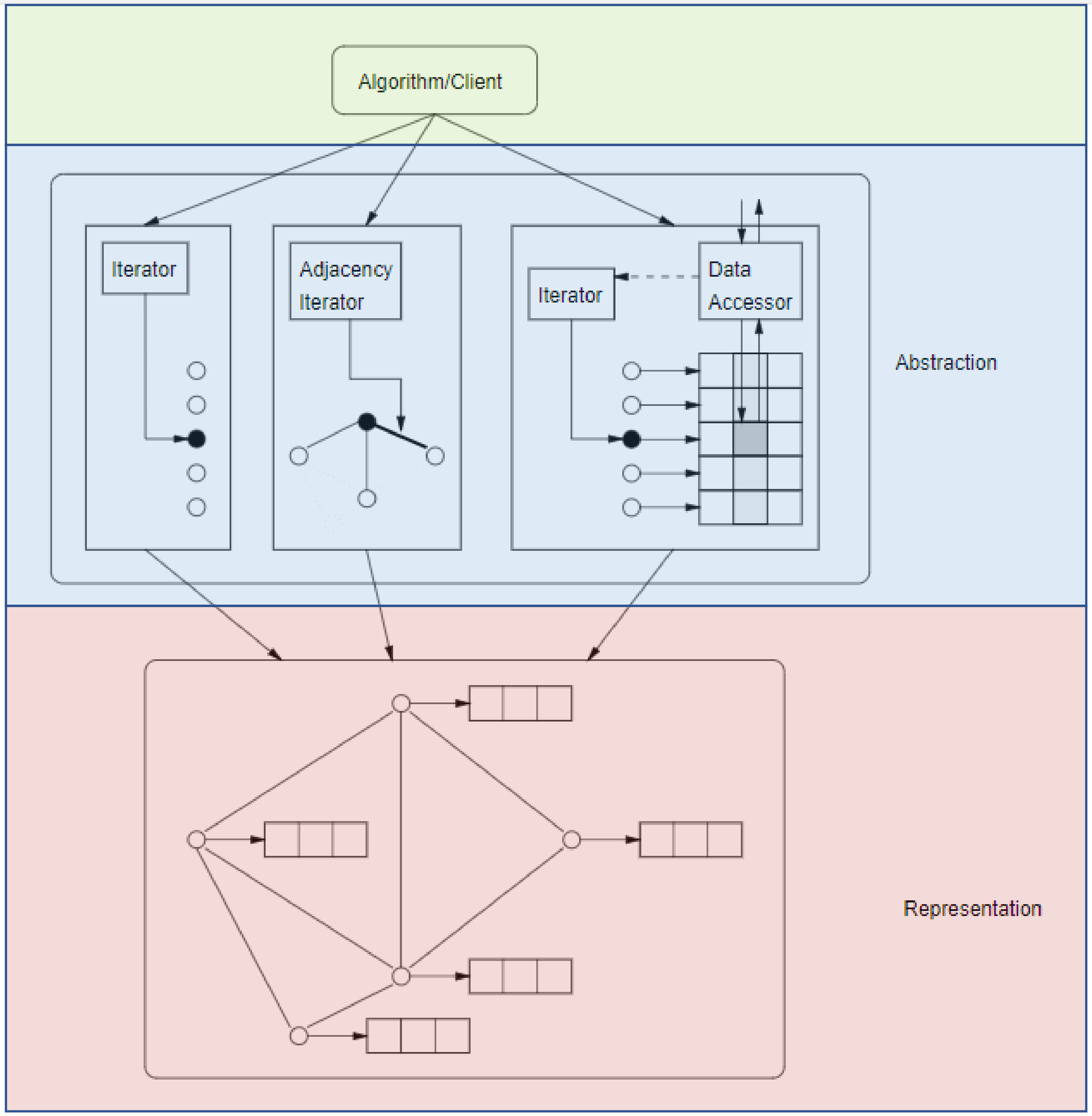
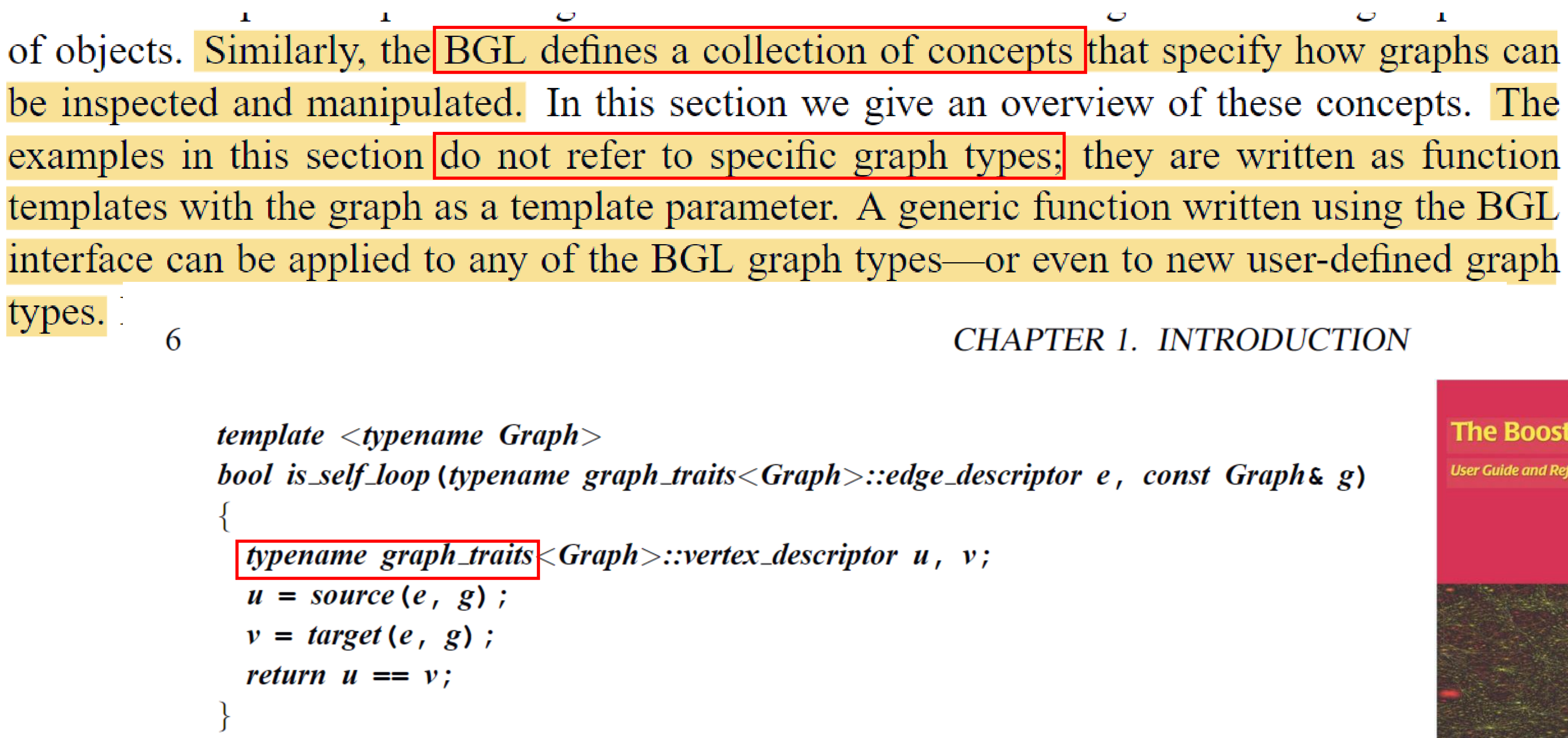


Figure 4.1: Components of the abstraction from the representation

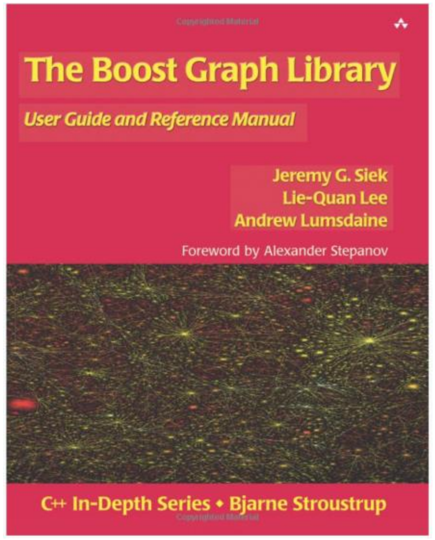
The Design of C++ Graph Libraries: Boost Graph Library



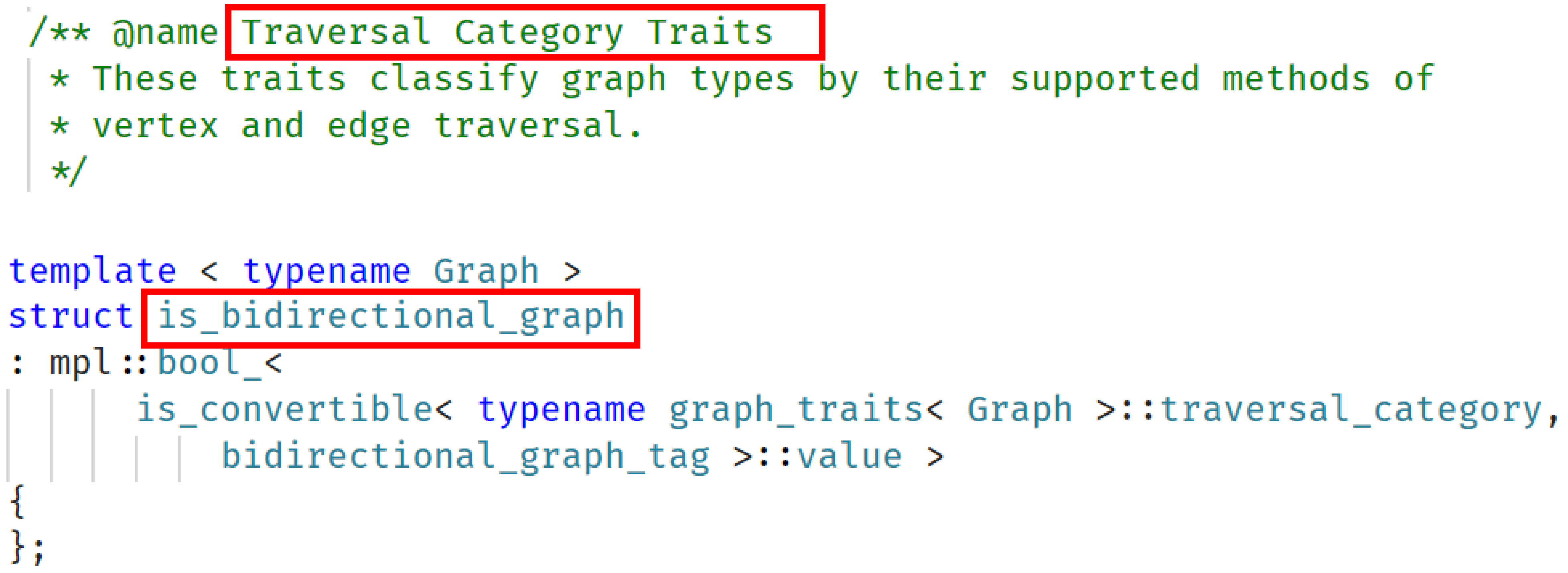
of objects. Similarly, the BGL defines a collection of concepts that specify how graphs can be inspected and manipulated. In this section we give an overview of these concepts. The examples in this section do not refer to specific graph types; they are written as function templates with the graph as a template parameter. A generic function written using the BGL interface can be applied to any of the BGL graph types—or even to new user-defined graph types.

6 CHAPTER 1. INTRODUCTION

```
template <typename Graph>
bool is_self_loop(typename graph_traits<Graph>::edge_descriptor e, const Graph& g)
{
  typename graph_traits<Graph>::vertex_descriptor u, v;
  u = source(e, g);
  v = target(e, g);
  return u == v;
}
```


The Boost Graph Library
User Guide and Reference Manual
Jeremy G. Siek
Lie-Quan Lee
Andrew Lumsdaine
Foreword by Alexander Stepanov
C++ In-Depth Series • Bjarne Stroustrup

The Design of C++ Graph Libraries: Boost Graph Library

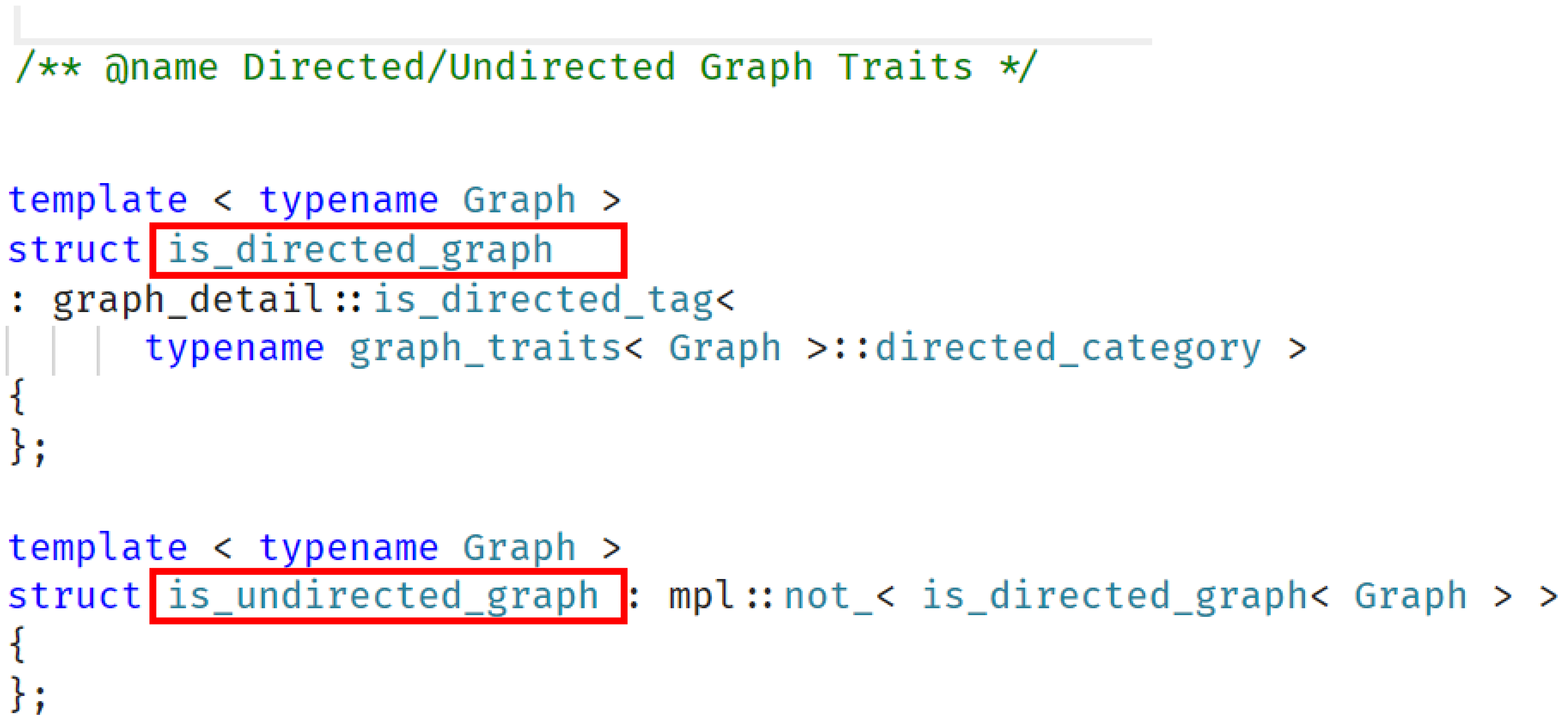


```
/** @name Traversal Category Traits
 * These traits classify graph types by their supported methods of
 * vertex and edge traversal.
 */

template < typename Graph >
struct is_bidirectional_graph
: mpl::bool_<
      is_convertible< typename graph_traits< Graph >::traversal_category,
          bidirectional_graph_tag >::value >
{
};
```

The Design of C++ Graph Libraries: Boost Graph Library

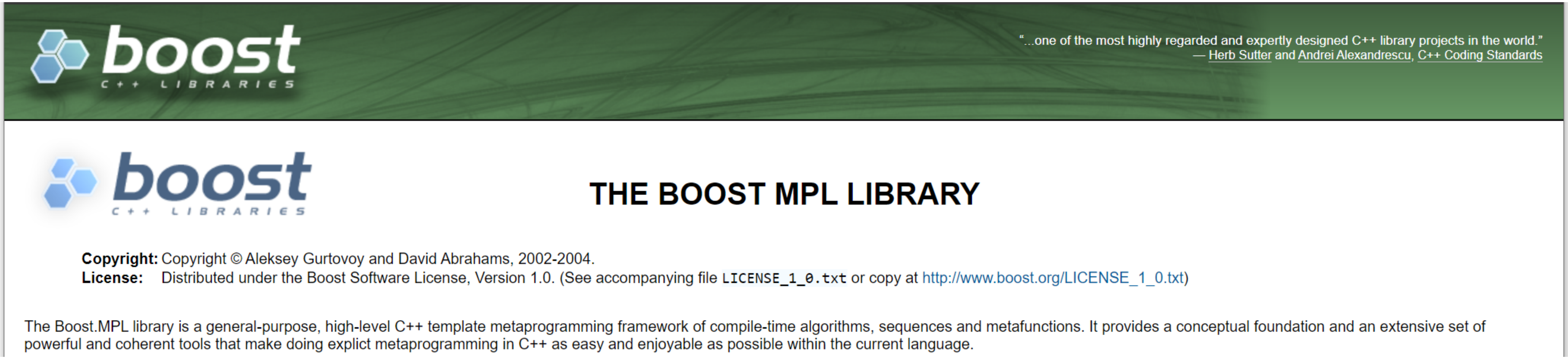
```
/** @name Traversal Category Traits
 * These traits classify graph types by their supported methods of
 * vertex and edge traversal.
 */

template < typename Graph >
struct is_adjacency_matrix
: mpl::bool_<
      is_convertible< typename graph_traits< Graph >::traversal_category,
          adjacency_matrix_tag >::value >
{
};
```

The Design of C++ Graph Libraries: Boost Graph Library



```cpp
/** @name Directed/Undirected Graph Traits */

template < typename Graph >
struct is_directed_graph
: graph_detail::is_directed_tag<
          typename graph_traits< Graph >::directed_category >
{
};

template < typename Graph >
struct is_undirected_graph : mpl::not_< is_directed_graph< Graph > >
{
};
```

The Design of C++ Graph Libraries: Boost Graph Library

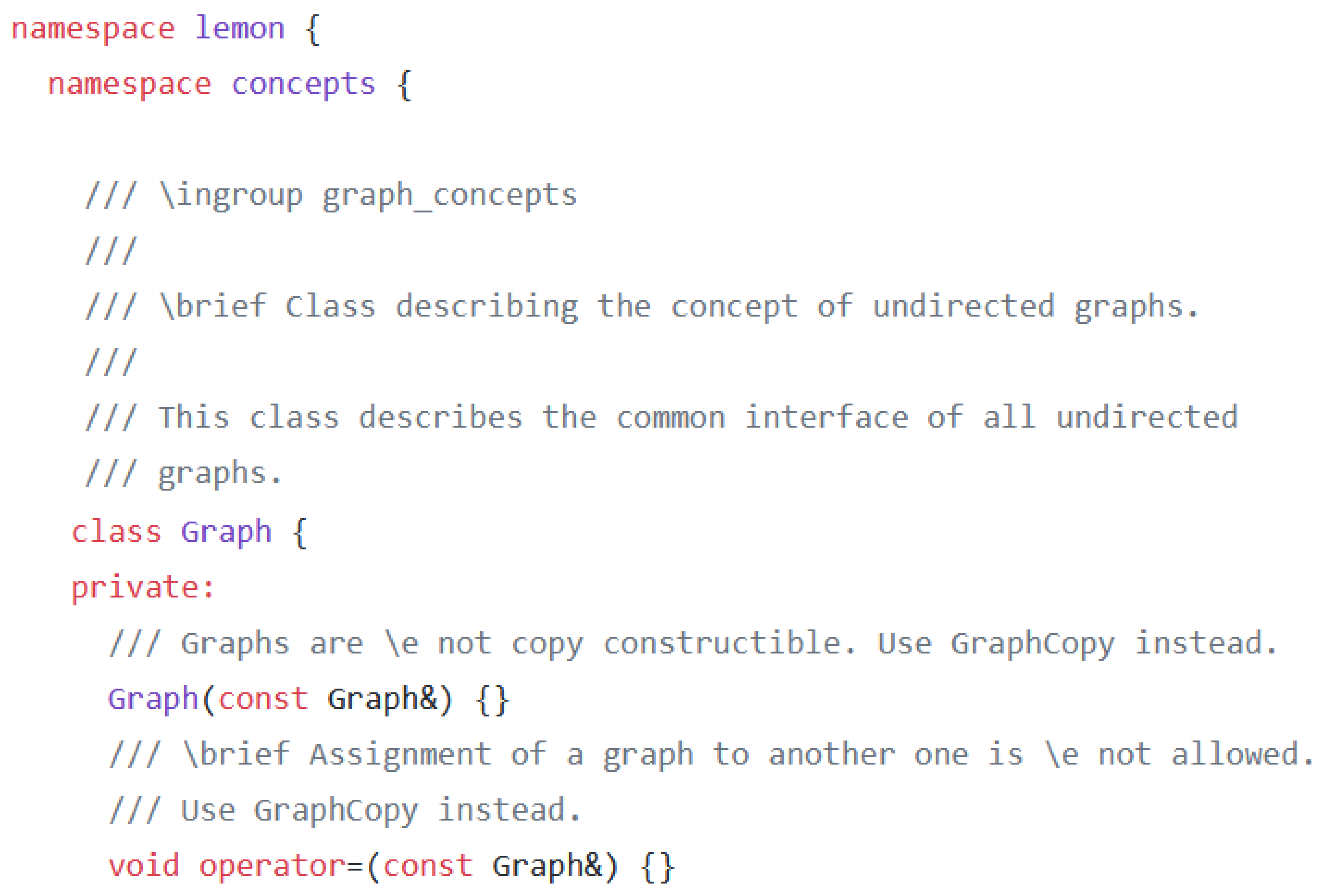


"...one of the most highly regarded and expertly designed C++ library projects in the world."
— Herb Sutter and Andrei Alexandrescu, C++ Coding Standards

THE BOOST MPL LIBRARY

The Boost.MPL library is a general-purpose, high-level C++ template metaprogramming framework of compile-time algorithms, sequences and metafunctions. It provides a conceptual foundation and an extensive set of powerful and coherent tools that make doing explict metaprogramming in C++ as easy and enjoyable as possible within the current language.

The Design of C++ Graph Libraries: LEMON



```cpp
namespace lemon {
  namespace concepts {

    /// \ingroup graph_concepts
    ///
    /// \brief Class describing the concept of undirected graphs.
    ///
    /// This class describes the common interface of all undirected
    /// graphs.
   class Graph {
   private:
     /// Graphs are \e not copy constructible. Use GraphCopy instead.
     Graph(const Graph&) {}
     /// \brief Assignment of a graph to another one is \e not allowed.
     /// Use GraphCopy instead.
     void operator=(const Graph&) {}
```

https://github.com/tpet/lemon/blob/master/lemon/concept_check.h

The Design of C++ Graph Libraries: LEMON

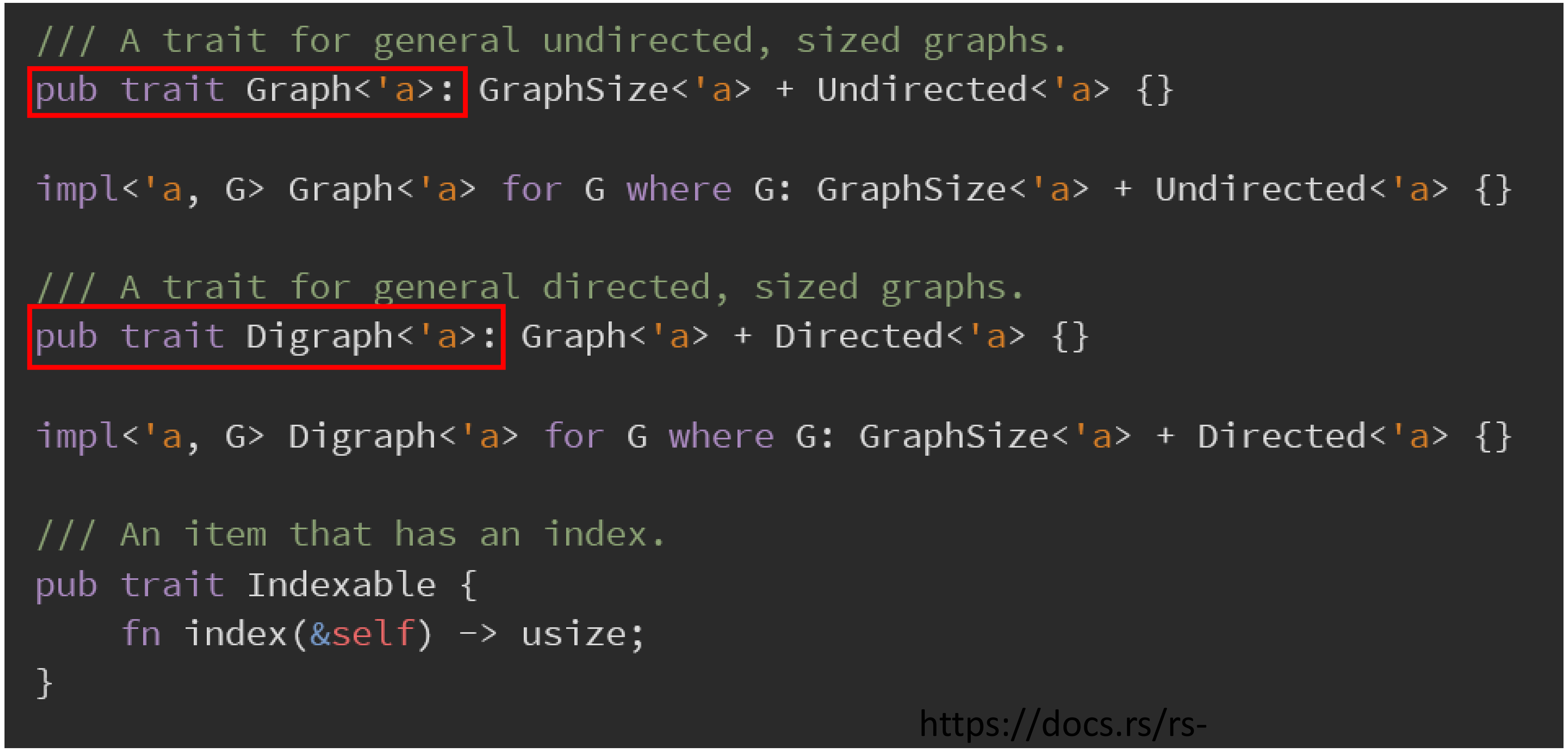
```
template <typename _Graph>
struct Constraints {
  void constraints() {
    checkConcept<BaseGraphComponent, _Graph>();
    checkConcept<IterableGraphComponent<>, _Graph>();
    checkConcept<IDableGraphComponent<>, _Graph>();
    checkConcept<MappableGraphComponent<>, _Graph>();
  }
};
```

```
  ///\e
  template <class Concept>
  inline void function_requires()
  {
#if !defined(NDEBUG)
    void (Concept::*x)() = & Concept::constraints;
    ::lemon::ignore_unused_variable_warning(x);
#endif
  }

  ///\e
  template <typename Concept, typename Type>
  inline void checkConcept() {
#if !defined(NDEBUG)
    typedef typename Concept::template Constraints<Type> ConceptCheck;
    void (ConceptCheck::*x)() = & ConceptCheck::constraints;
    ::lemon::ignore_unused_variable_warning(x);
#endif
  }
```

https://github.com/tpet/lemon/blob/master/lemon/concept_check.h

The Design of Rust Graph Libraries: (rs_graph)

- Traits are first class citizens in Rust!!

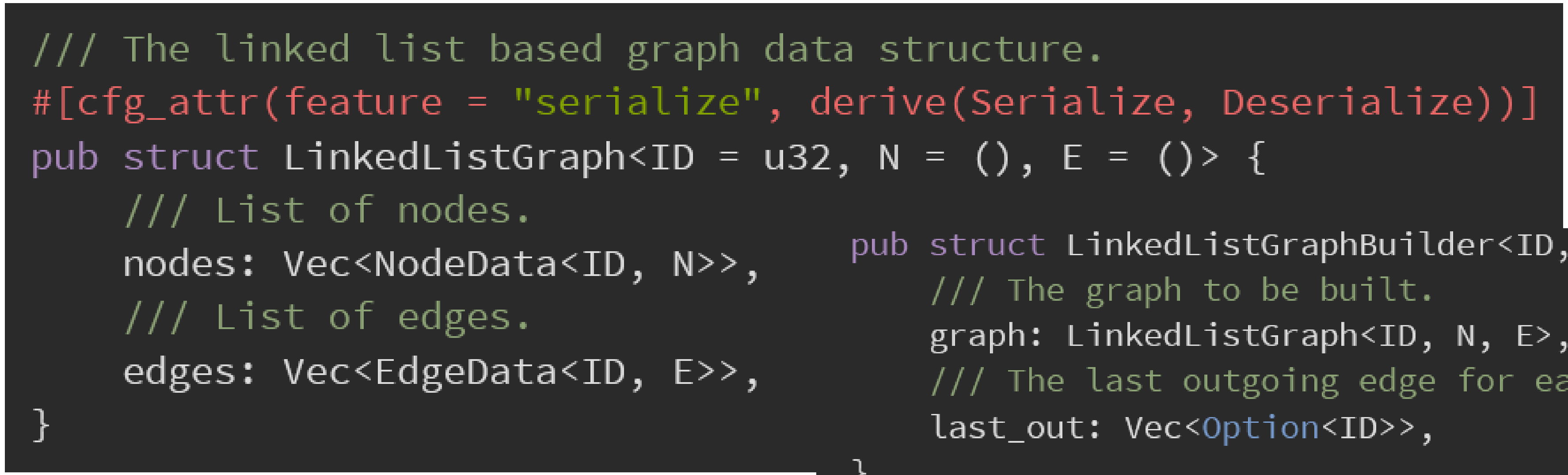


```rust
/// A trait for general undirected, sized graphs.
pub trait Graph<'a>: GraphSize<'a> + Undirected<'a> {}

impl<'a, G> Graph<'a> for G where G: GraphSize<'a> + Undirected<'a> {}

/// A trait for general directed, sized graphs.
pub trait Digraph<'a>: Graph<'a> + Directed<'a> {}

impl<'a, G> Digraph<'a> for G where G: GraphSize<'a> + Directed<'a> {}

/// An item that has an index.
pub trait Indexable {
    fn index(&self) -> usize;
}
```

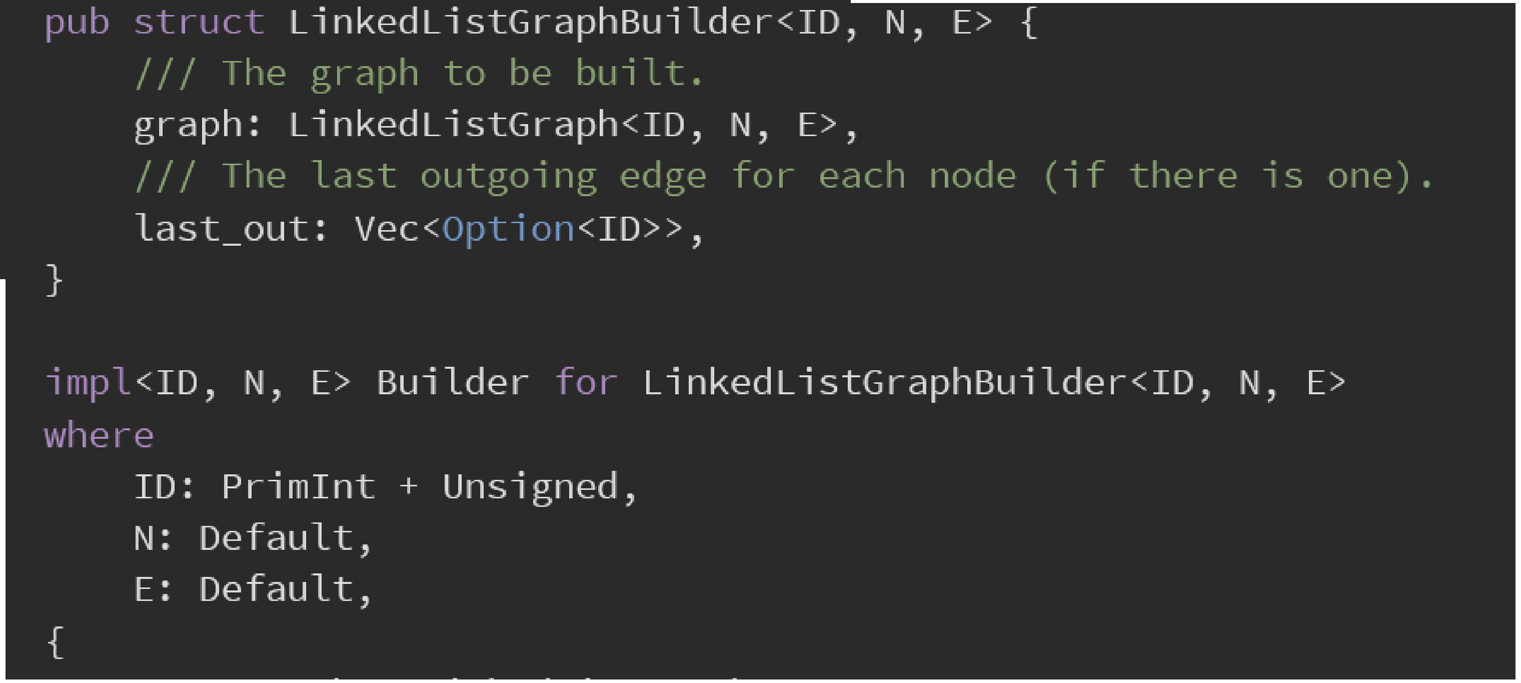
https://docs.rs/rs-graph/0.19.2/src/rs_graph/linkedlistgraph.rs.html

The Design of Rust Graph Libraries: (rs_graph)

- Implementing graph traits to get concrete graph types:



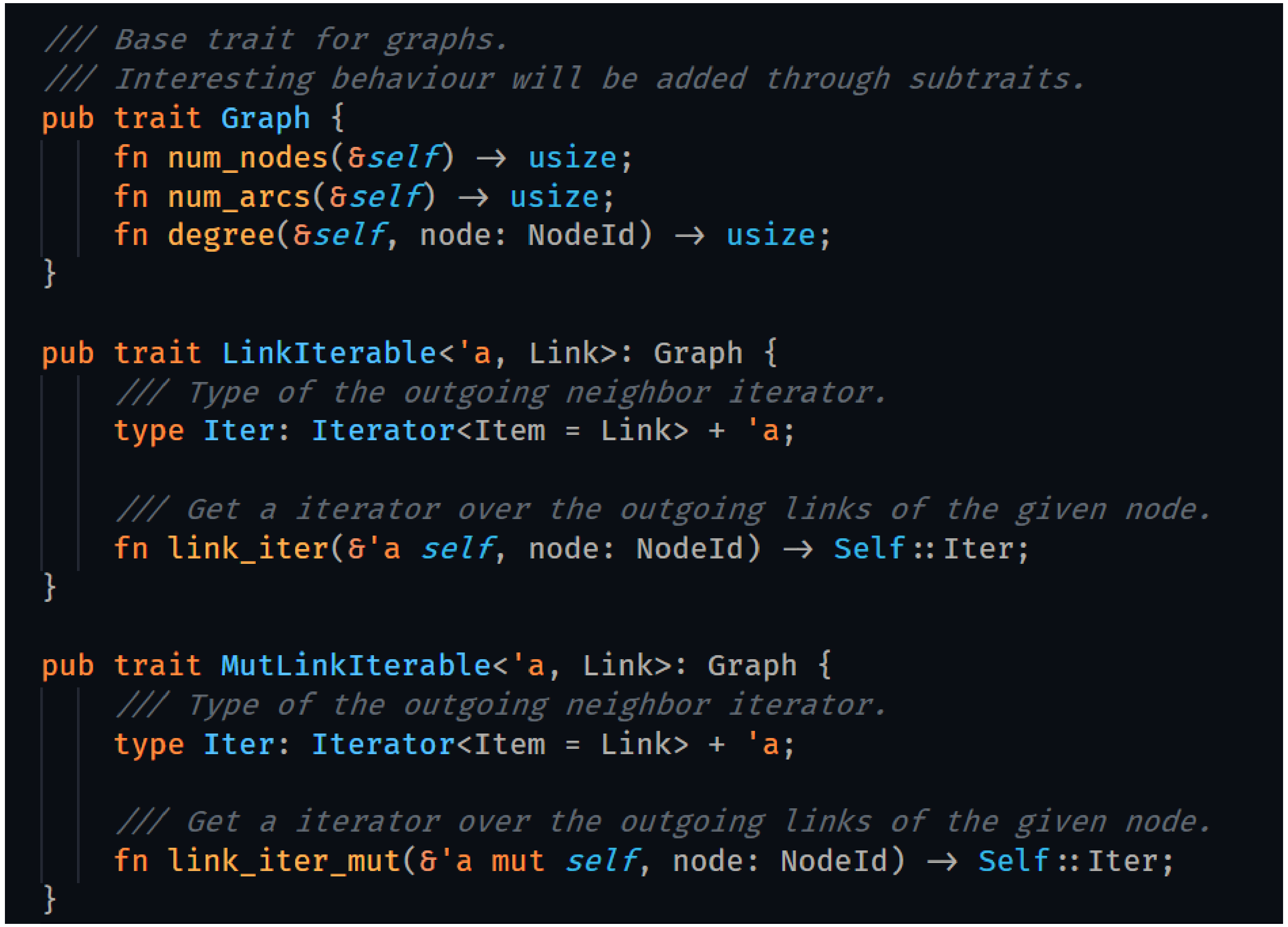
```rust
/// The linked list based graph data structure.
#[cfg_attr(feature = "serialize", derive(Serialize, Deserialize))]
pub struct LinkedListGraph<ID = u32, N = (), E = ()> {
    /// List of nodes.
    nodes: Vec<NodeData<ID, N>>,
    /// List of edges.
    edges: Vec<EdgeData<ID, E>>,
}
```



```rust
pub struct LinkedListGraphBuilder<ID, N, E> {
    /// The graph to be built.
    graph: LinkedListGraph<ID, N, E>,
    /// The last outgoing edge for each node (if there is one).
    last_out: Vec<Option<ID>>,
}

impl<ID, N, E> Builder for LinkedListGraphBuilder<ID, N, E>
where
    ID: PrimInt + Unsigned,
    N: Default,
    E: Default,
{
```

The Design of Rust Graph Libraries: (rust_road_router)

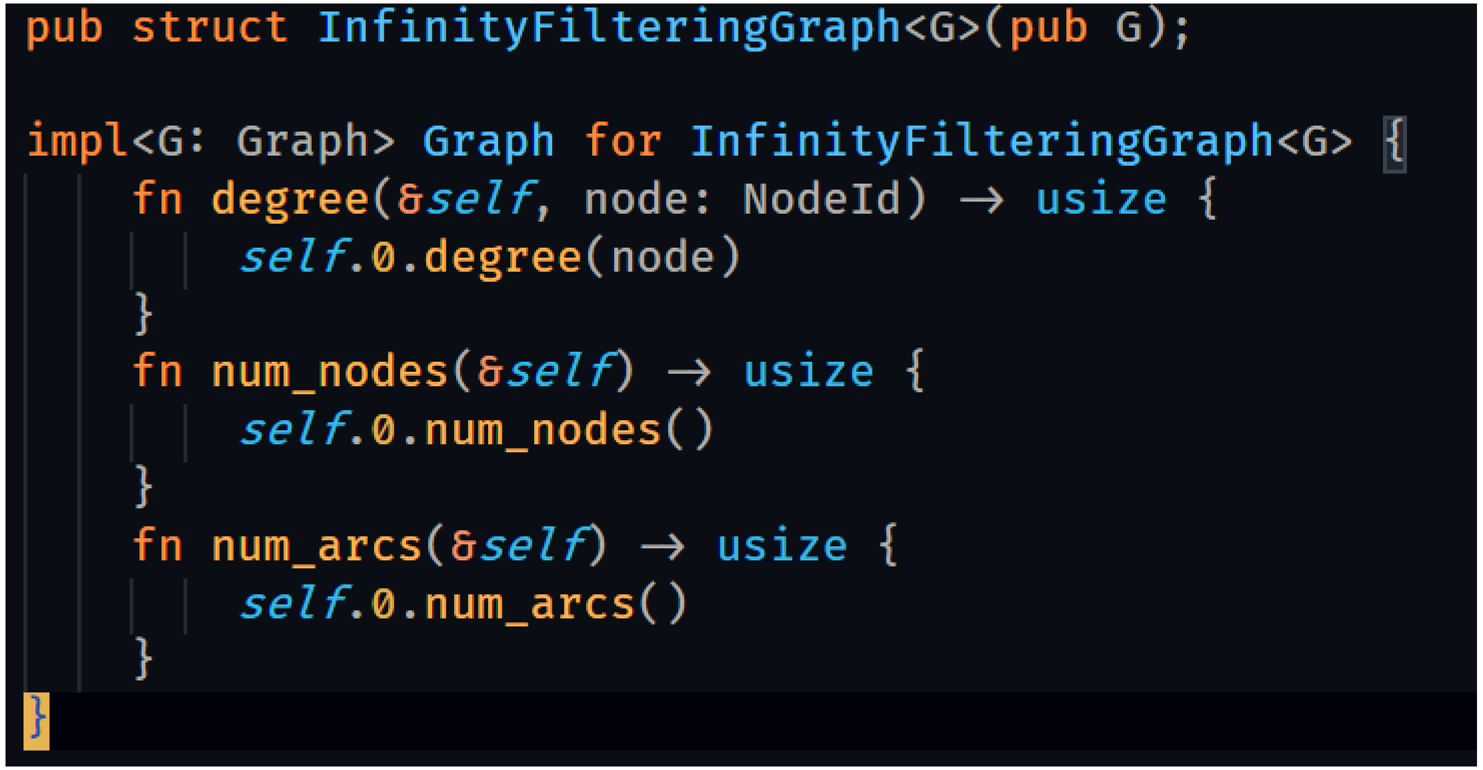
- Graph Traits:



```rust
/// Base trait for graphs.
/// Interesting behaviour will be added through subtraits.
pub trait Graph {
    fn num_nodes(&self) -> usize;
    fn num_arcs(&self) -> usize;
    fn degree(&self, node: NodeId) -> usize;
}

pub trait LinkIterable<'a, Link>: Graph {
    /// Type of the outgoing neighbor iterator.
    type Iter: Iterator<Item = Link> + 'a;

    /// Get a iterator over the outgoing links of the given node.
    fn link_iter(&'a self, node: NodeId) -> Self::Iter;
}

pub trait MutLinkIterable<'a, Link>: Graph {
    /// Type of the outgoing neighbor iterator.
    type Iter: Iterator<Item = Link> + 'a;

    /// Get a iterator over the outgoing links of the given node.
    fn link_iter_mut(&'a mut self, node: NodeId) -> Self::Iter;
}
```

The Design of Rust Graph Libraries: (rust_road_router)

- Graph Implementation:



```rust
pub struct InfinityFilteringGraph<G>(pub G);

impl<G: Graph> Graph for InfinityFilteringGraph<G> {
    fn degree(&self, node: NodeId) -> usize {
        self.0.degree(node)
    }
    fn num_nodes(&self) -> usize {
        self.0.num_nodes()
    }
    fn num_arcs(&self) -> usize {
        self.0.num_arcs()
    }
}
```

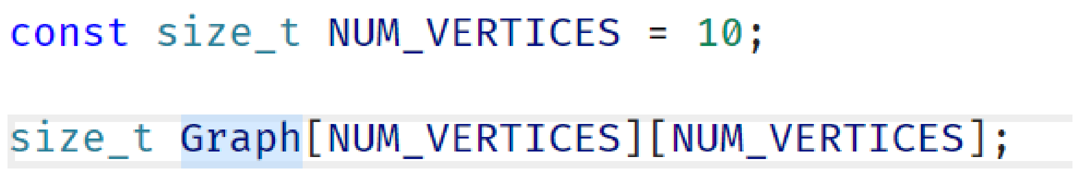
Comparing the C++ and Rust Libraries: Graph Interfaces

- Traits being first class citizens in Rust saves us from metaprogramming tricks and Boost MPL!

So, are generic graph interfaces hard in Rust?
No.

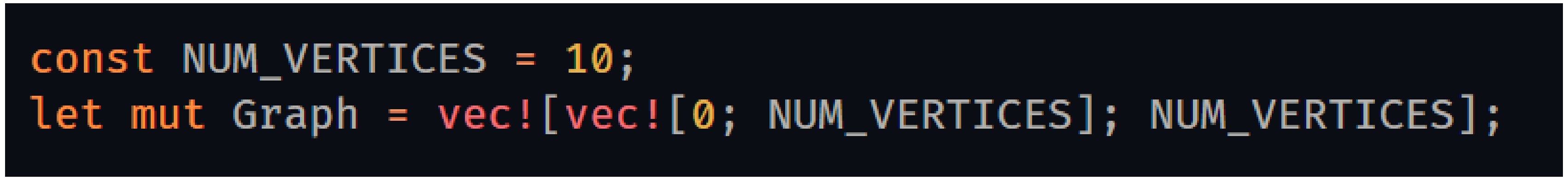
Comparing the C++ and Rust Libraries: Graph Representations – Adj. Matrix

C++



```cpp
const size_t NUM_VERTICES = 10;

size_t Graph[NUM_VERTICES][NUM_VERTICES];
```

Rust



```rust
const NUM_VERTICES = 10;
let mut Graph = vec![vec![0; NUM_VERTICES]; NUM_VERTICES];
```

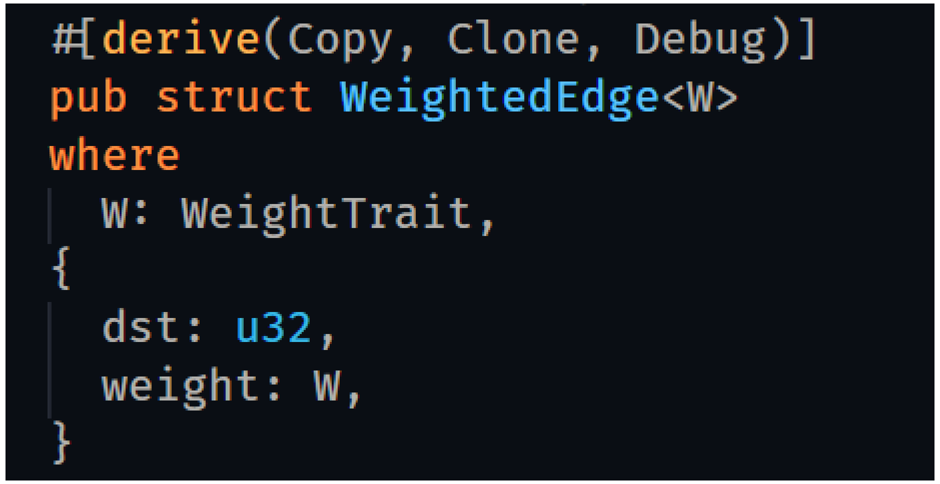
Comparing the C++ and Rust Libraries: Graph Representations – Adj. List

C++

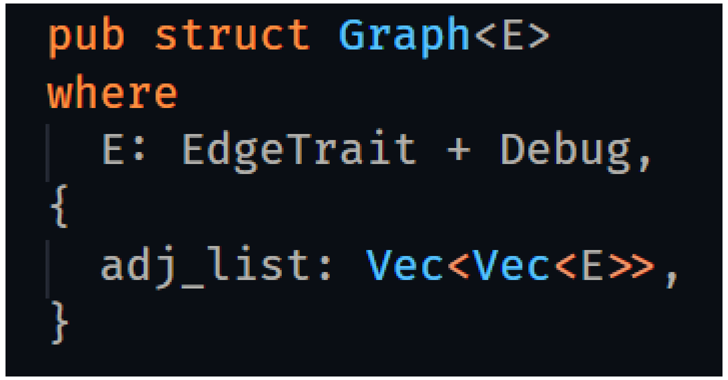
```cpp
struct Arc{
  size_t node;
  unsigned weight;
};
```

```cpp
std::vector<std::vector<Arc>> Graph;
```

Rust



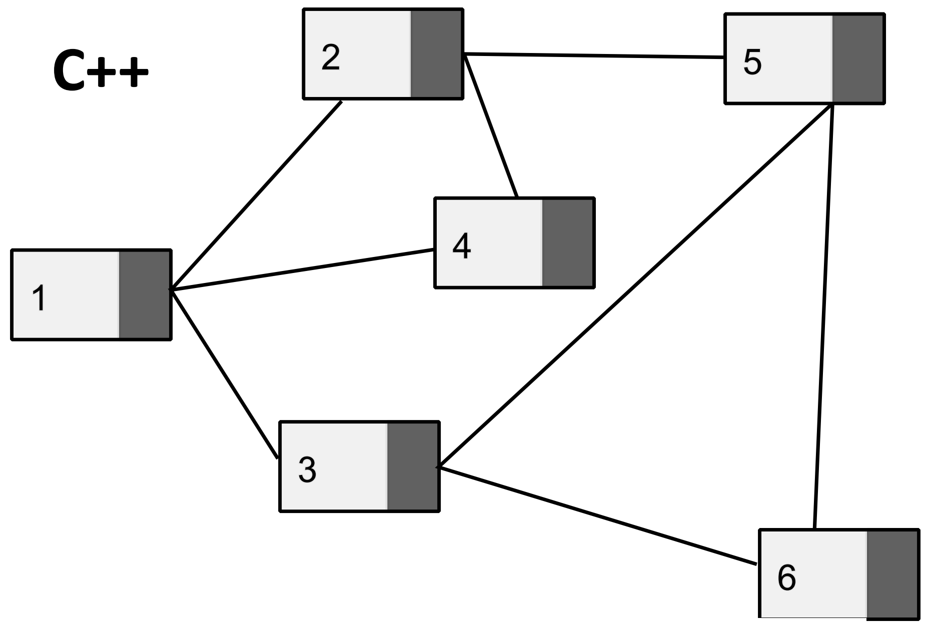
```rust
#[derive(Copy, Clone, Debug)]
pub struct WeightedEdge<W>
where
  W: WeightTrait,
{
  dst: u32,
  weight: W,
}
```



```rust
pub struct Graph<E>
where
  E: EdgeTrait + Debug,
{
  adj_list: Vec<Vec<E>>,
}
```

Comparing the C++ and Rust Libraries: Graph Representations – Pointer & Struct

C++



```
struct Node {
  int node_ID;
  std::vector<Node*> out_edges;
};

std::vector<Node> Graph;
```

Rust

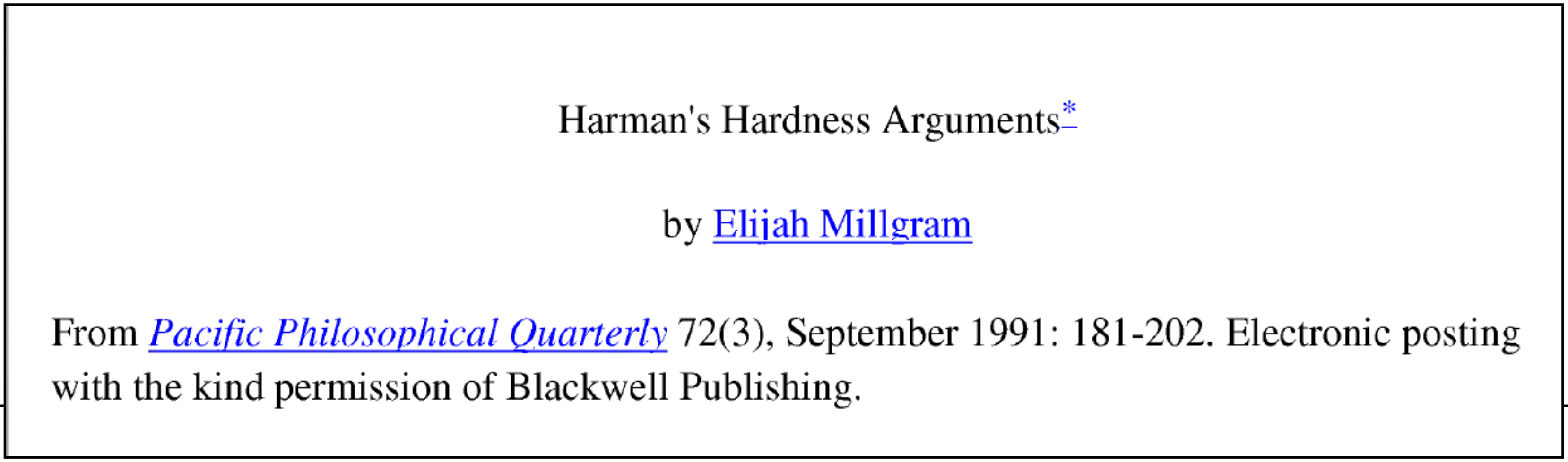
- ???
- Do you need it?
- Do you ***really*** need it?
- Use ‘unsafe’ and write tests.

Conclusion (?)

- So, are generic graph interfaces hard in Rust?
  No.
- So, are some graph representations hard in Rust?
  No, but pointer & struct graphs need ‘unsafe’ and tests.
- But you shouldn’t be using pointer & struct graphs anyway.
- For representations that you _should_ use, Rust code looks a lot like C++ code.

Are Graphs Hard in Rust?

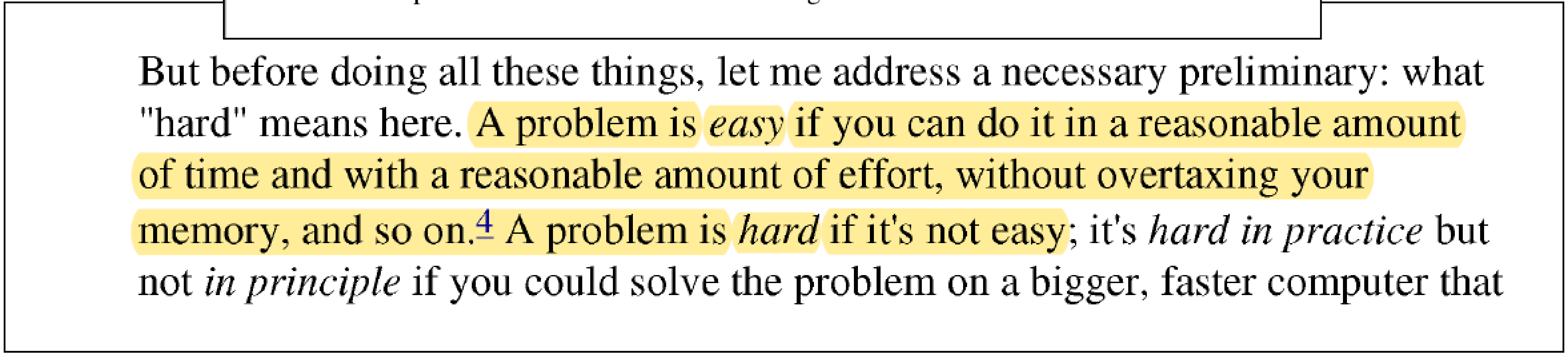
Are Graphs Hard in Rust?



Harman's Hardness Arguments[*]

by Elijah Millgram

From *Pacific Philosophical Quarterly* 72(3), September 1991: 181-202. Electronic posting with the kind permission of Blackwell Publishing.



But before doing all these things, let me address a necessary preliminary: what "hard" means here. A problem is *easy* if you can do it in a reasonable amount of time and with a reasonable amount of effort, without overtaxing your memory, and so on.[4] A problem is *hard* if it's not easy; it's *hard in practice* but not *in principle* if you could solve the problem on a bigger, faster computer that

Thank you!

- Tim Zeitz
- Dev Purandare
- Daniel Bittman
- Peter Wilcox
- Lawrence Lawson
- Soham

American Red Cross

Children's Cancer Research Fund®

Thank you for your time.

@PayasR

(Twitter, GitHub, LinkedIn)

Bonus Slide: What about C++ Concepts?

```
BOOST_concept(Graph, (G))
{
    typedef typename graph_traits< G >::vertex_descriptor vertex_descriptor;
    typedef typename graph_traits< G >::edge_descriptor edge_descriptor;
    typedef typename graph_traits< G >::directed_category directed_category;
    typedef typename graph_traits< G >::edge_parallel_category
        edge_parallel_category;
    typedef typename graph_traits< G >::traversal_category traversal_category;

    BOOST_CONCEPT_USAGE(Graph)
    {
        BOOST_CONCEPT_ASSERT((DefaultConstructible< vertex_descriptor >));
        BOOST_CONCEPT_ASSERT((EqualityComparable< vertex_descriptor >));
        BOOST_CONCEPT_ASSERT((Assignable< vertex_descriptor >));
    }
    G g;
};
```