1. Dummy Chapter

1. h1 title

2. h2 title

3. h3 title

4. h3(2) title

5. h4 title

6. h4(2) title

7. h5 title

8. h5(2) title

9. h6 title

10. h6(2) title

11. h6(3) title

12. h2(2) title

13. h3 title

14. h4 title

15. h5 title

16. h6 title
p paragraph

```
pre paragraph
```

div paragraph (class=b)

a link: hyperlink
left aligned

right aligned

centered

paragraph with **bold (b)** elements

paragraph with *emphasized (em)* elements

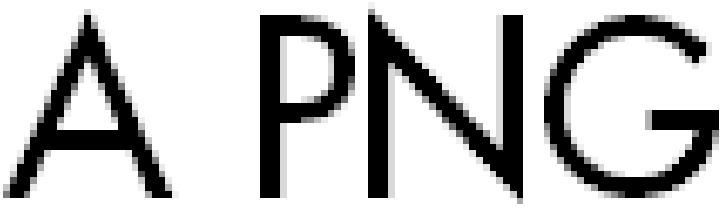
paragraph with typewriter (tt) elements

paragraph with code (code) elements

paragraph with span elements (class = em)

- Unordered
- list

1. Ordered
2. list
    - sub-item 1
    - sub item 2
2. ordered item



2. A Wiki page

The tutorial graph example of the [[BGL]] is the problem of tracking
dependencies for the compilation of files (

http://www.boost.org/libs/graph/doc/file_dependency_example.html).

In this page, we will transform the file dependency problem following the
[[BGL]] tutorial.

1. The File Dependency Problem:
The File Dendency Problem is introduced as such:

2. Example graph
The graph described below represents the
source files of a killer application. Eeach source file (header, source,
object) is represented by a vertex. The edges represent which files are
included into others. The arrow direction means "used by" and the opposite of
the arrow direction means "depends on".

The entire source code of the example is
available in the ~QuickGraphTest application (the ~FileDependencyTest class).

3. Graph Setup
To represent the graph, we are going to use
the AdjacencyGraph class which contains two methods that we need:

1. ~AddVertex which adds a new vertex to the graph,

2. ~AddEdge(

   u,v) which creates a new edge from
   vertex

   u to

   v.

```
 // create a new adjacency graph
 AdjacencyGraph g = new AdjacencyGraph(new ~VertexAndEdgeProvider(), false);

// a vertex name map to store the file names
~VertexStringDictionary names = new ~VertexStringDictionary();

// adding files and storing names
[[IVertex]] zig_cpp = g.~AddVertex();
names[zig_cpp]="zip.cpp";
[[IVertex]] boz_h = g.~AddVertex();
names[boz_h]="boz.h";

// adding dependencies
g.~AddEdge(dax_h, foo_cpp);
g.~AddEdge(dax_h, bar_cpp);
...
```

4. Drawing the files

In order to have the nice figure above, we first use a special algorithm, GraphvizAlgorithm, that outputs the graph and renders it using the GraphViz library:

```
// outputing graph to png
GraphvizAlgorithm gw = new GraphvizAlgorithm(
    g,                  // graph to draw
    "filedependency",       // output file prefix
    ".",                // output file path
    GraphvizImageType.Png   // output file type
    );
    // outputing to graph.
    gw.Write();
```

Compiling and executing the code leads to the following image:

This not exactly what we expected since it is not the file names that appear bu some mysterious numbers. In fact, the graphviz algorithm does not know that the vertices have names so it uses their hash code number to name them. In order to add the names on the graph you must create a

visitor for the algorithm.

5. Graphivz output with names

As mentionned before, it is your job to tell to the GraphvizAlgorithm the name of each vertices. In the [[BGL]], this is done by defining Visitors. Visitors

are class instance, whose member function are called on specific events. For instance, the graphviz algorithm has 3 events: ~WriteGraph which is called to set-up the global graph properties, ~WriteVertex which is called on each vertex and ~WriteEdge which is called on each edge.

In QuickGraph, the ideas remains the same but there are no visitors as such anymore. The algorithms now use

events and delegates to trigger the ... events. (This topic is more detailled in the QuickGraphAndTheBoostGraphLibrary page). So basically what you need to do is write a method that will be attached to the ~WriteVertex event and will specify the vertex name.

```
public class ~FileDependencyTest
{
    private ~GraphvizVertex m_Vertex;
    private ~VertexStringDictionary m_Names;
    public ~FileDependencyTest()
    {
        m_Vertex = new ~GraphvizVertex();
        m_Names = new ~VertexStringDictionary();
    }
    #region Properties
    public ~GraphvizVertex Vertex
    {
        get
        {
            return m_Vertex;
        }
    }
    public ~VertexStringDictionary Names
    {
        get
        {
            return m_Names;
        }
    }
    #endregion
```

We can then add a handler to be attached to the ~WriterVertex event:

```
public void ~WriteVertex(Object sender, VertexEventArgs args)
{
    // label is drawed on the dot diagram
    // args.Vertex is the examined vertex
    Vertex.Label = Names[ args.Vertex ];
```

```
    // outputing to dot
    // sender is the calling algorithm
    // ToDot produces the dot code
    ((GraphvizAlgorithm)sender).Output.Write( Vertex.ToDot() );
}
```

Once the hanlder is ready, you have to attach it to the GraphvizAlgorithm gw and recall the algorithm.

```
...
gw.~WriteVertex += new VertexHandler( this.~WriteVertex );
// rerendering
gw.Write();
```

Relaunching the application, we have the expected result! Victory.

The tutorial is continued in FileDependencyPart2.

3. Boost page