**Project #2 – Package Dependency Graph Facility** due Tuesday, March 06
version 2.1 – additions to requirement #9

Purpose:

This project requires you to build a template-based facility to represent graph data structures. You will then use that facility to store package calling dependency structures. Dependencies between calls have the interesting property that there may be mutual dependency relationships between two or more calls, due to indirect recursion. We call such a mutual dependency set a strong component in the package dependency graph. Another interesting possibility is that the graph has disjoint parts.

A lot of literature has been devoted to efficient algorithms for establishing the existence, and analyzing membership, of strong components. You will be required, as part of this project, to analyze the strong component structure of any graph given in your representation. Knowledge of mutual dependencies between packages is important for designing test plans. When you make a change to a package you are obligated to test all of the packages that depend on it, and so you must test all of the elements of a strong component each time a change is made to any one of them. If there are disjoint components in a dependency graph, that implies some additional freedom in the order in which packages are tested. Can you think of a way to find disjoint components?

A smart test plan starts by testing those packages that don’t depend on any other package, then proceeds recursively to test all the packages that depend only on the already tested code. This way, when we find a defect and make a change, we don’t need to retest already tested code. Note that that plan is not possible for strong components. The best you can do is to order all the strong components[[1]](#footnote-1) in dependency order and then test bottom-up.

Requirements:

Your DEPENDENCY program:

1. **shall** use standard C++[[2]](#footnote-2) and the standard library, compile and link from the command line, using Visual Studio 2010, as provided in the ECS clusters and operate in the environment provided there[[3]](#footnote-3).
2. **shall** use services of the C++ std::iostream library for all input and output to and from the user’s console and C++ operator new and delete for all dynamic memory management.
3. **shall** use a template-based graph facility, of your design. This facility **shall** be composed of a graph class which uses the services of a vertex class, in an adjacency structure[[4]](#footnote-4). Each graph vertex has an entry in the adjacency structure. Edges are simply references that a vertex holds to other vertices. Each vertex is accompanied by an instance of a Vertex type. Each vertex **shall** hold a std::vector of std::pairs. Each pair holds a reference to a child vertex and an instance of an Edge type[[5]](#footnote-5).
4. Your graph class **shall** provide correct assignment and copy construction, and shall provide for searching the graph in several ways. This search **shall** accept a function or functor operation, to be executed on each vertex and/or edge of the graph. Your project **shall** provide a functor that, using Depth First Search (DFS) enunciates each vertex visited and each edge traversed.
5. The graph package **shall** support analysis of strong components[[6]](#footnote-6) for any representable graph, **shall** support the creation of a condensed graph (the graph of strong components), and implement a Topological Sorting algorithm on condensed graphs.
6. The graph package **shall** support finding partitions of vertices in which each partition is connected but all of the partitions are mutually disconnected. How will you do that?
7. Your graph class **shall** support containment of an instance of a parametrized type in both graph vertices and edges that join vertices, as required by 3, above. Vertices and edges will each have their own template parameter, so that different types may be so contained. That is, the graph class will be parameterized on both types, like so:

 template <typename VertexType, typename EdgeType> class graph { … };
8. **shall** provide two global algorithms that return collections of all vertices (or references to vertices) that match a specific value for the contained instance of the vertex parameterized type, and all edges (or references to edges) that match a specific value of the contained instance of the edge parameterized type.
9. The graph package **shall** provide a global template function that builds a graph instance from dependency relationships in an XML file[[7]](#footnote-7). Please demonstrate the requirements above with a graph or graphs built from a demonstration XML file or files which are distinct from the XML file you generate to satisfy requirement #10, using the format provided here: <http://www.lcs.syr.edu/faculty/fawcett/handouts/CSE687/projects/GraphXML/>.
10. **Shall** provide a facility to generate an XML file describing dependency relationships between all the files residing in a directory on a specified path. You may use #include relationships to determine those dependencies. This may be a separate executable.
11. **shall** provide a test executive package and a display package, that, combined with the graph facility, demonstrates you meet all the requirements of this specification. This should demonstrate building an XML file describing dependencies among files on a path specified on the command line, create a graph from that XML file, analyze its strong components, and place them in a topological order, then display that optimal test sequence, e.g. the name of each file in each strong component in the order determined by the sort.
12. Your project submission **shall** be uploaded in a zip file archive, including two batch files named compile.bat and run.bat that compile your project and run it using appropriate command line arguments. Please also include a Visual Studio solution that when run demonstrates you meet these requirements.

Note that requirement #10 does not ask you to provide a graphical representation of the dependency relationships. You may simply provide a text representation (please design this yourself).

You should think carefully about the output of this program. The quality of your design is measured, in part, by how well you compose the structure of your output. Note that there is no requirement to provide a graphical user interface. This tool can be implemented very effectively with a command-line input and file and console outputs.

The diagrams below illustrate the relationships between the graph structure and the adjacency and vertex structures.

1. A single package with no mutual dependencies is considered to be a strong component of size 1. Since dependency relations don’t provide a total ordering, we use topological sorting to provide a non-unique optimal ordering. [↑](#footnote-ref-1)
2. This means, for example that you may not use the .Net managed extensions to C++. [↑](#footnote-ref-2)
3. VC++ 2010 is available in all the ECS clusters. [↑](#footnote-ref-3)
4. We will discuss adjacency collections in class. [↑](#footnote-ref-4)
5. We have intentionally been somewhat vague about the nature of an edge reference. This will be discussed carefully in class. Note that the details of this requirement preclude you from using a separate edge class in this project. [↑](#footnote-ref-5)
6. <http://www.cs.cmu.edu/afs/cs/academic/class/15451-s06/www/lectures/DFS-strong-components.pdf> [↑](#footnote-ref-6)
7. We will discuss the format of the XML file in class. [↑](#footnote-ref-7)