Boost.Graph tutorial

Richel Bilderbeek

December 11, 2015

Contents

1 Introduction **3**
- 1.1 Code snippets . . . 3
- 1.2 Coding style . . . 4
- 1.3 Feedback . . . 4

2 Building a graph without properties **4**
- 2.1 Creating an empty (directed) graph . . . 5
- 2.2 Creating an empty undirected graph . . . 5
- 2.3 Add a vertex . . . 7
- 2.4 Vertex descriptors . . . 7
- 2.5 Get the vertices . . . 8
- 2.6 Get all vertex descriptors . . . 9
- 2.7 Add an edge . . . 11
- 2.8 boost::add_edge result . . . 12
- 2.9 Getting the edges . . . 12
- 2.10 Edge descriptors . . . 13
- 2.11 Get all edge descriptors . . . 14
- 2.12 Creating K_2, a fully connected graph with two vertices . . . 15

3 Working on a graph without properties **17**
- 3.1 Counting the number of vertices . . . 17
- 3.2 Counting the number of edges . . . 18
- 3.3 Getting the vertices' out degree . . . 19
- 3.4 Storing a graph as a .dot . . . 21

4 Building graphs with built-in properties **22**
- 4.1 Creating an empty graph with named vertices . . . 23
- 4.2 Add a vertex with a name . . . 25
- 4.3 Getting the vertices' names . . . 26
- 4.4 Creating K_2 with named vertices . . . 28
- 4.5 Creating an empty graph with named edges and vertices . . . 30
- 4.6 Adding a named edge . . . 32
- 4.7 Getting the edges' names . . . 34

4.8 Creating K_3 with named edges and vertices 35

5 Working with graphs with named edges and vertices 37
5.1 Check if there exists a vertex with a certain name 38
5.2 Find a vertex by its name 39
5.3 Get a (named) vertex its degree, in degree and out degree 41
5.4 Get a (named) vertex its name from its vertex descriptor 42
5.5 Set a (named) vertex its name from its vertex descriptor 44
5.6 Setting all vertices' names 45
5.7 Clear the edges of a named vertex 46
5.8 Remove a named vertex 48
5.9 Check if there exists an edge with a certain name 49
5.10 Find an edge by its name 51
5.11 Get a (named) edge its name from its edge descriptor 52
5.12 Set a (named) edge its name from its edge descriptor 53
5.13 Removing a named edge 54
5.13.1 Removing the first edge with a certain name 54
5.13.2 Removing the edge between two named vertices 55
5.14 Storing a graph with named vertices as a .dot 57
5.15 Storing a graph with named vertices and edges as a .dot 59

6 Building graphs with custom properties 61
6.1 Create an empty graph with custom vertices 61
6.1.1 Creating the custom vertex class 61
6.1.2 Installing the new property 62
6.1.3 Create the empty graph with custom vertices 63
6.2 Add a custom vertex 64
6.3 Getting the vertices' my_vertexes 64
6.4 Creating K_2 with custom vertices 65
6.5 Create an empty graph with custom edges and vertices 66
6.5.1 Creating the custom edge class 67
6.5.2 Installing the new property 67
6.5.3 Create the empty graph with custom edges and vertices . 69
6.6 Add a custom edge 70
6.7 Creating K_3 with custom edges and vertices 70

7 Measuring simple graphs traits of a graph with custom edges and vertices 73
7.1 Has a my_vertex 73
7.2 Find a vertex with a certain my_vertex 74
7.3 Get a vertex its my_vertex 76
7.4 Set a vertex its my_vertex 77
7.5 Setting all vertices' my_vertex objects 79
7.6 Has a my_edge 80
7.7 Find a my_edge 82
7.8 Get an edge its my_edge 84

7.9 Set an edge its my_edge 85
7.10 Storing a graph with custom vertices as a .dot 86

8 Measuring more complex graphs traits **88**
8.1 Count the number of self-loops 88

9 Misc functions **88**
9.1 Getting a data type as a std::string 88

10 Errors **89**
10.1 Formed reference to void 89
10.2 No matching function for call to 'clear_out_edges' 90
10.3 No matching function for call to 'clear_in_edges' 90

1 Introduction

I needed this tutorial in 2006, when I started experimenting with Boost.Graph. More specifically, I needed a tutorial that:

- Orders concepts chronologically
- Increases complexity gradually
- Shows complete pieces of code

What I had were the book [2] and the Boost.Graph website, both did not satisfy these requirements.

This tutorial is intended to take the reader to the level of understanding the book [2] and the Boost.Graph website require.

The chapters of this tutorial are also like a well-connected graph. To allow for quicker learners to skim chapters, or for beginners looking to find the patterns, some chapters are repetitions of each other (for example, getting an edge its name is very similar to getting a vertex its name).

A pivorial chapter is chapter 5.2, 'Finding the first vertex with a name', as this opens up the door to finding a vertex and manipulating it.

1.1 Code snippets

For every concept, I will show

- the 'do' function: a function that achieves a goal, for example 'create_empty_undirected_graph'
- the 'demo' function: a function that demonstrates how to call the first, for example 'demonstrate_create_empty_undirected_graph'

I enjoy to show concepts by putting those in (long-named) functions. These functions sometimes border the trivial, by, for example, only calling a Boost.Graph function. On the other hand, these functions have more English-sounding names, resulting in demonstration code that is readable.

All coding snippets are taken from compiled C++ code.

1.2 Coding style

I use the coding style from the Core C++ Guidelines. At the time of this writing, the Core C++ Guidelines were still in early development, so I can only hope the conventions I then chose to follow are still Good Ideas.

I prefer to use the keyword auto over doubling the lines of code for using statements. Because the 'do' functions return an explicit data type, these can be used for reference. If you really want to know a type, you can use the 'get_type_name' function (chapter 9.1). On the other hand, I am explicit of which data types I choose: I will prefix the types by thir namespace, so to distinguish between types like 'std::array' and 'boost::array'. Note that the heavily-use 'get' function must reside in the namespace of the graph to work on. In this tutorial, this is in the global namespace. Thus, I will write 'get', instead of 'boost::get', as the latter does not compile.

1.3 Feedback

I have tried hard to strictly follow the style as described above. If you find I deviated from these decisions somewhere, I would be grateful if you'd let know.

2 Building a graph without properties

Boost.Graph is about creating graphs. In this chapter we create the simplest of graphs, in which edges and nodes have no properties (e.g. having a name). We will build:

- An empty (directed) graph, which is the default type: see chapter 2.1
- An empty (undirected) graph: see chapter 2.2
- K_2, an undirected graph with two vertices and one edge, chapter 2.12

In the process, some basic (sometimes bordering trivial) functions are shown:

- Adding a vertex: see chapter 2.3
- Getting all vertices: see chapter 2.5
- Getting all vertex descriptors: see chapter 2.6
- Adding an edge: see chapter 2.7
- Getting all edges: see chapter 2.9
- Getting all edge descriptors: see chapter 2.11

These functions are mostly there for completion and showing which data types are used.

2.1 Creating an empty (directed) graph

Let's create a trivial empty graph!

Algorithm 1 shows the function to create an empty (directed) graph.

Algorithm 1 Creating an empty (directed) graph

```
#include <boost/graph/adjacency_list.hpp>

boost::adjacency_list<>
create_empty_directed_graph() noexcept
{
  return boost::adjacency_list<>();
}
```

Algorithm 2 demonstrates the 'create_empty_directed_graph' function.

Algorithm 2 Demonstration of 'create_empty_directed_graph'

```
#include "create_empty_directed_graph.h"

void create_empty_directed_graph_demo() noexcept
{
  const auto g = create_empty_directed_graph();
}
```

Congratulations, you've just created a boost::adjacency_list with its default template arguments. For your reference, these default template argument denote that you've just created a graph, in which:

- The out edges are stored in a std::vector
- The vertices are stored in a std::vector
- The edges have a direction
- The vertices, edges and graph have no properties
- The edges are stored in a std::list

The boost::adjacency_list is the most commonly used graph type, the other is the boost::adjacency_matrix.

2.2 Creating an empty undirected graph

Let's create another trivial empty graph! This time, we make the graph undirected.

Algorith 3 shows how to create an undirected graph.

Algorithm 3 Creating an empty undirected graph

```
#include <boost/graph/adjacency_list.hpp>

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS
>
create_empty_undirected_graph() noexcept
{
  return boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS
  >();
}
```

Algorithm 4 demonstrates the 'create_empty_undirected_graph' function.

Algorithm 4 Demonstration of 'create_empty_undirected_graph'

```
#include "create_empty_undirected_graph.h"

void create_empty_undirected_graph_demo() noexcept
{
  const auto g = create_empty_undirected_graph();
}
```

Congratulations, you've just created an undirected graph in which:

- The out edges are stored in a std::vector. This way to store out edges is selected by the first 'boost::vecS'
- The vertices are stored in a std::vector. This way to store vertices is selected by the second 'boost::vecS'
- The graph is undirected. This directionality is selected for by the third template argument, 'boost::undirectedS'
- Vertices, edges and graph have no properties
- Edges are stored in a std::list

The difference between directed and undirected graphs is in the edges: in an undirected graph, an edge connects two vertices without any directionality. In a directed graph, an edge goes from a certain vertex, its source, to another (which may actually be the same), its target.

2.3 Add a vertex

Empty graphs are nice, now its time to add a vertex!

To add a vertex to a graph, the boost::add_vertex function is used as shows in algorithm 5.

Algorithm 5 Adding a vertex to a graph

```
#include <boost/graph/adjacency_list.hpp>

template <typename graph>
void add_vertex(graph& g) noexcept
{
  boost::add_vertex(g);
}
```

Algorithm 6 shows how to add a vertex to a directed and undirected graph.

Algorithm 6 Demonstration of the 'add_vertex' function

```
#include "add_vertex.h"
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"

void add_vertex_demo() noexcept
{
  auto g = create_empty_undirected_graph();
  add_vertex(g);

  auto h = create_empty_directed_graph();
  add_vertex(h);
}
```

Note that boost::add_vertex (in the 'add_vertex' function) returns a vertex descriptor, which is ignored for now. Vertex descriptors are looked at in more details at the chapter 2.4.

2.4 Vertex descriptors

A vertex descriptor is a handle to a vertex within a graph.

Vertex descriptors can be obtained by:

- dereference a vertex iterator, see chapter 2.6

Vertex descriptors are used to:

- add and edge between two vertices, see chapter 2.7
- obtain properties of vertex a vertex, for example the vertex its out degrees (chapter 23), the vertex its name (chapter 33), or a custom vertex property (chapter 81)

In this tutorial, vertex descriptors have named prefixed with 'vd_', for example 'vd_1'.

2.5 Get the vertices

You cannot get the vertices. This may sound unexpected, as it must be possible to work on the vertices of a graph. Working on the vertices of a graph is done throught these steps:

- Obtain a vertex iterator pair from the graph
- Dereference a vertex iterator to obtain a vertex descriptor

boost::vertices is used to obtain a vertex iterator pair, as shown in algorithm 7. The first vertex iterator points to the first vertex (its descriptor, to be precise), the second points to beyond the last vertex. In this tutorial, vertex iterator pairs have named prefixed with 'vip_', for example 'vip_1'.

Algorithm 7 Get the vertex iterators of a graph

```
#include <boost/graph/adjacency_list.hpp>

template <class graph>
std::pair<
  typename graph::vertex_iterator,
  typename graph::vertex_iterator
>
get_vertices(const graph& g) noexcept
{
  return boost::vertices(g);
}
```

These vertex iterators can be dereferenced to obtain the vertex descriptors. Note that 'get_vertices' will not be used often in isolation: usually one obtains the vertex descriptors immediatly. Just for your references, algorithm 8 demonstrates of the 'get_vertices' function, by showing that the vertex iterators of an empty graph point to the same location.

Algorithm 8 Demonstration of 'get_vertices'

```
#include <cassert>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_vertices.h"

void get_vertices_demo() noexcept
{
  const auto g = create_empty_undirected_graph();
  const auto vip_g = get_vertices(g);
  assert(vip_g.first == vip_g.second);

  const auto h = create_empty_directed_graph();
  const auto vip_h = get_vertices(h);
  assert(vip_h.first == vip_h.second);
}
```

2.6 Get all vertex descriptors

Vertex descriptors are the way to manipulate those vertices. Let's get the all!

Vertex descriptors are obtained from dereferencing vertex iterators. Algorithm 9 shows how to obtain all vertex descriptors from a graph.

Algorithm 9 Get all vertex descriptors of a graph

```
#include <vector>
#include "boost/graph/graph_traits.hpp"

template <class graph>
std::vector<
  typename boost::graph_traits<graph>::vertex_descriptor
> get_vertex_descriptors(const graph& g) noexcept
{
  using boost::graph_traits;
  std::vector<
    typename graph_traits<graph>::vertex_descriptor
  > v;
  for (auto vi = vertices(g);
    vi.first != vi.second;
    ++vi.first)
  {
    v.emplace_back(*vi.first);
  }
  return v;
}
```

The 'get_vertex_descripors' function shows an important concept of the Boost.Graph library: boost::vertices returns two vertex iterators, which in turn can be dereferenced to obtain the vertex descriptors. Algorithm 10 demonstrates that an empty graph has no vertex descriptors.

Algorithm 10 Demonstration of 'get_vertex_descriptors'

```
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_vertex_descriptors.h"

void get_vertex_descriptors_demo() noexcept
{
  const auto g = create_empty_undirected_graph();
  const auto vds_g = get_vertex_descriptors(g);
  assert(vds_g.empty());

  const auto h = create_empty_directed_graph();
  const auto vds_h = get_vertex_descriptors(h);
  assert(vds_h.empty());
}
```

2.7 Add an edge

To add an edge to a graph, two vertex descriptors are needed. A vertex descriptor is a handle to the vertex with in graph (vertex descriptors are looked at in more details in chapter 2.4). Algorithm 11 adds two vertices to a graph, and connects these two using boost::add_edge:

Algorithm 11 Adding (two vertices and) an edge to a graph

```
#include <boost/graph/adjacency_list.hpp>

#include <cassert>

template <typename graph>
void add_edge(graph& g) noexcept
{
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto aer = boost::add_edge(
    vd_a,
    vd_b,
    g
  );

  assert(aer.second);
}
```

This algorithm only shows how to add an isolated edge to a graph, instead of allowing for graphs with higher connectivities. The function boost::add_vertex returns a vertex descriptor, which I prefix with 'vd'. The function boost::add_edge returns a std::pair, consisting of an edge descriptor and a boolean success indicator. In algorithm 11 we assert that this insertion was successfull. Insertion can fail if an edge is already present and duplicates are not allowed.

A demonstration of add_edge is shown in algorith 12, in which an edge is added to both a directed and undirected graph.

Algorithm 12 Demonstration of add_edge

```
#include "add_edge.h"
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"

void add_edge_demo() noexcept
{
  auto g = create_empty_undirected_graph();
  add_edge(g);

  auto h = create_empty_directed_graph();
  add_edge(h);
}
```

2.8 boost::add_edge result

When using the function 'boost::add_edge', a 'std::pair<edge_descriptor,bool>' is returned. It contains both the edge descriptor (see chapter 2.10) and a boolean indicating insertion success.

In this tutorial, boost::add_edge results have named prefixed with 'aer_', for example 'aer_1'.

2.9 Getting the edges

You cannot get the edges directly. Working on the edges of a graph is done throught these steps:

- Obtain an edge iterator pair from the graph
- Dereference an edge iterator to obtain an edge descriptor

boost::edges is used to obtain an edge iterator pair. The first edge iterator points to the first edge (its descriptor, to be precise), the second points to beyond the last edge. In this tutorial, edge iterator pairs have named prefixed with 'eip_', for example 'eip_1'.

Algorithm 13 Get the edge iterators of a graph

```
#include <boost/graph/adjacency_list.hpp>

template <class graph>
std::pair<
  typename graph::edge_iterator,
  typename graph::edge_iterator
>
get_edges(const graph& g) noexcept
{
  return boost::edges(g);
}
```

These edge iterators can be dereferenced to obtain the edge descriptors. Note that this function will not be used often in isolation: usually one obtains the edge descriptors immediatly.

Algorithm 14 demonstrates 'get_edges' by showing that both iterators of the edge iterator pair point to the same location, when the graph is empty.

Algorithm 14 Demonstration of get_edges

```
#include <cassert>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_edges.h"

void get_edges_demo() noexcept
{
  const auto g = create_empty_undirected_graph();
  const auto eip_g = get_edges(g);
  assert(eip_g.first == eip_g.second);

  auto h = create_empty_directed_graph();
  const auto eip_h = get_edges(h);
  assert(eip_h.first == eip_h.second);
}
```

2.10 Edge descriptors

An edge descriptor is a handle to an edge within a graph. Edge descriptors are used to:

- obtain the name, or other properties, of an edge

In this tutorial, edge descriptors have named prefixed with 'ed_', for example 'ed_1'.

2.11 Get all edge descriptors

Obtaining all edge descriptors is not as simple of a function as you'd guess:

Algorithm 15 Get all edge descriptors of a graph

```
#include <vector>
#include "boost/graph/graph_traits.hpp"

template <class graph>
std::vector<
  typename boost::graph_traits<graph>::edge_descriptor
> get_edge_descriptors(const graph& g) noexcept
{
  using boost::graph_traits;
  std::vector<
    typename graph_traits<graph>::edge_descriptor
  > v;
  for (auto vi = edges(g);
    vi.first != vi.second;
    ++vi.first)
  {
    v.emplace_back(*vi.first);
  }
  return v;
}
```

This does show an important concept of the Boost.Graph library: boost::edges returns to vertex iterators, that can be dereferenced to obtain the vertex descriptors.

Algorithm 16 demonstrates the 'get_edge_descriptor', by showing that empty graphs do not have any edge descriptors.

Algorithm 16 Demonstration of get_edge_descriptors

```
#include <cassert>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_edge_descriptors.h"

void get_edge_descriptors_demo() noexcept
{
  const auto g = create_empty_directed_graph();
  const auto eds_g = get_edge_descriptors(g);
  assert(eds_g.empty());

  const auto h = create_empty_undirected_graph();
  const auto eds_h = get_edge_descriptors(h);
  assert(eds_h.empty());
}
```

2.12 Creating K_2, a fully connected graph with two vertices

Finally, we are going to create a graph!

To create a fully connected graph with two vertices (also called K_2), one needs two vertices and one (undirected) edge, as depicted in figure 1.

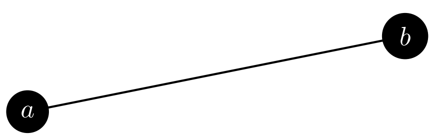

Figure 1: K_2: a fully connected graph with two vertices named a and b

To create K_2, the following code can be used:

Algorithm 17 Creating K_2 as depicted in figure 1

```
#include "create_empty_undirected_graph.h"

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS
>
create_k2_graph() noexcept
{
  auto g = create_empty_undirected_graph();
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto aer = boost::add_edge(vd_a, vd_b, g);
  assert(aer.second);
  return g;
}
```

To save defining the type, we call the 'create_empty_undirected_graph' function. The vertex descriptors (see chapter 2.4) created by two boost::add_vertex calls are stored to add an edge to the graph. From boost::add_edge its return type (see chapter 2.8), it is only checked that insertion has been successfull.

Note that the graph lacks all properties: nodes do not have names, nor do edges.

Algorithm 18 demonstrates how to 'create_k2_graph' and uses all functions currently described by this tutorial.

Algorithm 18 Demonstration of 'create_k2_graph'

```
#include <cassert>
#include <iostream>

#include "create_k2_graph.h"
#include "get_edge_descriptors.h"
#include "get_edges.h"
#include "get_vertex_descriptors.h"
#include "get_vertices.h"

void create_k2_graph_demo() noexcept
{
  const auto g = create_k2_graph();
  const auto vip = get_vertices(g);
  assert(vip.first != vip.second);
  const auto vds = get_vertex_descriptors(g);
  assert(vds.size() == 2);
  const auto eip = get_edges(g);
  assert(eip.first != eip.second);
  const auto eds = get_edge_descriptors(g);
  assert(eds.size() == 1);
}
```

3 Working on a graph without properties

Graphs without edge and vertex properties have plenty of things to measure. These simple getters and setters will allow you to work with, test and debug your code:

- Counting the number of vertices: see chapter 3.1
- Counting the number of edges: see chapter 3.2
- Getting the vertices' out degrees: see chapter 3.3

3.1 Counting the number of vertices

Use boost::num_vertices, as shown here:

Algorithm 19 Count the numbe of vertices

```
#include <boost/graph/adjacency_list.hpp>

///Get the number of vertices a graph has
template <class graph>
int get_n_vertices(const graph& g) noexcept
{
  return static_cast<int>(boost::num_vertices(g));
}
```

The function 'get_n_vertices' is demonstrated in algorithm 20, to measure the number of vertices of an empty (zero) and K_2 (two) graph.

Algorithm 20 Demonstration of the 'get_n_vertices' function

```
#include "get_n_vertices.h"

#include <cassert>

#include "create_empty_directed_graph.h"
#include "create_k2_graph.h"
#include "get_n_vertices.h"

void get_n_vertices_demo() noexcept
{
  const auto g = create_empty_directed_graph();
  assert(get_n_vertices(g) == 0);

  const auto h = create_k2_graph();
  assert(get_n_vertices(h) == 2);
}
```

3.2 Counting the number of edges

Use boost::num_edges, as shown here:

Algorithm 21 Count the number of edges

```
#include <boost/graph/adjacency_list.hpp>

///Get the number of edges a graph has
template <class graph>
int get_n_edges(const graph& g) noexcept
{
  return static_cast<int>(boost::num_edges(g));
}
```

The function 'get_n_edges' is demonstrated in algorithm 22, to measure the number of vertices of an empty (zero) and K_2 (one) graph.

Algorithm 22 Demonstration of the 'get_n_edges' function

```
#include "get_n_edges.h"

#include <cassert>

#include "create_empty_directed_graph.h"
#include "create_k2_graph.h"

void get_n_edges_demo() noexcept
{
  const auto g = create_empty_directed_graph();
  assert(get_n_edges(g) == 0);

  const auto h = create_k2_graph();
  assert(get_n_edges(h) == 1);
}
```

3.3 Getting the vertices' out degree

The out degree of a vertex is the number of edges that originate at it.

Algorithm 23 Get the vertices' out degrees

```
#include <vector>

template <typename graph>
std::vector<int> get_vertex_out_degrees(const graph& g)
    noexcept
{
  std::vector<int> v;
  for (auto p = vertices(g);
    p.first != p.second;
    ++p.first) {
    v.emplace_back(out_degree(*p.first,g));
  }
  return v;
}
```

The out degrees of the vertices are obtained directly from the vertex descriptor and then put into a std::vector. Note that the std::vector has element type 'int', instead of 'graph::degree_size_type', as one should prefer using int (over unsigned int) in an interface [1][1]. Also, avoid using an unsigned an int for the sake of gaining that one more bit [3][2].

Albeit K_2 is a simple graph, we can use it to demonstrate 'get_vertex_out_degrees' on, as shown in algorithm 24.

Algorithm 24 Demonstration of the 'get_vertex_out_degrees' function

```
#include <cassert>

#include "create_k2_graph.h"
#include "get_vertex_out_degrees.h"

void get_vertex_out_degrees_demo() noexcept
{
  const auto g = create_k2_graph();
  const std::vector<int> expected_out_degrees{1,1};
  const std::vector<int> vertex_out_degrees{
      get_vertex_out_degrees(g)};
  assert(expected_out_degrees == vertex_out_degrees);
}
```

[1] Chapter 9.2.2
[2] Chapter 4.4

3.4 Storing a graph as a .dot

Graph are easily saved to a .dot file:

Algorithm 25 Storing a graph as a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>

///Save a graph to a .dot file
template <typename graph>
void save_graph_to_dot(const graph& g, const std::string&
    filename) noexcept
{
  std::ofstream f(filename);
  boost::write_graphviz(f,g);
}
```

Algorithm 26 shows how to use this function.

Algorithm 26 Demonstration of the 'save_graph_to_dot' function

```
#include "create_k2_graph.h"
#include "create_named_vertices_k2_graph.h"
#include "save_graph_to_dot.h"

void save_graph_to_dot_demo() noexcept
{
  const auto g = create_k2_graph();
  save_graph_to_dot(g,"save_graph_to_dot_k2_graph.dot");

  const auto h = create_named_vertices_k2_graph();
  save_graph_to_dot(h,"
      save_graph_to_dot_named_vertices_k2_graph.dot");
}
```

When using the 'create_k2_graph function' (algorithm 17) to create a K_2 graph, the .dot file created is displayed in algorithm 27:

Algorithm 27 .dot file created from the create_k2_graph function (algorithm 17)

```
graph G {
0;
1;
0--1 ;
}
```

This .dot file corresponds to figure 2:

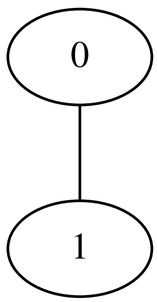

Figure 2: .svg file created from the create_k2_graph function (algorithm 17)

If you used the create_named_vertices_k2_graph function (algorithm 35) to produce a K_2 graph with named vertices, you see that the .dot file does not have stored the vertex names:

Algorithm 28 .dot file created from the create_named_vertices_k2_graph function (algorithm 35)

```
graph G {
0;
1;
0--1 ;
}
```

So, the 'save_graph_to_dot' function (algorithm 25) saves only the structure of the graph.

4 Building graphs with built-in properties

Up until now, the graphs created have had edges and vertices without any propery. In this chapter, graphs will be created, in which edges vertices can have a (std::string) name. There are many more built-in properties edges and nodes can have (see the boost/graph/properties.hpp file for these).

In this chapter, we will build the following graphs:

- An empty (undirected) graph that allows for vertices with names: see chapter 4.1
- $K_2$with named vertices: see chapter 4.4
- An empty (undirected) graph that allows for edges and vertices with names: see chapter 4.5
- $K_3$with named edges and vertices: see chapter 4.8

In the process, some basic (sometimes bordering trivial) functions are shown:

- Adding a named vertex: see chapter 4.2
- Getting the vertices' names: see chapter 4.3
- Adding an named edge: see chapter 4.6
- Getting the edges' names: see chapter 4.7

These functions are mostly there for completion and showing which data types are used.

4.1 Creating an empty graph with named vertices

Let's create a trivial empty graph, in which the vertices can have a name:

Algorithm 29 Creating an empty graph with named vertices

```
#include <boost/graph/adjacency_list.hpp>

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<
    boost::vertex_name_t,std::string
  >
>
create_empty_named_vertices_graph() noexcept
{
  return boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
      boost::vertex_name_t,std::string
    >
  >();
}
```

This graph:

- has its out edges stored in a std::vector (due to the first boost::vecS)
- has its vertices stored in a std::vector (due to the second boost::vecS)
- is undirected (due to the boost::undirectedS)
- The vertices have one property: they have a name, that is of data type std::string (due to the boost::property< boost::vertex_name_t,std::string>')
- Edges and graph have no properties
- Edges are stored in a std::list

The boost::adjacency_list has a new, fourth template argument 'boost::property< boost::vertex_name_t,std::string>'. This can be read as: "vertices have the property 'boost::vertex_name_t', that is of data type 'std::string"'. Or simply: "vertices have a name that is stored as a std::string".

Algorithm 30 shows how to create this graph. Note that all the earlier functions defined in this tutorial keep working as expected.

Algorithm 30 Demonstration if the 'create_empty_named_vertices_graph' function

```
#include <cassert>

#include "create_empty_named_vertices_graph.h"
#include "get_edge_descriptors.h"
#include "get_edges.h"
#include "get_vertex_descriptors.h"
#include "get_vertices.h"

void create_empty_named_vertices_graph_demo() noexcept
{
  const auto g = create_empty_named_vertices_graph();
  const auto vip = get_vertices(g);
  assert(vip.first == vip.second);
  const auto vds = get_vertex_descriptors(g);
  assert(vds.empty());
  const auto eip = get_edges(g);
  assert(eip.first == eip.second);
  const auto eds = get_edge_descriptors(g);
  assert(eds.empty());
}
```

4.2 Add a vertex with a name

Adding a vertex without a name was trivially easy (see chapter 5). Adding a vertex with a name is less easy:

Algorithm 31 Add a vertex with a name

```
#include <boost/graph/adjacency_list.hpp>

template <typename graph>
void add_named_vertex(const std::string& name, graph& g)
   noexcept
{
  const auto vd_a = boost::add_vertex(g);
  auto vertex_name_map = get(boost::vertex_name,g);
  vertex_name_map[vd_a] = name;
}
```

Instead of calling 'boost::add_vertex' with an additional argument containing the name of the vertex[3] , multiple things need to be done. When adding a new vertex to the graph, the vertex descriptor (as describes in chapter 2.4) is stored. After obtaining the name map from the graph (using 'boost::get(boost::vertex_name,g)'), the name of the vertex is set using that vertex descriptor.

Using add_named_vertex is straightforward, as demonstrated by algorithm 32.

Algorithm 32 Demonstration of 'add_named_vertex'

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_named_vertices_graph.h"
#include "get_vertex_descriptors.h"

void add_named_vertex_demo() noexcept
{
  auto g = create_empty_named_vertices_graph();
  add_named_vertex("Lex", g);
  assert(get_vertex_descriptors(g).size() == 1);
}
```

4.3 Getting the vertices' names

When the vertices of a graph have named vertices, one can extract them as such:

[3] I am unsure if this would have been a good interface. I am sure I expected this interface myself. I do see a problem with multiple properties and the order of initialization, but initialization can follow the same order as the the property list.

Algorithm 33 Get the vertices' names

```
#include <string>
#include <vector>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

//TODO: generalize to return any type
template <typename graph>
std::vector<std::string> get_vertex_names(const graph& g)
    noexcept
{
  std::vector<std::string> v;

  const auto vertex_name_map = get(boost::vertex_name,g);

  for (auto p = vertices(g);
    p.first != p.second;
    ++p.first) {
    v.emplace_back(get(vertex_name_map, *p.first));
  }
  return v;
}
```

The names of the vertices are obtained from a boost::property_map and then put into a std::vector. Note that the std::vector has element type 'std::string', instead of extracting the type from the graph. If you know how to do so, please email me.

When trying to get the vertices' names from a graph without vertices with names, you will get the error 'formed reference to void' (see chapter 10.1).

Algorithm 34 shows how to add two named vertices and how to get their names.

Algorithm 34 Demonstration of 'get_vertex_names'

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_named_vertices_graph.h"
#include "get_vertex_names.h"

void get_vertex_names_demo() noexcept
{
  auto g = create_empty_named_vertices_graph();
  const std::string vertex_name_1{"Chip"};
  const std::string vertex_name_2{"Chap"};
  add_named_vertex(vertex_name_1, g);
  add_named_vertex(vertex_name_2, g);
  const std::vector<std::string> expected_names{
      vertex_name_1, vertex_name_2};
  const std::vector<std::string> vertex_names{
      get_vertex_names(g)};
  assert(expected_names == vertex_names);
}
```

4.4 Creating K_2 with named vertices

We extend K_2 of chapter 2.12 by naming the vertices 'from' and 'to', as depicted in figure 3:

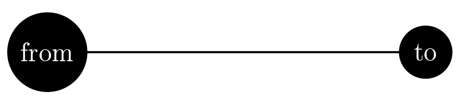

Figure 3: K_2: a fully connected graph with two vertices with the text 'from' and 'to'

To create K_2, the following code can be used:

Algorithm 35 Creating K_2 as depicted in figure 3

```
#include "create_named_vertices_k2_graph.h"
#include "create_empty_named_vertices_graph.h"

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<boost::vertex_name_t,std::string>
>
create_named_vertices_k2_graph() noexcept
{
  auto g = create_empty_named_vertices_graph();
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto aer = boost::add_edge(
    vd_a,
    vd_b,
    g
  );
  assert(aer.second);

  auto name_map = get(boost::vertex_name,g);
  name_map[vd_a] = "from";
  name_map[vd_b] = "to";

  return g;
}
```

Most of the code is a repeat of algorithm 17. In the end, the names are obtained as a boost::property_map and set.

Also the demonstration code (algorithm) is very similar to the demonstration code of the create_k2_graph function ().

Algorithm 36 Demonstrating the 'create_k2_graph' function

```
#include <cassert>

#include "create_named_vertices_k2_graph.h"
#include "get_vertex_names.h"

void create_named_vertices_k2_graph_demo() noexcept
{
  const auto g = create_named_vertices_k2_graph();
  const std::vector<std::string> expected_names{"from", "
      to"};
  const std::vector<std::string> vertex_names =
      get_vertex_names(g);
  assert(expected_names == vertex_names);
}
```

4.5 Creating an empty graph with named edges and vertices

Let's create a trivial empty graph, in which the both the edges and vertices can have a name:

Algorithm 37 Creating an empty graph with named edges and vertices

```
#include <string>
#include <boost/graph/adjacency_list.hpp>

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<boost::vertex_name_t,std::string>,
  boost::property<boost::edge_name_t,std::string>
>
create_empty_named_edges_and_vertices_graph() noexcept
{
  return boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
      boost::vertex_name_t,std::string
    >,
    boost::property<
      boost::edge_name_t,std::string
    >
  >();
}
```

This graph:

- has its out edges stored in a std::vector (due to the first boost::vecS)
- has its vertices stored in a std::vector (due to the second boost::vecS)
- is undirected (due to the boost::undirectedS)
- The vertices have one property: they have a name, that is of data type std::string (due to the boost::property< boost::vertex_name_t,std::string>')
- The edges have one property: they have a name, that is of data type std::string (due to the boost::property< boost::edge_name_t,std::string>')
- The graph has no properties
- Edges are stored in a std::list

The boost::adjacency_list has a new, fifth template argument 'boost::property< boost::edge_name_t,std::string>'. This can be read as: "edges have the property 'boost::edge_name_t', that is of data type 'std::string"'. Or simply: "edges have a name that is stored as a std::string".

Algorithm 38 shows how to create this graph. Note that all the earlier functions defined in this tutorial keep working as expected.

Algorithm 38 Demonstration if the 'create_empty_named_edges_and_vertices_graph' function

```
#include <cassert>

#include "add_named_edge.h"
#include "create_empty_named_edges_and_vertices_graph.h"
#include "get_edge_names.h"
#include "get_vertex_names.h"

void create_empty_named_edges_and_vertices_graph_demo()
    noexcept
{
  auto g = create_empty_named_edges_and_vertices_graph();
  add_named_edge("Reed", g);
  const std::vector<std::string> expected_vertex_names{""
      ,""};
  const std::vector<std::string> vertex_names =
      get_vertex_names(g);
  assert(expected_vertex_names == vertex_names);
  const std::vector<std::string> expected_edge_names{"
      Reed"};
  const std::vector<std::string> edge_names =
      get_edge_names(g);
  assert(expected_edge_names == edge_names);
}
```

4.6 Adding a named edge

Adding an edge with a name:

Algorithm 39 Add a vertex with a name

```
#include <boost/graph/adjacency_list.hpp>

#include <cassert>

template <typename graph>
void add_named_edge(const std::string& edge_name, graph&
    g) noexcept
{
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto aer = boost::add_edge(vd_a, vd_b, g);
  assert(aer.second);

  auto edge_name_map = get(boost::edge_name,g);
  edge_name_map[aer.first] = edge_name;
}
```

In this code snippet, the edge descriptor (see chapter 2.10 if you need to refresh your memory) when using 'boost::add_edge' is used as a key to change the edge its name map.

The algorithm 40 shows how to add a named edge to an empty graph. When trying to add named vertices to graph without this property, you will get the error 'formed reference to void' (see chapter 10.1).

Algorithm 40 Demonstration of the 'add_named_edge' function

```
#include <cassert>

#include "add_named_edge.h"
#include "create_empty_named_edges_and_vertices_graph.h"
#include "get_n_edges.h"

void add_named_edge_demo() noexcept
{
  auto g = create_empty_named_edges_and_vertices_graph();
  add_named_edge("Richards", g);
  assert(get_n_edges(g) == 1);
}
```

4.7 Getting the edges' names

When the edges of a graph have named vertices, one can extract them as such:

Algorithm 41 Get the edges' names

```
#include <string>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

//TODO: generalize to return any type
template <typename graph>
std::vector<std::string> get_edge_names(const graph& g)
    noexcept
{
  std::vector<std::string> v;

  const auto edge_name_map = get(boost::edge_name,g);

  for (auto p = boost::edges(g);
    p.first != p.second;
    ++p.first) {
    v.emplace_back(get(edge_name_map, *p.first));
  }
  return v;
}
```

The names of the edges are obtained from a boost::property_map and then put into a std::vector. The algorithm 42 shows how to apply this function.

Would you dare to try to get the edges' names from a graph without vertices with names, you will get the error 'formed reference to void' (see chapter 10.1).

Algorithm 42 Demonstration of the 'get_edge_names' function

```
#include <cassert>

#include "add_named_edge.h"
#include "create_empty_named_edges_and_vertices_graph.h"
#include "get_edge_names.h"

void get_edge_names_demo() noexcept
{
  auto g = create_empty_named_edges_and_vertices_graph();
  const std::string edge_name_1{"Eugene"};
  const std::string edge_name_2{"Another Eugene"};
  add_named_edge(edge_name_1, g);
  add_named_edge(edge_name_2, g);
  const std::vector<std::string> expected_names{
      edge_name_1, edge_name_2};
  const std::vector<std::string> edge_names{
      get_edge_names(g)};
  assert(expected_names == edge_names);
}
```

4.8 Creating K_3 with named edges and vertices

We extend the graph K_2 with named vertices of chapter 4.4 by adding names to the edges, as depicted in figure 4:

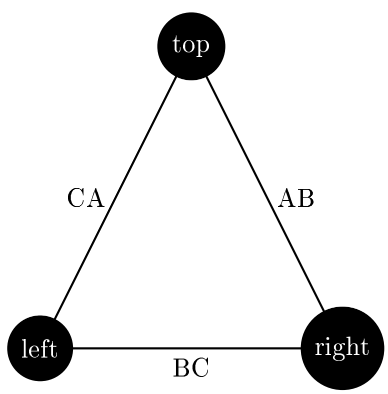

Figure 4: K_3: a fully connected graph with three named edges and vertices

To create K_3, the following code can be used:

Algorithm 43 Creating K_3 as depicted in figure 4

```
#include <boost/graph/adjacency_list.hpp>
#include <string>

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<boost::vertex_name_t,std::string>,
  boost::property<boost::edge_name_t,std::string>
>
create_named_edges_and_vertices_k3_graph() noexcept
{
  auto g = create_empty_named_edges_and_vertices_graph();
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto vd_c = boost::add_vertex(g);
  const auto aer_ab = boost::add_edge(vd_a, vd_b, g);
  assert(aer_ab.second);
  const auto aer_bc = boost::add_edge(vd_b, vd_c, g);
  assert(aer_bc.second);
  const auto aer_ca = boost::add_edge(vd_c, vd_a, g);
  assert(aer_ca.second);

  //Add vertex names
  auto vertex_name_map = get(boost::vertex_name,g);
  vertex_name_map[vd_a] = "top";
  vertex_name_map[vd_b] = "right";
  vertex_name_map[vd_c] = "left";

  //Add edge names
  auto edge_name_map = get(boost::edge_name,g);
  edge_name_map[aer_ab.first] = "AB";
  edge_name_map[aer_bc.first] = "BC";
  edge_name_map[aer_ca.first] = "CA";

  return g;
}
```

Most of the code is a repeat of algorithm 35. In the end, the edge names are obtained as a boost::property_map and set. Algorithm 44 shows how to create the graph and measure its edge and vertex names.

Algorithm 44 Demonstration of the 'create_named_edges_and_vertices_k3' function

```
#include <cassert>
#include <iostream>
#include "create_named_edges_and_vertices_k3_graph.h"
#include "get_edge_names.h"
#include "get_vertex_names.h"

void create_named_edges_and_vertices_k3_graph_demo()
    noexcept
{
  const auto g = create_named_edges_and_vertices_k3_graph
      ();
  const std::vector<std::string> expected_vertex_names{"
      top", "right", "left"};
  const std::vector<std::string> vertex_names{
      get_vertex_names(g)};
  assert(expected_vertex_names == vertex_names);
  const std::vector<std::string> expected_edge_names{"AB"
      , "BC", "CA"};
  const std::vector<std::string> edge_names{
      get_edge_names(g)};
  assert(expected_edge_names == edge_names);
}
```

5 Working with graphs with named edges and vertices

Measuring simple traits of the graphs created allows

- Check if there exists a vertex with a certain name: chapter 5.1
- Find a (named) vertex by its name: chapter 5.2
- Get a (named) vertex its degree, in degree and out degree: chapter: 5.3
- Get a (named) vertex its name from its vertex descriptor: chapter 5.4
- Set a (named) vertex its name using its vertex descriptor: chapter 5.5
- Setting all vertices' names: chapter 5.6
- Clear a named vertex its edges: chapter 5.7
- Remove a named vertex: chapter 5.8

- Check if there exists an edge with a certain name: chapter 5.9
- Find a (named) edge by its name: chapter 5.10
- Get a (named) edge its name from its edge descriptor: chapter
- Set a (named) edge its name using its edge descriptor: chapter
- Remove a named edge: chapter
- Storing a graph with named vertices as a .dot: chapter 5.14
- Storing a graph with named edges and vertices as a .dot: chapter 5.15

Especially the first paragraph is important: 'find_first_vertex_by_name' shows how to obtain a vertex descriptor, which is used in later algorithms.

5.1 Check if there exists a vertex with a certain name

Before modifying our vertices, let's first determine if we can find a vertex by its name in a graph. After obtaing a name map, we obtain the vertex iterators, dereference these to obtain the vertex descriptors and then compare each vertex its name with the one desired.

Algorithm 45 Find if there is vertex with a certain name

```
#include <string>
#include <boost/graph/properties.hpp>

template <typename graph>
bool has_vertex_with_name(
  const std::string& name,
  const graph& g
) noexcept
{
  const auto vertex_name_map = get(boost::vertex_name, g)
     ;

  for (auto p = vertices(g);
    p.first != p.second;
    ++p.first) {
    if (get(vertex_name_map, *p.first) == name) {
      return true;
    }
  }
  return false;
}
```

This function can be demonstrated as in algorithm 46, where a certain name cannot be found in an empty graph. After adding the desired name, it is found.

Algorithm 46 Demonstration of the 'has_vertex_with_name' function

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_named_vertices_graph.h"
#include "has_vertex_with_name.h"

void has_vertex_with_name_demo() noexcept
{
  auto g = create_empty_named_vertices_graph();
  assert(!has_vertex_with_name("Felix",g));
  add_named_vertex("Felix",g);
  assert(has_vertex_with_name("Felix",g));
}
```

Note that this function only finds if there is at least one vertex with that name: it does not tell how many vertices with that name exist in the graph.

5.2 Find a vertex by its name

Where STL functions work with iterators, here we obtain a vertex descriptor (see chapter 2.4) to obtain a handle to the desired vertex. Algorithm 47 shows how to obtain a vertex descriptor to the first (name) vertex found with a specific name.

Algorithm 47 Find the first vertex by its name

```
#include <string>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

template <typename graph>
typename boost::graph_traits<graph>::vertex_descriptor
find_first_vertex_with_name(
  const std::string& name,
  const graph& g
) noexcept
{
  const auto vertex_name_map = get(boost::vertex_name,g);

  for (auto p = vertices(g);
    p.first != p.second;
    ++p.first) {
    const std::string s{
      get(vertex_name_map, *p.first)
    };
    if (s == name) { return *p.first; }
  }
  return *vertices(g).second;

}
```

With the vertex descriptor obtained, one can read and modify the vertex and the edges surrounding it. Algorithm 48 shows some examples of how to do so.

Algorithm 48 Demonstration of the 'find_first_vertex_by_name' function

```
#include <cassert>

#include "create_named_vertices_k2_graph.h"
#include "find_first_vertex_with_name.h"

void find_first_vertex_with_name_demo() noexcept
{
  const auto g = create_named_vertices_k2_graph();
  const auto vd = find_first_vertex_with_name("from", g);
  assert(boost::out_degree(vd,g) == 1);
  assert(boost::in_degree(vd,g) == 1);
}
```

5.3 Get a (named) vertex its degree, in degree and out degree

We already obtained all out degrees of all vertices in chapter 3.3 by just collecting all vertex descriptors. Here, we will search for a vertex with a certain name, obtain its vertex descriptor and find the number of connections it has. The number of connections is called the 'degree' of the vertex. There are three types of degrees:

- in degree: the number of incoming connections, using boost::in_degree
- out degree: the number of outgoing connections, using boost::in_degree
- degree: sum of the in degree and out degree, using boost::in_degree

With a vertex descriptor, we can read a vertex its types of degrees. Algorithm 23 shows how to find a vertex, obtain its vertex descriptor and then obtain the out degree from it.

Algorithm 49 Get the first vertex with a certain name its out degree from its vertex descriptor

```
#include <cassert>
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "find_first_vertex_with_name.h"
#include "has_vertex_with_name.h"

template <typename graph>
int get_first_vertex_with_name_out_degree(
  const std::string& name,
  const graph& g) noexcept
{
  assert(has_vertex_with_name(name, g));
  const auto vd = find_first_vertex_with_name(name, g);
  return static_cast<int>(boost::out_degree(vd, g));
}
```

Algorithm 34 shows how to use this function.

Algorithm 50 Demonstration of the 'get_first_vertex_with_name_out_degree' function

```
#include <cassert>
#include "create_named_vertices_k2_graph.h"
#include "get_first_vertex_with_name_out_degree.h"

void get_first_vertex_with_name_out_degree_demo()
    noexcept
{
  const auto g = create_named_vertices_k2_graph();
  assert(get_first_vertex_with_name_out_degree("from", g)
       == 1);
  assert(get_first_vertex_with_name_out_degree("to", g)
     == 1);
}
```

5.4 Get a (named) vertex its name from its vertex descriptor

This may seem a trivial paragraph, as chapter 4.3 describes the 'get_vertex_names' algorithm, in which we get all vertices' names. But it does not allow to first find a vertex of interest and subsequently getting only that one its name.

To obtain the name from a vertex descriptor, one needs to pull out the name map and then look up the vertex of interest (I like to compare it as such: the vertex descriptor is a last name, the name map is a phone book, the desired info a phone number).

Algorithm 51 Get a vertex its name from its vertex descriptor

```
#include <string>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

template <typename graph>
std::string get_vertex_name(
  const typename boost::graph_traits<graph>::
      vertex_descriptor& vd,
  const graph& g
) noexcept
{
  const auto vertex_name_map = get(boost::vertex_name,g);
  return vertex_name_map[vd];
}
```

To use 'get_vertex_name', one first needs to obtain a vertex descriptor. Algorithm 34 shows a simple example.

Algorithm 52 Demonstration if the 'get_vertex_name' function

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_named_vertices_graph.h"
#include "find_first_vertex_with_name.h"
#include "get_vertex_name.h"

void get_vertex_name_demo() noexcept
{
  auto g = create_empty_named_vertices_graph();
  const std::string name{"Dex"};
  add_named_vertex(name, g);
  const auto vd = find_first_vertex_with_name(name,g);
  assert(get_vertex_name(vd,g) == name);
}
```

5.5 Set a (named) vertex its name from its vertex descriptor

If you know how to get the name from a vertex descriptor, setting it is just as easy, as shown in algorithm 53.

Algorithm 53 Set a vertex its name from its vertex descriptor

```
#include <string>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

template <typename graph>
void set_vertex_name(
  const std::string& name,
  const typename boost::graph_traits<graph>::
     vertex_descriptor& vd,
  graph& g
) noexcept
{
  auto vertex_name_map = get(boost::vertex_name,g);
  vertex_name_map[vd] = name;
}
```

To use 'set_vertex_name', one first needs to obtain a vertex descriptor. Algorithm 54 shows a simple example.

Algorithm 54 Demonstration if the 'set_vertex_name' function

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_named_vertices_graph.h"
#include "find_first_vertex_with_name.h"
#include "get_vertex_name.h"
#include "set_vertex_name.h"

void set_vertex_name_demo() noexcept
{
  auto g = create_empty_named_vertices_graph();
  const std::string old_name{"Dex"};
  add_named_vertex(old_name, g);
  const auto vd = find_first_vertex_with_name(old_name,g)
     ;
  assert(get_vertex_name(vd,g) == old_name);
  const std::string new_name{"Diggy"};
  set_vertex_name(new_name, vd, g);
  assert(get_vertex_name(vd,g) == new_name);
}
```

5.6 Setting all vertices' names

When the vertices of a graph have named vertices and you want to set all their names at once:

Algorithm 55 Setting the vertices' names

```
#include <string>
#include <vector>

#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

//TODO: generalize 'names'
template <typename graph>
void set_vertex_names(
  graph& g,
  const std::vector<std::string>& names
) noexcept
{
  const auto vertex_name_map = get(boost::vertex_name,g);

  auto names_begin = std::begin(names);
  const auto names_end = std::end(names);
  for (auto vi = vertices(g);
    vi.first != vi.second;
    ++vi.first, ++names_begin)
  {
    assert(names_begin != names_end);
    put(vertex_name_map, *vi.first,*names_begin);
  }
}
```

An impressive feature is that getting the property map holding the graph its names is not a copy, but a reference. Otherwise, modifying 'name_map' (obtained by non-reference) would only modify a copy.

5.7 Clear the edges of a named vertex

A vertex descriptor can be used to clear all in/out/both edges connected to a vertex. It is necessary to remove these connections before the vertex itself can be removed. There are three functions to remove the edges connected to a vertex:

- boost::clear_vertex: removes all edges to and from the vertex
- boost::clear_out_edges: removes all outgoing edges from the vertex (in directed graphs only, else you will get a 'error: no matching function for call to clear_out_edges', as described in chapter 10.2)
- boost::clear_in_edges: removes all incoming edges from the vertex (in

directed graphs only, else you will get a 'error: no matching function for call to clear_in_edges', as described in chapter 10.3)

In the algorithm 'clear_first_vertex_with_name' the 'boost::clear_vertex' algorithm is used, as the graph used is undirectional:

Algorithm 56 Clear the first vertex with a certain name

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "find_first_vertex_with_name.h"
#include "has_vertex_with_name.h"

template <class graph>
void clear_first_vertex_with_name(
  const std::string& name,
  graph& g
) noexcept
{
  assert(has_vertex_with_name(name,g));
  const auto vd = find_first_vertex_with_name(name,g);
  boost::clear_vertex(vd,g);
}
```

Algorithm 57 shows the clearing of the first named vertex found.

Algorithm 57 Demonstration of the 'clear_first_vertex_with_name' function

```
#include <cassert>

#include "clear_first_vertex_with_name.h"
#include "create_named_vertices_k2_graph.h"
#include "get_n_edges.h"

void clear_first_vertex_with_name_demo() noexcept
{
  auto g = create_named_vertices_k2_graph();
  assert(get_n_edges(g) == 1);
  clear_first_vertex_with_name("from",g);
  assert(get_n_edges(g) == 0);
}
```

5.8 Remove a named vertex

A vertex descriptor can be used to remove a vertex from a graph. It is necessary to remove these connections (e.g. using clear_first_vertex_with_name', algorithm 56) before the vertex itself can be removed.

Removing a named vertex goes as follows: use the name of the vertex to get a first vertex descriptor, then call 'boost::remove_vertex', shown in algorithm 56.

Algorithm 58 Remove the first vertex with a certain name

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "find_first_vertex_with_name.h"
#include "has_vertex_with_name.h"

template <class graph>
void remove_first_vertex_with_name(
  const std::string& name,
  graph& g
) noexcept
{
  assert(has_vertex_with_name(name,g));
  const auto vd = find_first_vertex_with_name(name,g);
  assert(boost::degree(vd,g) == 0);
  boost::remove_vertex(vd,g);
}
```

Algorithm 59 shows the removal of the first named vertex found.

Algorithm 59 Demonstration of the 'remove_first_vertex_with_name' function

```
#include <cassert>

#include "clear_first_vertex_with_name.h"
#include "create_named_vertices_k2_graph.h"
#include "get_n_edges.h"
#include "get_n_vertices.h"
#include "remove_first_vertex_with_name.h"

void remove_first_vertex_with_name_demo() noexcept
{
  auto g = create_named_vertices_k2_graph();
  clear_first_vertex_with_name("from",g);
  remove_first_vertex_with_name("from",g);
  assert(get_n_edges(g) == 0);
  assert(get_n_vertices(g) == 1);
}
```

Again, be sure that the vertex removed does not have any connections!

5.9 Check if there exists an edge with a certain name

Before modifying our edges, let's first determine if we can find an edge by its name in a graph. After obtaing a name map, we obtain the edge iterators, dereference these to obtain the edge descriptors and then compare each edge its name with the one desired.

Algorithm 60 Find if there is an edge with a certain name

```
#include <string>
#include <boost/graph/properties.hpp>

template <typename graph>
bool has_edge_with_name(
  const std::string& name,
  const graph& g
) noexcept
{
  const auto edge_name_map = get(boost::edge_name,g);

  for (auto p = edges(g);
    p.first != p.second;
    ++p.first) {
    if (get(edge_name_map, *p.first) == name) {
      return true;
    }
  }
  return false;
}
```

This function can be demonstrated as in algorithm 61, where a certain name cannot be found in an empty graph. After adding the desired name, it is found.

Algorithm 61 Demonstration of the 'has_edge_with_name' function

```
#include <cassert>

#include "add_named_edge.h"
#include "create_empty_named_edges_and_vertices_graph.h"
#include "has_edge_with_name.h"

void has_edge_with_name_demo() noexcept
{
  auto g = create_empty_named_edges_and_vertices_graph();
  assert(!has_edge_with_name("Edward",g));
  add_named_edge("Edward",g);
  assert(has_edge_with_name("Edward",g));
}
```

Note that this function only finds if there is at least one edge with that name: it does not tell how many edges with that name exist in the graph.

5.10 Find an edge by its name

Where STL functions work with iterators, here we obtain an edge descriptor (see chapter 2.10) to obtain a handle to the desired edge. Algorithm 62 shows how to obtain an edge descriptor to the first (name) edge found with a specific name.

Algorithm 62 Find the first edge by its name

```
#include <string>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
find_first_edge_with_name(
  const std::string& name,
  const graph& g
) noexcept
{
  const auto edge_name_map = get(boost::edge_name,g);

  for (auto p = edges(g);
    p.first != p.second;
    ++p.first) {
    const std::string s{
      get(edge_name_map, *p.first)
    };
    if (s == name) { return *p.first; }
  }
  return *edges(g).second;
}
```

With the edge descriptor obtained, one can read and modify the graph. Algorithm 63 shows some examples of how to do so.

Algorithm 63 Demonstration of the 'find_first_edge_by_name' function

```
#include <cassert>

#include "create_named_edges_and_vertices_k3_graph.h"
#include "find_first_edge_with_name.h"

void find_first_edge_with_name_demo() noexcept
{
  const auto g = create_named_edges_and_vertices_k3_graph
      ();
  const auto ed = find_first_edge_with_name("AB", g);
  assert(boost::source(ed,g) != boost::target(ed,g));
}
```

5.11 Get a (named) edge its name from its edge descriptor

This may seem a trivial paragraph, as chapter 4.7 describes the 'get_edge_names' algorithm, in which we get all edges' names. But it does not allow to first find an edge of interest and subsequently getting only that one its name.

To obtain the name from an edgedescriptor, one needs to pull out the name map and then look up the edge of interest.

Algorithm 64 Get an edge its name from its edge descriptor

```
#include <string>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

template <typename graph>
std::string get_edge_name(
  const typename boost::graph_traits<graph>::
      edge_descriptor& vd,
  const graph& g
) noexcept
{
  const auto edge_name_map = get(boost::edge_name,g);
  return edge_name_map[vd];
}
```

To use 'get_edge_name', one first needs to obtain an edge descriptor. Algorithm 34 shows a simple example.

Algorithm 65 Demonstration if the 'get_edge_name' function

```
#include <cassert>

#include "add_named_edge.h"
#include "create_empty_named_edges_and_vertices_graph.h"
#include "find_first_edge_with_name.h"
#include "get_edge_name.h"

void get_edge_name_demo() noexcept
{
  auto g = create_empty_named_edges_and_vertices_graph();
  const std::string name{"Dex"};
  add_named_edge(name, g);
  const auto ed = find_first_edge_with_name(name,g);
  assert(get_edge_name(ed,g) == name);
}
```

5.12 Set a (named) edge its name from its edge descriptor

If you know how to get the name from an edge descriptor, setting it is just as easy, as shown in algorithm 66.

Algorithm 66 Set an edge its name from its edge descriptor

```
#include <string>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

template <typename graph>
void set_edge_name(
  const std::string& name,
  const typename boost::graph_traits<graph>::
      edge_descriptor& vd,
  graph& g
) noexcept
{
  auto edge_name_map = get(boost::edge_name,g);
  edge_name_map[vd] = name;
}
```

To use 'set_edge_name', one first needs to obtain an edge descriptor. Algorithm 67 shows a simple example.

Algorithm 67 Demonstration if the 'set_edge_name' function

```
#include <cassert>

#include "add_named_edge.h"
#include "create_empty_named_edges_and_vertices_graph.h"
#include "find_first_edge_with_name.h"
#include "get_edge_name.h"
#include "set_edge_name.h"

void set_edge_name_demo() noexcept
{
  auto g = create_empty_named_edges_and_vertices_graph();
  const std::string old_name{"Dex"};
  add_named_edge(old_name, g);
  const auto vd = find_first_edge_with_name(old_name,g);
  assert(get_edge_name(vd,g) == old_name);
  const std::string new_name{"Diggy"};
  set_edge_name(new_name, vd, g);
  assert(get_edge_name(vd,g) == new_name);
}
```

5.13 Removing a named edge

There are two ways to remove an edge:

1. Get an edge descriptor and call 'boost::remove_edge' on that descriptor: chapter 5.13.1
2. Get two vertex descriptors and call 'boost::remove_edge' on those two descriptors: chapter 5.13.2

5.13.1 Removing the first edge with a certain name

An edge descriptor can be used to remove an edge from a graph.

Removing a named edge goes as follows: use the name of the edge to get a first edge descriptor, then call 'boost::remove_edge', shown in algorithm 56.

Algorithm 68 Remove the first edge with a certain name

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "find_first_edge_with_name.h"
#include "has_edge_with_name.h"

template <class graph>
void remove_first_edge_with_name(
  const std::string& name,
  graph& g
) noexcept
{
  assert(has_edge_with_name(name,g));
  const auto vd = find_first_edge_with_name(name,g);
  boost::remove_edge(vd,g);
}
```

Algorithm 69 shows the removal of the first named edge found.

Algorithm 69 Demonstration of the 'remove_first_edge_with_name' function

```
#include <cassert>

#include "create_named_edges_and_vertices_k3_graph.h"
#include "get_n_edges.h"
#include "get_n_vertices.h"
#include "remove_first_edge_with_name.h"

void remove_first_edge_with_name_demo() noexcept
{
  auto g = create_named_edges_and_vertices_k3_graph();
  assert(get_n_edges(g) == 3);
  assert(get_n_vertices(g) == 3);
  remove_first_edge_with_name("AB",g);
  assert(get_n_edges(g) == 2);
  assert(get_n_vertices(g) == 3);
}
```

5.13.2 Removing the edge between two named vertices

Instead of looking for an edge descriptor, one can also remove an edge from two vertex descriptors (which is: the edge between the two vertices). Removing an edge between two named vertices named edge goes as follows: use the names of the vertices to get both vertex descriptors, then call 'boost::remove_edge' on those two, as shown in algorithm 56.

Algorithm 70 Remove the first edge with a certain name

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "find_first_vertex_with_name.h"
#include "has_vertex_with_name.h"
#include "has_edge_between_vertices.h"

template <typename graph>
void remove_edge_between_vertices_with_names(
  const std::string& name_1,
  const std::string& name_2,
  graph& g
) noexcept
{
  assert(has_vertex_with_name(name_1, g));
  assert(has_vertex_with_name(name_2, g));
  const auto vd_1 = find_first_vertex_with_name(name_1, g
    );
  const auto vd_2 = find_first_vertex_with_name(name_2, g
    );
  assert(has_edge_between_vertices(vd_1, vd_2, g));
  boost::remove_edge(vd_1, vd_2, g);
}
```

Algorithm 71 shows the removal of the first named edge found.

Algorithm 71 Demonstration of the 'remove_edge_between_vertices_with_names' function

```
#include <cassert>

#include "create_named_edges_and_vertices_k3_graph.h"
#include "get_n_edges.h"

void remove_edge_between_vertices_with_names_demo()
    noexcept
{
  auto g = create_named_edges_and_vertices_k3_graph();
  assert(get_n_edges(g) == 3);
  remove_edge_between_vertices_with_names("top","right",g
      );
  assert(get_n_edges(g) == 2);
}
```

5.14 Storing a graph with named vertices as a .dot

If you used the create_named_vertices_k2_graph function (algorithm 35) to produce a K_2 graph with named vertices, you can store these names additionally with algorithm 72:

Algorithm 72 Storing a graph with named vertices as a .dot file

```
#include <string>
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/properties.hpp>

#include "get_vertex_names.h"

///Save a graph with named vertices to a .dot file
template <typename graph>
void save_named_vertices_graph_to_dot(const graph& g,
    const std::string& filename) noexcept
{
  std::ofstream f(filename);
  const auto names = get_vertex_names(g);
  boost::write_graphviz(f,g,boost::make_label_writer(&
      names[0]));
}
```

The .dot file created is displayed in algorithm 73:

Algorithm 73 .dot file created from the create_named_vertices_k2_graph function (algorithm 35)

```
graph G {
0[label=from];
1[label=to];
0--1 ;
}
```

This .dot file corresponds to figure 5:

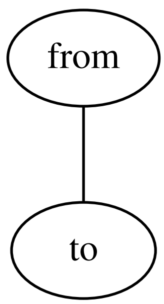

Figure 5: .svg file created from the create_k2_graph function (algorithm 35)

If you used the create_named_edges_and_vertices_k3_graph function (algorithm 43) to produce a K_3 graph with named edges and vertices, you see that the .dot file does not have stored the edge names:

Algorithm 74 .dot file created from the create_named_edges_and_vertices_k3_graph function (algorithm 43)

```
graph G {
0[label=top];
1[label=right];
2[label=left];
0--1 ;
1--2 ;
2--0 ;
}
```

So, the 'save_named_vertices_graph_to_dot' function (algorithm 25) saves only the structure of the graph and its vertex names.

5.15 Storing a graph with named vertices and edges as a .dot

If you used the create_named_edges_and_vertices_k3_graph function (algorithm 43) to produce a K_3 graph with named edges and vertices, you can store these names additionally with algorithm 75:

Algorithm 75 Storing a graph with named edges and vertices as a .dot file

```
#include <string>
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/properties.hpp>

#include "get_edge_names.h"
#include "get_vertex_names.h"

///Save a graph with named vertices to a .dot file
template <typename graph>
void save_named_edges_and_vertices_graph_to_dot(const
    graph& g, const std::string& filename)
{
  std::ofstream f(filename);
  const auto vertex_names = get_vertex_names(g);
  const auto edge_name_map = boost::get(boost::edge_name,
      g);
  boost::write_graphviz(
    f,
    g,
    boost::make_label_writer(&vertex_names[0]),
    [edge_name_map](std::ostream& out, const auto& e) {
      out << "[label=\"" << edge_name_map[e] << "\"]";
    }
  );
}
```

Note that this algorithm uses C++17.

The .dot file created is displayed in algorithm 76:

Algorithm 76 .dot file created from the create_named_edges_and_vertices_k3_graph function (algorithm 35)

```
graph G {
0[label=top];
1[label=right];
2[label=left];
0--1 [label="AB"];
1--2 [label="BC"];
2--0 [label="CA"];
}
```

This .dot file corresponds to figure 6:

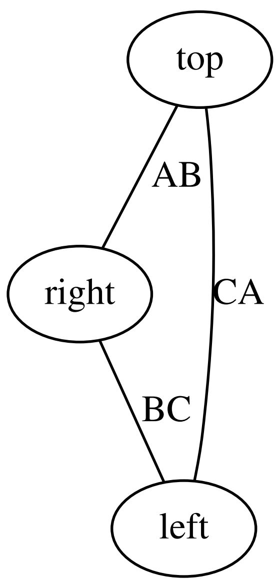

Figure 6: .svg file created from the create_named_edges_and_vertices_k3_graph function (algorithm 35)

If you created a graph with edges more complex than just a name, you will still just write these to the .dot file. Chapter 7.10 shows how to write custom vertices to a .dot file.

So, the 'save_named_edges_and_vertices_graph_to_dot' function (algorithm 25) saves only the structure of the graph and its edge and vertex names.

6 Building graphs with custom properties

Up until now, the graphs created have had edges and vertices with the built-in name propery. In this chapter, graphs will be created, in which the edges and vertices can have a custom 'my_edge' and 'my_edge' type[4].

- An empty (undirected) graph that allows for custom vertices: see chapter 6.1
- $K_2$with custom vertices: see chapter 6.4
- An empty (undirected) graph that allows for custom edges and vertices: see chapter 6.5
- $K_3$with custom edges and vertices: see chapter 6.7

In the process, some basic (sometimes bordering trivial) functions are shown:

- Adding a custom vertex: see chapter 6.2
- Adding a custom edge: see chapter 6.6

These functions are mostly there for completion and showing which data types are used.

6.1 Create an empty graph with custom vertices

Say we want to use our own vertex class as graph nodes. This is done in multiple steps:

1. Create a custom vertex class, called 'my_vertex'
2. Install a new property, called 'vertex_custom_type'
3. Use the new property in creating a boost::adjacency_list

6.1.1 Creating the custom vertex class

In this example, I create a custom vertex class. Here I will show the header file of it, as the implementation of it is not important yet.

[4] I do not intend to be original in naming my data types

Algorithm 77 Declaration of my_vertex

```
#ifndef MY_VERTEX_H
#define MY_VERTEX_H

#include <string>

class my_vertex
{
public:
  my_vertex(
    const std::string& name = "",
    const std::string& description = "",
    const double x = 0.0,
    const double y = 0.0
  ) noexcept;
  std::string m_name;
  std::string m_description;
  double m_x;
  double m_y;
};

bool operator==(const my_vertex& lhs, const my_vertex&
    rhs) noexcept;

#endif // MY_VERTEX_H
```

my_vertex is a class that has multiple properties: two doubles 'm_x' ('m_' stands for member) and 'm_y', and two std::strings m_name and m_description. my_vertex is copyable, but cannot trivially be converted to a std::string.

6.1.2 Installing the new property

Installing a new property would have been easier, if 'more C++ compilers were standards conformant' ([2], chapter 3.6, footnote at page 52). Boost.Graph uses the BOOST_INSTALL_PROPERTY macro to allow using a custom property:

Algorithm 78 Installing the vertex_custom_type property

```
#include <boost/graph/properties.hpp>

namespace boost {
  enum vertex_custom_type_t { vertex_custom_type = 314 };
  BOOST_INSTALL_PROPERTY(vertex,custom_type);
}
```

The enum value 314 must be unique.

6.1.3 Create the empty graph with custom vertices

Algorithm 79 Creating an empty graph with custom vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "install_vertex_custom_type.h"
#include "my_vertex.h"

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<
    boost::vertex_custom_type_t,my_vertex
  >
>
create_empty_custom_vertices_graph() noexcept
{
  return boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
      boost::vertex_custom_type_t,my_vertex
    >
  >();
}
```

This graph:

- has its out edges stored in a std::vector (due to the first boost::vecS)
- has its vertices stored in a std::vector (due to the second boost::vecS)
- is undirected (due to the boost::undirectedS)
- The vertices have one property: they have a custom type, that is of data type my_vertex (due to the boost::property< boost::vertex_custom_type_t,my_vertex>')
- The edges and graph have no properties
- Edges are stored in a std::list

The boost::adjacency_list has a new, fourth template argument 'boost::property< boost::vertex_custom_type_t,my_vertex>'. This can be read as: "vertices have the property 'boost::vertex_custom_type_t', which is of data type 'my_vertex"'. Or simply: "vertices have a custom type called my_vertex".

6.2 Add a custom vertex

Adding a custom vertex is very similar to adding a named vertex (chapter 4.2).

Algorithm 80 Add a custom vertex

```
#include <boost/graph/adjacency_list.hpp>
#include "install_vertex_custom_type.h"
#include "my_vertex.h"

template <typename graph>
void add_custom_vertex(const my_vertex& v, graph& g)
    noexcept
{
  const auto vd_a = boost::add_vertex(g);
  const auto my_vertex_map = get(boost::
      vertex_custom_type,g);
  my_vertex_map[vd_a] = v;
}
```

When having added a new (abstract) vertex to the graph, the vertex descriptor is used to set the my_vertex in the graph its my_vertex map (using 'boost::get(boost::vertex_custom_type,g)').

6.3 Getting the vertices' my_vertexes[5]

When the vertices of a graph have any associated my_vertex, one can extract these as such:

[5] the name 'my_vertexes' is chosen to indicate this function returns a container of my_vertex

Algorithm 81 Get the vertices' my_vertexes

```
#include <vector>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "install_vertex_custom_type.h"
#include "my_vertex.h"

//TODO: generalize to return any type
template <typename graph>
std::vector<my_vertex> get_vertex_my_vertexes(const graph
    & g) noexcept
{
  std::vector<my_vertex> v;

  const auto my_vertexes_map = get(boost::
      vertex_custom_type,g);

  for (auto p = vertices(g);
    p.first != p.second;
    ++p.first) {
    v.emplace_back(get(my_vertexes_map, *p.first));
  }
  return v;
}
```

The my_vertex object associated with the vertices are obtained from a boost::property_map and then put into a std::vector.

When trying to get the vertices' my_vertex from a graph without my_vertex objects associated, you will get the error 'formed reference to void' (see chapter 10.1).

6.4 Creating K_2 with custom vertices

We reproduce the K_2 with named vertices of chapter 4.4 , but with our custom vertices intead:

Algorithm 82 Creating K_2 as depicted in figure 3

```
#include "install_vertex_custom_type.h"
#include "my_vertex.h"
#include "create_empty_custom_vertices_graph.h"

#include <boost/graph/adjacency_list.hpp>

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<
    boost::vertex_custom_type_t,my_vertex
  >
>
create_custom_vertices_k2_graph() noexcept
{
  auto g = create_empty_custom_vertices_graph();
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto aer = boost::add_edge(vd_a, vd_b, g);
  assert(aer.second);

  //Add names
  auto my_vertexes_map = get(boost::vertex_custom_type,g)
      ;
  my_vertexes_map[vd_a]
    = my_vertex("from","source",0.0,0.0);
  my_vertexes_map[vd_b]
    = my_vertex("to","target",3.14,3.14);

  return g;
}
```

Most of the code is a slight modification of algorithm 35. In the end, the my_vertices are obtained as a boost::property_map and set with two custom my_vertex objects.

6.5 Create an empty graph with custom edges and vertices

Say we want to use our own edge class as graph nodes. This is done in multiple steps:

1. Create a custom edge class, called 'my_edge'

2. Install a new property, called 'edge_custom_type'
3. Use the new property in creating a boost::adjacency_list

6.5.1 Creating the custom edge class

In this example, I create a custom edge class. Here I will show the header file of it, as the implementation of it is not important yet.

Algorithm 83 Declaration of my_edge

```
#ifndef MY_EDGE_H
#define MY_EDGE_H

#include <string>

class my_edge
{
public:
  my_edge(
    const std::string& name = "",
    const std::string& description = "",
    const double width = 1.0,
    const double height = 1.0
  ) noexcept;
  std::string m_name;
  std::string m_description;
  double m_width;
  double m_height;
};

bool operator==(const my_edge& lhs, const my_edge& rhs)
   noexcept;

#endif // MY_EDGE_H
```

my_edge is a class that has multiple properties: two doubles 'm_width' ('m_' stands for member) and 'm_height', and two std::strings m_name and m_description. my_edge is copyable, but cannot trivially be converted to a std::string.

6.5.2 Installing the new property

Installing a new property would have been easier, if 'more C++ compilers were standards conformant' ([2], chapter 3.6, footnote at page 52). Boost.Graph uses the BOOST_INSTALL_PROPERTY macro to allow using a custom property:

Algorithm 84 Installing the edge_custom_type property

```
#include <boost/graph/properties.hpp>

namespace boost {
  enum edge_custom_type_t { edge_custom_type = 3142 };
  BOOST_INSTALL_PROPERTY(edge,custom_type);
}
```

The enum value 3142 must be unique.

6.5.3 Create the empty graph with custom edges and vertices

Algorithm 85 Creating an empty graph with custom vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "install_edge_custom_type.h"
#include "install_vertex_custom_type.h"
#include "my_vertex.h"
#include "my_edge.h"

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<
    boost::vertex_custom_type_t,my_vertex
  >,
  boost::property<
    boost::edge_custom_type_t,my_edge
  >
>
create_empty_custom_edges_and_vertices_graph() noexcept
{
  return boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
      boost::vertex_custom_type_t,my_vertex
    >,
    boost::property<
      boost::edge_custom_type_t,my_edge
    >
  >();
}
```

This graph:

- has its out edges stored in a std::vector (due to the first boost::vecS)
- has its vertices stored in a std::vector (due to the second boost::vecS)
- is undirected (due to the boost::undirectedS)
- The vertices have one property: they have a custom type, that is of data type my_vertex (due to the boost::property< boost::vertex_custom_type_t, my_vertex>')

- The edges have one property: they have a custom type, that is of data type my_edge (due to the boost::property< boost::edge_custom_type_t, my_edge>')
- The graph has no properties
- Edges are stored in a std::list

The boost::adjacency_list has a new, fifth template argument 'boost::property< boost::edge_custom_type_t, my_edge>'. This can be read as: "edges have the property 'boost::edge_custom_type_t', which is of data type 'my_edge"'. Or simply: "edges have a custom type called my_edge".

6.6 Add a custom edge

Adding a custom edge is very similar to adding a named edge (chapter 4.6).

Algorithm 86 Add a custom edge

```
#include <cassert>
#include <boost/graph/adjacency_list.hpp>
#include "install_edge_custom_type.h"
#include "install_vertex_custom_type.h"
#include "my_vertex.h"
#include "my_edge.h"

template <typename graph>
void add_custom_edge(const my_edge& v, graph& g) noexcept
{
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);

  const auto aer = boost::add_edge(vd_a, vd_b, g);
  assert(aer.second);
  const auto my_edge_map = get(boost::edge_custom_type, g
      );
  my_edge_map[aer.first] = v;
}
```

When having added a new (abstract) edge to the graph, the edge descriptor is used to set the my_edge in the graph its my_edge map (using 'boost::get(boost::edge_custom_type,g)').

6.7 Creating K_3 with custom edges and vertices

Instead of using edges with a name, or other properties, here we use a custom edge class called 'my_edge'.

We reproduce the K_3 with named edges and vertices of chapter 4.8 , but with our custom edges and vertices intead:

Algorithm 87 Creating K_3 as depicted in figure 4

```
#include "install_vertex_custom_type.h"
#include "my_vertex.h"
#include "create_empty_custom_edges_and_vertices_graph.h"

#include <boost/graph/adjacency_list.hpp>

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<
    boost::vertex_custom_type_t,my_vertex
  >,
  boost::property<
    boost::edge_custom_type_t,my_edge
  >
>
create_custom_edges_and_vertices_k3_graph() noexcept
{
  auto g = create_empty_custom_edges_and_vertices_graph()
      ;
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto vd_c = boost::add_vertex(g);
  const auto aer_a = boost::add_edge(vd_a, vd_b, g);
  const auto aer_b = boost::add_edge(vd_b, vd_c, g);
  const auto aer_c = boost::add_edge(vd_c, vd_a, g);
  assert(aer_a.second);
  assert(aer_b.second);
  assert(aer_c.second);

  auto my_vertex_map = get(boost::vertex_custom_type,g);
  my_vertex_map[vd_a]
    = my_vertex("top","source",0.0,0.0);
  my_vertex_map[vd_b]
    = my_vertex("right","target",3.14,0);
  my_vertex_map[vd_c]
    = my_vertex("left","target",0,3.14);

  auto my_edge_map = get(boost::edge_custom_type,g);
  my_edge_map[aer_a.first]
    = my_edge("AB","first",0.0,0.0);
  my_edge_map[aer_b.first]
    = my_edge("BC","second",3.14,3.14);
  my_edge_map[aer_c.first]
    = my_edge("CA","third",3.14,3.14);

  return g;
}
```

Most of the code is a slight modification of algorithm 43. In the end, the my_edges and my_vertices are obtained as a boost::property_map and set with the custom my_edge and my_vertex objects.

7 Measuring simple graphs traits of a graph with custom edges and vertices

7.1 Has a my_vertex

Before modifying our vertices, let's first determine if we can find a vertex by its custom type ('my_vertex') in a graph. After obtaing a my_vertex map, we obtain the vertex iterators, dereference these to obtain the vertex descriptors and then compare each vertex its my_vertex with the one desired.

Algorithm 88 Find if there is vertex with a certain my_vertex

```
#include <string>
#include <boost/graph/properties.hpp>
#include "install_vertex_custom_type.h"
#include "my_vertex.h"


template <typename graph>
bool has_vertex_with_my_vertex(
  const my_vertex& v,
  const graph& g
) noexcept
{
  const auto my_vertexes_map = get(boost::
      vertex_custom_type, g);

  for (auto p = vertices(g);
    p.first != p.second;
    ++p.first) {
    if (get(my_vertexes_map, *p.first) == v) {
      return true;
    }
  }
  return false;
}
```

This function can be demonstrated as in algorithm 89, where a certain my_vertex cannot be found in an empty graph. After adding the desired my_vertex, it is found.

Algorithm 89 Demonstration of the 'has_vertex_with_my_vertex' function

```
#include <cassert>
#include <iostream>

#include "add_custom_vertex.h"
#include "create_empty_custom_vertices_graph.h"
#include "has_vertex_with_my_vertex.h"
#include "install_vertex_custom_type.h"
#include "my_vertex.h"

void has_vertex_with_my_vertex_demo() noexcept
{
  auto g = create_empty_custom_vertices_graph();
  assert(!has_vertex_with_my_vertex(my_vertex("Felix"),g)
    );
  add_custom_vertex(my_vertex("Felix"),g);
  assert(has_vertex_with_my_vertex(my_vertex("Felix"),g))
    ;
}
```

Note that this function only finds if there is at least one vertex with that my_vertex: it does not tell how many vertices with that my_vertex exist in the graph.

7.2 Find a vertex with a certain my_vertex

Where STL functions work with iterators, here we obtain a vertex descriptor (see chapter 2.4) to obtain a handle to the desired vertex. Algorithm 90 shows how to obtain a vertex descriptor to the first vertex found with a specific my_vertex value.

Algorithm 90 Find the first vertex with a certain my_vertex

```
#include <string>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "has_vertex_with_my_vertex.h"
#include "install_vertex_custom_type.h"
#include "my_vertex.h"

template <typename graph>
typename boost::graph_traits<graph>::vertex_descriptor
find_first_vertex_with_my_vertex(
  const my_vertex& v,
  const graph& g
) noexcept
{
  assert(has_vertex_with_my_vertex(v, g));
  const auto my_vertexes_map = get(boost::
     vertex_custom_type, g);

  for (auto p = vertices(g);
    p.first != p.second;
    ++p.first) {
    const auto w = get(my_vertexes_map, *p.first);
    if (w == v) { return *p.first; }
  }
  return *vertices(g).second;

}
```

With the vertex descriptor obtained, one can read and modify the vertex and the edges surrounding it. Algorithm 91 shows some examples of how to do so.

Algorithm 91 Demonstration of the 'find_first_vertex_with_my_vertex' function

```
#include <cassert>

#include "create_custom_vertices_k2_graph.h"
#include "find_first_vertex_with_my_vertex.h"

void find_first_vertex_with_my_vertex_demo() noexcept
{
  const auto g = create_custom_vertices_k2_graph();
  const auto vd = find_first_vertex_with_my_vertex(
    my_vertex("from","source",0.0,0.0),
    g
  );
  assert(boost::out_degree(vd,g) == 1);
  assert(boost::in_degree(vd,g) == 1);
}
```

7.3 Get a vertex its my_vertex

To obtain the name from a vertex descriptor, one needs to pull out the my_vertexes[6] map and then look up the vertex of interest.

Algorithm 92 Get a vertex its my_vertex from its vertex descriptor

```
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "install_vertex_custom_type.h"
#include "my_vertex.h"

template <typename graph>
my_vertex get_vertex_my_vertex(
  const typename boost::graph_traits<graph>::
      vertex_descriptor& vd,
  const graph& g
) noexcept
{
  const auto my_vertexes_map = get(boost::
      vertex_custom_type, g);
  return my_vertexes_map[vd];
}
```

[6] Bad English intended: my_vertexes = multiple my_vertex objects, vertices = multiple graph nodes

To use 'get_vertex_my_vertex', one first needs to obtain a vertex descriptor. Algorithm 93 shows a simple example.

Algorithm 93 Demonstration if the 'get_vertex_my_vertex' function

```
#include <cassert>

#include "add_custom_vertex.h"
#include "create_empty_custom_vertices_graph.h"
#include "find_first_vertex_with_my_vertex.h"
#include "get_vertex_my_vertex.h"

void get_vertex_my_vertex_demo() noexcept
{
  auto g = create_empty_custom_vertices_graph();
  const my_vertex name{"Dex"};
  add_custom_vertex(name, g);
  const auto vd = find_first_vertex_with_my_vertex(name,g
     );
  assert(get_vertex_my_vertex(vd,g) == name);
}
```

7.4 Set a vertex its my_vertex

If you know how to get the my_vertex from a vertex descriptor, setting it is just as easy, as shown in algorithm 94.

Algorithm 94 Set a vertex its my_vertex from its vertex descriptor

```
#include <string>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "install_vertex_custom_type.h"
#include "my_vertex.h"

template <typename graph>
void set_vertex_my_vertex(
  const my_vertex& v,
  const typename boost::graph_traits<graph>::
      vertex_descriptor& vd,
  graph& g
) noexcept
{
  const auto my_vertexes_map = get(boost::
      vertex_custom_type, g);
  my_vertexes_map[vd] = v;
}
```

To use 'set_vertex_my_vertex', one first needs to obtain a vertex descriptor. Algorithm 95 shows a simple example.

Algorithm 95 Demonstration if the 'set_vertex_my_vertex' function

```
#include <cassert>

#include "add_custom_vertex.h"
#include "create_empty_custom_vertices_graph.h"
#include "find_first_vertex_with_my_vertex.h"
#include "get_vertex_my_vertex.h"
#include "set_vertex_my_vertex.h"

void set_vertex_my_vertex_demo() noexcept
{
  auto g = create_empty_custom_vertices_graph();
  const my_vertex old_name{"Dex"};
  add_custom_vertex(old_name, g);
  const auto vd = find_first_vertex_with_my_vertex(
      old_name,g);
  assert(get_vertex_my_vertex(vd,g) == old_name);
  const my_vertex new_name{"Diggy"};
  set_vertex_my_vertex(new_name, vd, g);
  assert(get_vertex_my_vertex(vd,g) == new_name);
}
```

7.5 Setting all vertices' my_vertex objects

When the vertices of a graph are associated with my_vertex objects, one can set these my_vertexes as such:

Algorithm 96 Setting the vertices' my_vertexes

```
#include <string>
#include <vector>

#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

#include "install_vertex_custom_type.h"
#include "my_vertex.h"

//TODO: generalize 'my_vertexes'
template <typename graph>
void set_vertex_my_vertexes(
  graph& g,
  const std::vector<my_vertex>& my_vertexes
) noexcept
{
  const auto my_vertex_map = get(boost::
      vertex_custom_type,g);

  auto my_vertexes_begin = std::begin(my_vertexes);
  const auto my_vertexes_end = std::end(my_vertexes);
  for (auto vi = vertices(g);
    vi.first != vi.second;
    ++vi.first, ++my_vertexes_begin)
  {
    assert(my_vertexes_begin != my_vertexes_end);
    put(my_vertex_map, *vi.first,*my_vertexes_begin);
  }
}
```

An impressive feature is that getting the property map holding the graph its names is not a copy, but a reference. Otherwise, modifying 'my_vertexes_map' (obtained by non-reference) would only modify a copy.

7.6 Has a my_edge

Before modifying our edges, let's first determine if we can find an edge by its custom type ('my_edge') in a graph. After obtaing a my_edge map, we obtain the edge iterators, dereference these to obtain the edge descriptors and then compare each edge its my_edge with the one desired.

Algorithm 97 Find if there is an edge with a certain my_edge

```
#include <boost/graph/properties.hpp>
#include "install_edge_custom_type.h"
#include "my_edge.h"

template <typename graph>
bool has_edge_with_my_edge(
  const my_edge& e,
  const graph& g
) noexcept
{
  const auto my_edges_map = get(boost::edge_custom_type,g
    );

  for (auto p = edges(g);
    p.first != p.second;
    ++p.first) {
    if (get(my_edges_map, *p.first) == e) {
      return true;
    }
  }
  return false;
}
```

This function can be demonstrated as in algorithm 98, where a certain my_edge cannot be found in an empty graph. After adding the desired my_edge, it is found.

Algorithm 98 Demonstration of the 'has_edge_with_my_edge' function

```
#include <cassert>

#include "add_custom_edge.h"
#include "create_empty_custom_edges_and_vertices_graph.h"
#include "has_edge_with_my_edge.h"

void has_edge_with_my_edge_demo() noexcept
{
  auto g = create_empty_custom_edges_and_vertices_graph()
      ;
  assert(!has_edge_with_my_edge(my_edge("Edward"),g));
  add_custom_edge(my_edge("Edward"),g);
  assert(has_edge_with_my_edge(my_edge("Edward"),g));
}
```

Note that this function only finds if there is at least one edge with that my_edge: it does not tell how many edges with that my_edge exist in the graph.

7.7 Find a my_edge

Where STL functions work with iterators, here we obtain an edge descriptor (see chapter 2.10) to obtain a handle to the desired edge. Algorithm 99 shows how to obtain an edge descriptor to the first edge found with a specific my_edge value.

Algorithm 99 Find the first edge with a certain my_edge

```
#include <cassert>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "has_edge_with_my_edge.h"
#include "install_edge_custom_type.h"
#include "my_edge.h"

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
find_first_edge_with_my_edge(
  const my_edge& e,
  const graph& g
) noexcept
{
  assert(has_edge_with_my_edge(e, g));
  const auto my_edges_map = get(boost::edge_custom_type,
     g);

  for (auto p = edges(g);
    p.first != p.second;
    ++p.first) {

    if (get(my_edges_map, *p.first) == e) {
      return *p.first;
    }
  }
  return *edges(g).second;
}
```

With the edge descriptor obtained, one can read and modify the edge and the vertices surrounding it. Algorithm 100 shows some examples of how to do so.

Algorithm 100 Demonstration of the 'find_first_edge_with_my_edge' function

```
#include <cassert>

#include "create_custom_edges_and_vertices_k3_graph.h"
#include "find_first_edge_with_my_edge.h"

void find_first_edge_with_my_edge_demo() noexcept
{
  const auto g =
      create_custom_edges_and_vertices_k3_graph();
  const auto ed = find_first_edge_with_my_edge(
    my_edge("AB","first",0.0,0.0),
    g
  );
  assert(boost::source(ed,g) != boost::target(ed,g));
}
```

7.8 Get an edge its my_edge

To obtain the my_edeg from an edge descriptor, one needs to pull out the my_edges map and then look up the my_edge of interest.

Algorithm 101 Get a vertex its my_vertex from its vertex descriptor

```
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "install_edge_custom_type.h"
#include "my_edge.h"

template <typename graph>
my_edge get_edge_my_edge(
  const typename boost::graph_traits<graph>::
      edge_descriptor& vd,
  const graph& g
) noexcept
{
  const auto my_edge_map = get(boost::edge_custom_type, g
      );
  return my_edge_map[vd];
}
```

To use 'get_edge_my_edge', one first needs to obtain an edgedescriptor. Algorithm 102 shows a simple example.

Algorithm 102 Demonstration if the 'get_edge_my_edge' function

```
#include <cassert>

#include "add_custom_edge.h"
#include "create_empty_custom_edges_and_vertices_graph.h"
#include "find_first_edge_with_my_edge.h"
#include "get_edge_my_edge.h"

void get_edge_my_edge_demo() noexcept
{
  auto g = create_empty_custom_edges_and_vertices_graph()
      ;
  const my_edge name{"Dex"};
  add_custom_edge(name, g);
  const auto ed = find_first_edge_with_my_edge(name,g);
  assert(get_edge_my_edge(ed,g) == name);
}
```

7.9 Set an edge its my_edge

If you know how to get the my_edge from an edge descriptor, setting it is just as easy, as shown in algorithm 103.

Algorithm 103 Set an edge its my_edge from its edge descriptor

```
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "install_edge_custom_type.h"
#include "my_edge.h"

template <typename graph>
void set_edge_my_edge(
  const my_edge& name,
  const typename boost::graph_traits<graph>::
      edge_descriptor& vd,
  graph& g
) noexcept
{
  auto my_edge_map = get(boost::edge_custom_type, g);
  my_edge_map[vd] = name;
}
```

To use 'set_edge_my_edge', one first needs to obtain an edgedescriptor.

Algorithm 104 shows a simple example.

Algorithm 104 Demonstration if the 'set_edge_my_edge' function

```
#include <cassert>

#include "add_custom_edge.h"
#include "create_empty_custom_edges_and_vertices_graph.h"
#include "find_first_edge_with_my_edge.h"
#include "get_edge_my_edge.h"
#include "set_edge_my_edge.h"

void set_edge_my_edge_demo() noexcept
{
  auto g = create_empty_custom_edges_and_vertices_graph()
      ;
  const my_edge old_name{"Dex"};
  add_custom_edge(old_name, g);
  const auto vd = find_first_edge_with_my_edge(old_name,g
      );
  assert(get_edge_my_edge(vd,g) == old_name);
  const my_edge new_name{"Diggy"};
  set_edge_my_edge(new_name, vd, g);
  assert(get_edge_my_edge(vd,g) == new_name);
}
```

7.10 Storing a graph with custom vertices as a .dot

If you used the create_custom_vertices_k2_graph function (algorithm 82) to produce a K_2 graph with vertices associated with my_vertex objects, you can store these my_vertexes additionally with algorithm 105:

Algorithm 105 Storing a graph with custom vertices as a .dot file

```
#include <string>
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/properties.hpp>

#include "get_vertex_my_vertexes.h"

///Save a graph with named vertices to a .dot file
template <typename graph>
void save_custom_vertices_graph_to_dot(const graph& g,
    const std::string& filename)
{
  std::ofstream f(filename);
  const auto my_vertexes = get_vertex_my_vertexes(g);
  boost::write_graphviz(
    f,
    g,
    [my_vertexes](std::ostream& out, const auto& v) {
      const my_vertex m{my_vertexes[v]};
      out << "[label=\""
        << m.m_name
        << ","
        << m.m_description
        << ","
        << m.m_x
        << ","
        << m.m_y
        << "\"]";
    }
  );
}
```

Note that this algorithm uses C++17.

The .dot file created is displayed in algorithm 106:

Algorithm 106 .dot file created from the create_custom_vertices_k2_graph function (algorithm 35)

```
graph G {
0[label="from,source,0,0"];
1[label="to,target,3.14,3.14"];
0--1 ;
}
```

This .dot file corresponds to figure 106:

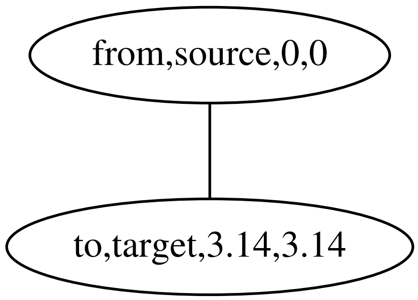

Figure 7: .svg file created from the create_custom_vertices_k2_graph function (algorithm 82)

8 Measuring more complex graphs traits

8.1 Count the number of self-loops

9 Misc functions

9.1 Getting a data type as a std::string

This function will only work under GCC.

Algorithm 107 Getting a data type its name as a std::string

```
#include <string>
#include <typeinfo>
#include <cstdlib>
#include <cxxabi.h>

//From http://stackoverflow.com/questions/1055452/c-get-
    name-of-type-in-template
//Thanks to m-dudley ( http://stackoverflow.com/users
    /111327/m-dudley )
template<typename T>
std::string get_type_name() noexcept
{
  std::string tname = typeid(T).name();
  int status = -1;
  char * const demangled_name{
    abi::__cxa_demangle(tname.c_str(), NULL, NULL, &
        status)
  };
  if(status == 0) {
    tname = demangled_name;
    std::free(demangled_name);
  }
  return tname;
}
```

10 Errors

Some common errors.

10.1 Formed reference to void

This compile-time error occurs when you create a graph without a certain property, then subsequently reading that property, as in algorithm 108:

Algorithm 108 Creating the error 'formed reference to void'

```
#include "create_k2_graph.h"
#include "get_vertex_names.h"

void formed_reference_to_void() noexcept
{
  get_vertex_names(create_k2_graph());
}
```

In algorithm 108 a graph is created with vertices of no properties. Then the names of these vertices, which do not exists, are tried to be read. If you want to read the names of the vertices, supply a graph that has this property.

10.2 No matching function for call to 'clear_out_edges'

This compile-time error occurs when you want to clear the outward edges from a vertex in an undirected graph.

Algorithm 109 Creating the error 'formed reference to void'

```
#include "create_k2_graph.h"

void no_matching_function_for_call_to_clear_out_edges()
   noexcept
{
  auto g = create_k2_graph();
  const auto vd = *boost::vertices(g).first;
  boost::clear_in_edges(vd,g);
}
```

In algorithm 109 an undirected graph is created, a vertex descriptor is obtained, then its out edges are tried to be cleared. Either use a directed graph (which has out edges), or use the 'boost::clear_vertex' function instead.

10.3 No matching function for call to 'clear_in_edges'

See chapter 10.2.

References

[1] John Lakos. *Large-scale C++ software design*, volume 10. Addison-Wesley Reading, 1996.

[2] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.

[3] Bjarne Stroustrup. *The C++ Programming Language (3rd edition)*. 1997.

Index

K_2 with named vertices, create, 28
K_2, create, 15
K_3 with named edges and vertices, create, 35
'demo' function, 3
'do' function, 3

Add a vertex, 7
Add an edge, 11
Add named edge, 32
Add named vertex, 25
add_custom_edge, 70
add_custom_vertex, 64
add_edge, 11
add_edge_demo, 12
add_named_edge, 33
add_named_edge_demo, 33
add_named_vertex, 25
add_named_vertex_demo, 26
add_vertex, 7
add_vertex_demo, 7
aer_, 12
assert, 11

boost::add_edge, 11, 16, 33
boost::add_edge result, 12
boost::add_vertex, 7, 16
boost::adjacency_list, 5, 24, 31, 63, 70
boost::adjacency_matrix, 5
boost::clear_in_edges, 46
boost::clear_out_edges, 46
boost::clear_vertex, 46
boost::degree, 41
boost::edge_custom_type, 70
boost::edge_custom_type_t, 70
boost::edge_name_t, 31
boost::edges, 12, 14
boost::get, 4, 26, 64, 70
boost::in_degree, 41
boost::num_edges, 18
boost::num_vertices, 17
boost::out_degree, 41
boost::property, 24, 31, 63, 69, 70
boost::remove_edge, 54, 56
boost::remove_vertex, 48
boost::undirectedS, 6, 24, 31, 63, 69
boost::vecS, 6, 24, 31, 63, 69
boost::vertex_custom_type, 64
boost::vertex_custom_type_t, 63, 69
boost::vertex_name, 26
boost::vertex_name_t, 24, 31
boost::vertices, 8
BOOST_INSTALL_PROPERTY, 62, 67

C++17, 59, 87
clear_first_vertex_with_name, 47
clear_first_vertex_with_name_demo, 47
Counting the number of edges, 18
Counting the number of vertices, 17
Create K_2, 15
Create K_2 with named vertices, 28
Create K_3 with named edges and vertices, 35
Create .dot from graph, 21
Create .dot from graph with custom vertices, 86
Create .dot from graph with named edges and vertices, 59
Create .dot from graph with named vertices, 57
Create an empty directed graph, 5
Create an empty graph, 5
Create an empty graph with named edges and vertices, 30
Create an empty graph with named vertices, 23
create_custom_edges_and_vertices_k3_graph, 72
create_custom_vertices_k2_graph, 66
create_empty_custom_vertices_graph, 63, 69
create_empty_directed_graph, 5
create_empty_directed_graph_demo, 5

create_empty_named_edges_and_vertices_graph, 31
create_empty_named_edges_and_vertices_graph_demo, 32
create_empty_named_vertices_graph, 24
create_empty_named_vertices_graph_demo, 25
create_empty_undirected_graph, 6
create_empty_undirected_graph_demo, 6
create_k2_graph, 16
create_k2_graph_demo, 17
create_named_edges_and_vertices_k3_graph, 36
create_named_edges_and_vertices_k3_graph_demo, 37
create_named_vertices_k2_graph, 29
create_named_vertices_k2_graph_demo, 30

Declaration, my_edge, 67
Declaration, my_vertex, 62

ed_, 14
Edge descriptor, 13
Edge descriptors, get, 14
Edge iterator, 12
Edge iterator pair, 12
Edge, add, 11
edge_custom_type, 67
Edges, counting, 18
eip_, 12
Empty directed graph, create, 5
Empty graph with named edges and vertices, create, 30
Empty graph with named vertices, create, 23
Empty graph, create, 5

find_first_edge_by_name, 51
find_first_edge_by_name_demo, 52
find_first_edge_with_my_edge, 83
find_first_edge_with_my_edge_demo, 84
find_first_vertex_by_name, 40
find_first_vertex_by_name_demo, 41, 76
find_first_vertex_with_my_vertex, 75
formed_reference_to_void, 90

get, 4
Get edge descriptors, 14
get_edge_descriptors, 14
get_edge_descriptors_demo, 15
get_edge_my_edge, 84
get_edge_my_edge_demo, 85
get_edge_name, 52
get_edge_name_demo, 53
get_edge_names, 34
get_edge_names_demo, 35
get_edges, 13
get_edges_demo, 13
get_first_vertex_with_name_out_degree, 42
get_first_vertex_with_name_out_degree_demo, 42
get_n_edges, 19
get_n_edges_demo, 19
get_n_vertices, 18
get_n_vertices_demo, 18
get_type_name, 89
get_vertex_descriptors, 10
get_vertex_descriptors_demo, 10
get_vertex_my_vertex, 76
get_vertex_my_vertex_demo, 77
get_vertex_my_vertexes, 65
get_vertex_name, 43
get_vertex_name_demo, 43
get_vertex_names, 27
get_vertex_names_demo, 28
get_vertex_out_degrees, 20
get_vertex_out_degrees_demo, 20
get_vertices, 8
get_vertices_demo, 9

has_edge_with_my_edge, 81
has_edge_with_my_edge_demo, 82
has_edge_with_name, 50
has_vertex_with_my_vertex, 73
has_vertex_with_my_vertex_demo, 74
has_vertex_with_name, 38

has_vertex_with_name_demo, 39, 50

install_vertex_custom_type, 62, 68

m_, 62, 67
macro, 62, 67
member, 62, 67
my_edge, 67, 70
my_edge declaration, 67
my_edge.h, 67
my_vertex, 62, 63, 69
my_vertex declaration, 62
my_vertex.h, 62

Named edge, add, 32
Named edges and vertices, create empty graph, 30
Named vertex, add, 25
Named vertices, create empty graph, 23

pi, 63, 68, 88

Reference to Fantastic Four, 32, 33
Reference to Superman, 26
remove_edge_between_vertices_with_names, 56
remove_edge_between_vertices_with_names_demo, 57
remove_first_edge_with_name, 55
remove_first_edge_with_name_demo, 55
remove_first_vertex_with_name, 48
remove_first_vertex_with_name_demo, 49

Save graph as .dot, 21
Save graph with custom vertices as .dot, 86
Save graph with name edges and vertices as .dot, 59
Save graph with name vertices as .dot, 57
save_custom_vertices_graph_to_dot, 87
save_graph_to_dot, 21
save_graph_to_dot_demo, 21
save_named_edges_and_vertices_graph_to_dot, 59
save_named_vertices_graph_to_dot, 57
Set vertices my_vertexes, 79
Set vertices names, 45
set_edge_my_edge, 85
set_edge_my_edge_demo, 86
set_edge_name, 53
set_edge_name_demo, 54
set_vertex_my_vertex, 78
set_vertex_my_vertex_demo, 79
set_vertex_my_vertexes, 80
set_vertex_name, 44
set_vertex_name_demo, 45
set_vertex_names, 46
std::pair, 11

vd, 11
vd_, 8
Vertex descriptor, 7, 11
Vertex descriptors, get, 9
Vertex iterator, 8
vertex iterator, 10, 14
Vertex iterator pair, 8
Vertex, add, 7
Vertex, add named, 25
vertex_custom_type, 61
Vertices, counting, 17
Vertices, set my_vertexes, 79
Vertices, set names, 45
vip_, 8