

Boost.Graph tutorial

Richel Bilderbeek

December 8, 2015

Contents

1	Introduction	2
1.1	Coding style used	3
1.2	Pitfalls	3
2	Building graphs and basic functions	3
2.1	Creating an empty (directed) graph	4
2.2	Creating an empty undirected graph	5
2.3	Add a vertex	5
2.4	Vertex descriptors	6
2.5	Get the vertices	6
2.6	Get all vertex descriptors	7
2.7	Add an edge	8
2.8	boost::add_edge result	8
2.9	Getting the edges	9
2.10	Edge descriptors	9
2.11	Get all edge descriptors	9
2.12	Creating K_2 , a fully connected graph with two vertices	10
2.13	Creating an empty graph with named vertices	11
2.14	Add a vertex with a name	12
2.15	Creating K_2 with named vertices	13
2.16	Creating an empty graph with named edges and vertices	14
2.17	Add an edge with a name	16
2.18	Creating K_3 with named edges and vertices	16
2.19	Create an empty graph with custom vertices	18
2.19.1	Creating the custom vertex class	18
2.19.2	Installing the new property	19
2.19.3	Create the empty graph with custom vertices	20
2.20	Add a custom vertex	21
2.21	Creating K_2 with custom vertices	21
2.22	Create an empty graph with custom edges and vertices	22
2.22.1	Creating the custom edge class	23
2.22.2	Installing the new property	23

2.22.3	Create the empty graph with custom edges and vertices .	25
2.23	Add a custom edge	26
2.24	Creating K_3 with custom edges and vertices	26
3	Measuring simple graphs traits	29
3.1	Counting the number of vertices	29
3.2	Counting the number of edges	29
3.3	Getting the vertices' out degree	29
3.4	Getting the vertices' names	30
3.5	Getting the edges' names	31
3.6	Getting the vertices' my_vertexes	32
3.7	Count vertex name	33
3.8	Find a vertex by its name	33
3.9	Get a named vertex its in-degree	33
3.10	Get a named vertex its out-degree	34
3.11	Count vertex my_vertex	34
3.12	Find a my_vertex	34
3.13	Find the vertices connected to a certain my_vertex	34
4	Modifying simple graphs traits	34
4.1	Setting all vertices' names	34
4.2	Setting all vertices' my_vertex objects	35
4.3	Replace a vertex its name	36
4.4	Replace an edge its name	36
4.5	Replace a my_vertex	37
4.6	Clear a named vertex	37
4.7	Remove a named vertex	37
4.8	Remove a named edge	37
4.9	Remove a my_vertex	37
5	Visualizing graphs	37
5.1	Storing a graph as a .dot	38
5.2	Storing a graph with named vertices as a .dot	39
5.3	Storing a graph with named vertices and edges as a .dot	41
5.4	Storing a graph with custom vertices as a .dot	43
6	Measuring more complex graphs traits	44
6.1	Count the number of self-loops	44
7	Errors	44
7.1	Formed reference to void	44

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.

1.1 Coding style used

I use the coding style from the Core C++ Guidelines.

I prefer to use the keyword `auto` over doubling the lines of code for using statements. There are plenty of algorithms that return an explicit data type, which can be used for reference.

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

1.2 Pitfalls

The choice between `'boost::get'`, `'std::get'` and `'get'`. AFAIKS, when in doubt, use `'get'`.

2 Building graphs and basic functions

Boost.Graph is about creating graphs. In this chapter we create graphs, starting from simple to more complex:

- 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
- An empty (undirected) graph that allows for vertices with names: see chapter 2.13
- K_2 with named vertices: see chapter 2.15
- An empty (undirected) graph that allows for edges and vertices with names: see chapter 2.16
- K_3 with named edges and vertices: see chapter 2.18
- An empty (undirected) graph that allows for custom vertices: see chapter 2.19
- K_2 with custom vertices: see chapter 2.21
- An empty (undirected) graph that allows for custom edges and vertices: see chapter 2.22

- K_3 with custom edges and vertices: see chapter 2.24

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 Creating an empty (directed) graph

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

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

Congratulations, you've just created a `boost::adjacency_list` in which:

- The out edges are stored in a `std::vector`
- The vertices are stored in a `std::vector`
- The graph is directed
- Vertices, edges and graph have no properties
- 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 a trivial empty graph:

Algorithm 2 Creating an empty 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
    >();
}
```

Congratulations, you've just created a `boost::adjacency_list` in which:

- The out edges are stored in a `std::vector`
- The vertices are stored in a `std::vector`
- The graph is undirected
- Vertices, edges and graph have no properties
- 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`.

2.3 Add a vertex

To add a vertex to a graph, the `boost::add_vertex` function is used as such:

Algorithm 3 Adding a vertex to a graph

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

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

Note that `boost::add_vertex` 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 28), the vertex its name (chapter 29), or a custom vertex property (chapter 31)

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 through 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. 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 4 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)
{
    return boost::vertices(g);
}
```

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

2.6 Get all vertex descriptors

Vertex descriptors are obtained from dereferencing vertex iterators:

Algorithm 5 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;
}
```

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

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 within graph (vertex descriptors are looked at in more details in chapter 2.4). Algorithm 6 adds two vertices to a graph, and connects these two using `boost::add_edge`:

Algorithm 6 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)
{
    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 6 we assert that this insertion was successful. Insertion can fail if an edge is already present and duplicates are not allowed.

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. Working on the edges of a graph is done through 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 prefixes with 'eip_', for example 'eip_1'.

Algorithm 7 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)
{
    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 immediately.

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 prefixes 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 8 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.

2.12 Creating K_2 , a fully connected graph with two vertices

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.

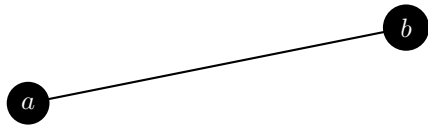


Figure 1: K_2 : a fully connected graph with two vertices named a and b

To create K_2 , the following code can be used:

Algorithm 9 Creating K_2 as depicted in figure 1

```
#include "create_k2_graph.h"
#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;
}
```

Note that this code has more lines of using statements than actual code! In this code, the third template argument of `boost::adjacency_list` is `boost::undirectedS`, to select (that is what the S means) for an undirected graph. Adding a vertex with `boost::add_vertex` results in a vertex descriptor, which is a handle to the vertex added to the graph. Two vertex descriptors are then used to add an edge to the graph. Adding an edge using `boost::add_edge` returns two things: an edge descriptor and a boolean indicating success. In the code example, we assume insertion is successful.

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

2.13 Creating an empty graph with named vertices

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

Algorithm 10 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`”.

2.14 Add a vertex with a name

Adding a vertex without a name was trivially easy (see chapter 3). Adding a vertex with a name is less easy:

Algorithm 11 Add a vertex with a name

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

template <typename graph>
void add_named_vertex(graph& g, const std::string& name)
{
    const auto vd_a = boost::add_vertex(g);
    auto vertex_name_map = boost::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, multiple things need to be done. When adding a new vertex to the graph, the vertex descriptor 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.

2.15 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 2:

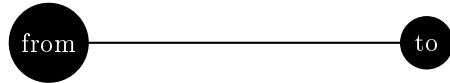


Figure 2: K_2 : a fully connected graph with two vertices with the text 'from' and 'to'

To create K_2 , the following code can be used:

Algorithm 12 Creating K_2 as depicted in figure 2

```
#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 = boost::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 9. In the end, the names are obtained as a `boost::property_map` and set.

2.16 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 13 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`".

2.17 Add an edge with a name

Adding an edge with a name:

Algorithm 14 Add a vertex with a name

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

#include <cassert>

template <typename graph>
void add_named_edge(graph& g, const std::string&
    edge_name)
{
    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 = boost::get(boost::edge_name, g);
    edge_name_map[aer.first] = edge_name;
}
```

In this code snippet, the edge descriptor when using 'boost::add_edge' is used as a key to change the edge name map.

2.18 Creating K_3 with named edges and vertices

We extend the graph K_2 with named vertices of chapter 2.15 by adding names to the edges, as depicted in figure 3:

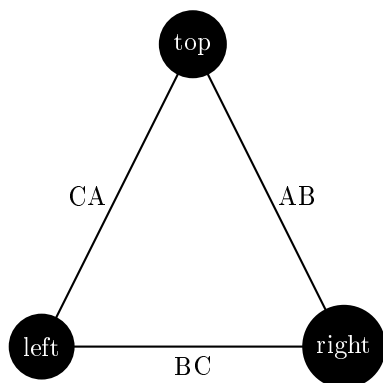


Figure 3: K_3 : a fully connected graph with three named edges and vertices

To create K_3 , the following code can be used:

Algorithm 15 Creating K_3 as depicted in figure 3

```
#include "create_named_edges_and_vertices_k3_graph.h"

#include "create_empty_named_edges_and_vertices_graph.h"

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 = boost::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 = boost::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 12. In the end, the edge names are obtained as a `boost::property_map` and set.

2.19 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`

2.19.1 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.

Algorithm 16 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.

2.19.2 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 17 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.

2.19.3 Create the empty graph with custom vertices

Algorithm 18 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`".

2.20 Add a custom vertex

Adding a custom vertex is very similar to adding a named vertex (chapter 2.14).

Algorithm 19 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(graph& g, const my_vertex& v)
{
    const auto vd_a = boost::add_vertex(g);
    const auto my_vertex_map = boost::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)').

2.21 Creating K_2 with custom vertices

We reproduce the K_2 with named vertices of chapter 2.15 , but with our custom vertices instead:

Algorithm 20 Creating K_2 as depicted in figure 2

```
#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_vertex_name_map = boost::get(boost::
        vertex_custom_type, g);
    my_vertex_name_map[vd_a]
        = my_vertex("from", "source", 0.0, 0.0);
    my_vertex_name_map[vd_b]
        = my_vertex("to", "target", 3.14, 3.14);

    return g;
}
```

Most of the code is a slight modification of algorithm 12. In the end, the `my_vertices` are obtained as a `boost::property_map` and set with two custom `my_vertex` objects.

2.22 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

2.22.1 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 21 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.

2.22.2 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 22 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.

2.22.3 Create the empty graph with custom edges and vertices

Algorithm 23 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`".

2.23 Add a custom edge

Adding a custom edge is very similar to adding a named edge (chapter 2.17).

Algorithm 24 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(graph& g, const my_edge& v)
{
    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 = boost::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)`').

2.24 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 2.18 , but with our custom edges and vertices intead:

Algorithm 25 Creating K_3 as depicted in figure 3

```
#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 = boost::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 = boost::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 15. 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.

3 Measuring simple graphs traits

Measuring simple traits of the graphs created allows you to debug your code.

3.1 Counting the number of vertices

Use `boost::num_vertices`, as shown here:

Algorithm 26 Count the number 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)
{
    return static_cast<int>(boost::num_vertices(g));
}
```

3.2 Counting the number of edges

Use `boost::num_edges`, as shown here:

Algorithm 27 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)
{
    return static_cast<int>(boost::num_edges(g));
}
```

3.3 Getting the vertices' out degree

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

Algorithm 28 Get the vertices' out degrees

```
#include <vector>

template <typename graph>
std::vector<int> get_vertex_out_degrees(const graph& g)
{
    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 `int` for the sake of gaining that one more bit [3]².

3.4 Getting the vertices' names

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

¹Chapter 9.2.2

²Chapter 4.4

Algorithm 29 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)
{
    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 7.1).

3.5 Getting the edges' names

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

Algorithm 30 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)
{
    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`.

When trying to get the edges' names from a graph without vertices with names, you will get the error 'formed reference to void' (see chapter 7.1).

3.6 Getting the vertices' `my_vertexes`³

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

³the name 'my_vertexes' is chosen to indicate this function returns a container of `my_vertex`

Algorithm 31 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)
{
    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 7.1).

3.7 Count vertex name

count_vertices_with_name

3.8 Find a vertex by its name

find_vertex_with_name

3.9 Get a named vertex its in-degree

get_named_vertex_in_degree

- degree_size_type in_degree(vertex_descriptor u, const adjacency_list& g) . Returns the in-degree of a vertex

3.10 Get a named vertex its out-degree

`get_named_vertex_out_degree`

- `degree_size_type in_degree(vertex_descriptor u, const adjacency_list& g)` . Returns the in-degree of a vertex

3.11 Count vertex `my_vertex`

`count_vertex_my_vertex`

3.12 Find a `my_vertex`

`find_my_vertex`

3.13 Find the vertices connected to a certain `my_vertex`

`find_vertices_connected_to_my_vertex`

4 Modifying simple graphs traits

It is useful to be able to modify every aspect of a graph. Adding nodes and edges are found in earlier chapters.

4.1 Setting all vertices' names

When the vertices of a graph have named vertices, one set their names as such:

Algorithm 32 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
)
{
    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.

4.2 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 33 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
)
{
    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.

4.3 Replace a vertex its name

rename_vertex

4.4 Replace an edge its name

rename_edge

4.5 Replace a my_vertex

replace_my_vertex

4.6 Clear a named vertex

clear_named_vertex

- void clear_vertex(vertex_descriptor u, adjacency_list& g) . Removes all edges to and from u
- void clear_out_edges(vertex_descriptor u, adjacency_list& g) . Removes all outgoing edges from vertex u in the directed graph g (not applicable for undirected graphs)
- void clear_in_edges(vertex_descriptor u, adjacency_list& g) . Removes all incoming edges from vertex u in the directed graph g (not applicable for undirected graphs)

4.7 Remove a named vertex

remove_named_vertex

4.8 Remove a named edge

remove_named_vertex

- void remove_edge(vertex_descriptor u, vertex_descriptor v, adjacency_list& g) . Removes an edge from g
- void remove_edge(edge_descriptor e, adjacency_list& g) . Removes an edge from g

4.9 Remove a my_vertex

remove_my_vertex

- void remove_vertex(vertex_descriptor u, adjacency_list& g) . Removes a vertex from graph g (It is expected that all edges associated with this vertex have already been removed using clear_vertex or another appropriate function.)

5 Visualizing graphs

Before graphs are visualized, they are stored as a file first. Here, I use the .dot file format.

5.1 Storing a graph as a .dot

Graph are easily saved to a .dot file:

Algorithm 34 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)
{
    std::ofstream f(filename);
    boost::write_graphviz(f,g);
}
```

Using the `create_k2_graph` function (algorithm 9) to create a K_2 graph, the .dot file created is displayed in algorithm 35:

Algorithm 35 .dot file created from the `create_k2_graph` function (algorithm 9)

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

This .dot file corresponds to figure 4:

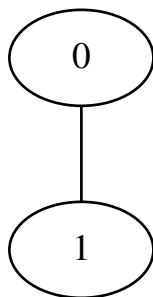


Figure 4: .svg file created from the `create_k2_graph` function (algorithm 9)

If you used the `create_named_vertices_k2_graph` function (algorithm 12) to produce a K_2 graph with named vertices, you see that the `.dot` file does not have stored the vertex names:

Algorithm 36 `.dot` file created from the `create_named_vertices_k2_graph` function (algorithm 12)

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

So, the `'save_graph_to_dot'` function (algorithm 34) saves the structure of the graph.

5.2 Storing a graph with named vertices as a `.dot`

If you used the `create_named_vertices_k2_graph` function (algorithm 12) to produce a K_2 graph with named vertices, you can store these names additionally with algorithm 37:

Algorithm 37 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)
{
    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 38:

Algorithm 38 .dot file created from the create_named_vertices_k2_graph function (algorithm 12)

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

This .dot file corresponds to figure 5:

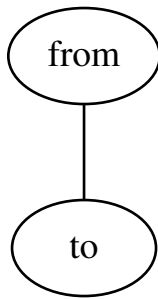


Figure 5: .svg file created from the create_k2_graph function (algorithm 12)

If you used the create_named_edges_and_vertices_k3_graph function (algorithm 15) 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 39 .dot file created from the create_named_edges_and_vertices_k3_graph function (algorithm 15)

```
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 34) saves only the structure of the graph and its vertex names.

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

If you used the `create_named_edges_and_vertices_k3_graph` function (algorithm 15) to produce a K_3 graph with named edges and vertices, you can store these names additionally with algorithm 40:

Algorithm 40 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 41:

Algorithm 41 .dot file created from the create_named_edges_and_vertices_k3_graph function (algorithm 12)

```
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:

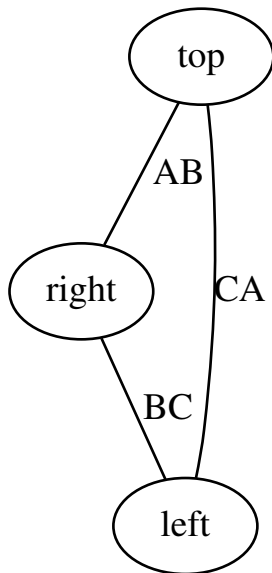


Figure 6: .svg file created from the create_named_edges_and_vertices_k3_graph function (algorithm 12)

If you created a graph with edges more complex than just a name, you will still just write these to the .dot file. Chapter 5.4 shows how to write custom vertices to a .dot file.

So, the 'save_named_edges_and_vertices_graph_to_dot' function (algorithm 34) saves only the structure of the graph and its edge and vertex names.

5.4 Storing a graph with custom vertices as a .dot

If you used the `create_custom_vertices_k2_graph` function (algorithm 20) to produce a K_2 graph with vertices associated with `my_vertex` objects, you can store these `my_vertexes` additionally with algorithm 42:

Algorithm 42 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 43:

Algorithm 43 .dot file created from the `create_custom_vertices_k2_graph` function (algorithm 12)

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

This .dot file corresponds to figure 43:

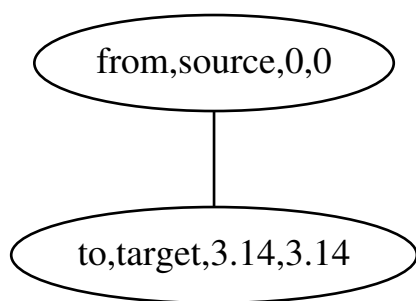


Figure 7: .svg file created from the `create_custom_vertices_k2_graph` function (algorithm 20)

6 Measuring more complex graphs traits

6.1 Count the number of self-loops

7 Errors

Some common errors.

7.1 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 44:

Algorithm 44 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 44 a graph is created with vertices of no properties. Then the names of these vertices, which do not exist, are tried to be read. If you want to read the names of the vertices, supply a graph that has this property.

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, 13
- K_2 , create, 10
- K_3 with named edges and vertices, create, 16
-
- Add a vertex, 5
- Add an edge, 8
- Add named edge, 16
- Add named vertex, 12
- add_custom_edge, 26
- add_custom_vertex, 21
- add_edge, 8
- add_named_edge, 16
- add_named_vertex, 13
- add_vertex, 6
- aer_, 8
- assert, 8
-
- boost::add_edge, 8, 11, 16
- boost::add_edge result, 8
- boost::add_vertex, 5, 11
- boost::adjacency_list, 4, 5, 11, 12, 15, 20, 26
- boost::adjacency_matrix, 4, 5
- boost::edge_custom_type, 26
- boost::edge_custom_type_t, 26
- boost::edge_name_t, 15
- boost::edges, 9, 10
- boost::get, 13, 21, 26
- boost::num_edges, 29
- boost::num_vertices, 29
- boost::property, 12, 15, 20, 25, 26
- boost::undirectedS, 11, 12, 15, 20, 25
- boost::vecS, 12, 15, 20, 25
- boost::vertex_custom_type, 21
- boost::vertex_custom_type_t, 20, 25
- boost::vertex_name, 13
- boost::vertex_name_t, 12, 15
- boost::vertices, 6
- BOOST_INSTALL_PROPERTY, 19, 23
-
- C++17, 41, 43
-
- Counting the number of edges, 29
- Counting the number of vertices, 29
- Create K_2 , 10
- Create K_2 with named vertices, 13
- Create K_3 with named edges and vertices, 16
- Create .dot from graph, 38
- Create .dot from graph with custom vertices, 43
- Create .dot from graph with named edges and vertices, 41
- Create .dot from graph with named vertices, 39
- Create an empty directed graph, 4
- Create an empty graph, 5
- Create an empty graph with named edges and vertices, 14
- Create an empty graph with named vertices, 11
- create_custom_edges_and_vertices_k3_graph, 28
- create_custom_vertices_k2_graph, 22
- create_empty_custom_vertices_graph, 20, 25
- create_empty_directed_graph, 4
- create_empty_named_edges_and_vertices_graph, 15
- create_empty_named_vertices_graph, 12
- create_empty_undirected_graph, 5
- create_k2_graph, 11
- create_named_edges_and_vertices_k3, 17
- create_named_vertices_k2_graph, 14
-
- Declaration, my_edge, 23
- Declaration, my_vertex, 18
-
- ed_, 9
- Edge descriptor, 9
- edge descriptor, 16
- Edge descriptors, get, 9
- Edge iterator, 9

- Edge iterator pair, 9
- Edge, add, 8
- edge_custom_type, 23
- Edges, counting, 29
- eip_, 9
- Empty directed graph, create, 4
- Empty graph with named edges and vertices, create, 14
- Empty graph with named vertices, create, 11
- Empty graph, create, 5
- formed_reference_to_void, 45
- Get edge descriptors, 9
- get_edge_descriptors, 10
- get_edge_names, 32
- get_edges, 9
- get_n_edges, 29
- get_n_vertices, 29
- get_vertex_descriptors, 7
- get_vertex_my_vertexes, 33
- get_vertex_names, 31
- get_vertex_out_degrees, 30
- get_vertices, 7
- install_vertex_custom_type, 19, 24
- m_, 19, 23
- macro, 19, 23
- member, 19, 23
- my_edge, 23, 26
- my_edge declaration, 23
- my_edge.h, 23
- my_vertex, 18, 20, 25
- my_vertex declaration, 18
- my_vertex.h, 18
- Named edge, add, 16
- Named edges and vertices, create empty graph, 14
- Named vertex, add, 12
- Named vertices, create empty graph, 11
- pi, 19, 24, 44
- Save graph as .dot, 38
- Save graph with custom vertices as .dot, 43
- Save graph with name edges and vertices as .dot, 41
- Save graph with name vertices as .dot, 39
- save_custom_vertices_graph_to_dot, 43
- save_graph_to_dot, 38
- save_named_edges_and_vertices_graph_to_dot, 41
- save_named_vertices_graph_to_dot, 39
- Set vertices my_vertexes, 35
- Set vertices names, 34
- set_vertex_my_vertexes, 36
- set_vertex_names, 35
- std::pair, 8
- vd, 8
- vd_, 6
- Vertex descriptor, 6, 8, 11
- Vertex descriptors, get, 7
- Vertex iterator, 6
- vertex iterator, 8, 10
- Vertex iterator pair, 6
- Vertex, add, 5
- Vertex, add named, 12
- vertex_custom_type, 18
- Vertices, counting, 29
- Vertices, set my_vertexes, 35
- Vertices, set names, 34
- vip_, 6