Fall 2017 OOP345 Project

Milestone 2.

```
Milestone 2 consists of several steps:

   1. Using the CSV reader, read the task data files from the project website.
   2. Create a file Task.cpp that defines a pair of classes to hold the task data.
   3. The task data type requires a 'manager' which contains (composition) a list
      of all task instances.
   4. The task class is the instance of a task, parsing, printing, graphing, and
      other functions as required.
```

Strategy

1. Factor (move the code) the csv read and csv print into three files: cvsdump.cpp, util.cpp, and util.h
   Function main moves to csvdump.cpp, All other functions move to util.cpp and are defined in the util.h header file
   Make urtil.cpp compile without using namespace std.
   Program csvdump should behave exactly the same as csvread.cpp. It is the same program, only the code has been split between the three files.

2. Copy csvdum.cpp to task.cpp.
   Compile it and verify task runs exactly as csvread.

3. Create a class TaskManager and a class Task.
   Think about the layout of workshop 2 (TTC stations and station) and 5 (Twitter Notifications and Message).

   TaskManager manages a collection of tasks. For each line in the csv task data file, parse the fileds on the line. If a field doesn't parse, throw an excepton with a string error message. Test the parser works on all data fields. Edit the task data file so it throws an error message. Edit it again to tst different fields.

   Once the parser has been verified, write a a pair of print functions for TaskManager and Task that print the parsed task data. Test it.

   Copy the pair of print functions to a pair of 'graph' member functions.

Task Parser

For example, task data can have optionally have 1, 2, 3, or 4 data fields: 'name', 'slots', 'passed', 'failed'. Name Task data fields to be 'name', 'slots', 'pass', 'fail'.

Field 1, 3, and 4 are task names. The syntax of a task name is not defined. Let's agree a task name starts with an alpha, a digit, or a _ character, followed by alphas, digits, spaces, '-', or '_'). Write a function that checks for a valid task name.

Field 2 is an integer. (one or more digits 0-9). Write a function that checks for a valid slot.

If there is only 1 field, 'slots' defaults to "1".

Pseudo code for a simple Task parser:

```
declare 'name', 'slots', 'pass', 'fail'
switch(# of CSV fields per line) {
case 4:
  if(field 4 is a valid task name) 'fail' = field 4
  else syntax error
  // fall through to 3 field case
case 3:
  if(field 3 is a valid task name) 'pass' = field 3
  else syntax error
  // fall through to 2 field case
case 2:
  if(field 2 is a valid slot) 'slots' = field 2
  else syntax error
  // fall through to 1 field case
case 1:
  if(field 1 is a valid task name) 'name' = field 1
  else syntax error
  break; // all done parsing this data row
default:
  syntax error - not 1, 2, 3, or 4 fields
}
store 'name', 'slots', 'passed', 'failed' values into the class 'Task' data
elements
```

Sample Task Layout

For each data type, define a class with that data type and a manager class which has list data type instances.

For example:

Create class Task with the data elements you need to capture for a task.

Create class TaskManager which maintains a list of Tasks. If you use a STL vector to build TaskMangaer, there are two ways to do it:

Either with inheritance

```
class TaskManger : public vector<Task> {
public:
  some_function (...) {
    ...
    push_back( move(Task(...)) );
    ...
  }
};
```

or composition

```
class TaskManger {
  vector<Task> taskList;
public:
  some_function (...) {
    ...
    taskList . push_back( move(Task(...)) );
    ...
  }
};
```

Do both approaches support

- inheriting vector methods, for example empty() ?
- using a range-based for-loop to walk the vector list ?

You decide which style makes more sense for you. Use that style.

Task Print

Write a pair of member functions that print the data.

Task Graph

It is useful to see a picture of the task routing information.

One can stare at tabular listing, even listings with only a few line, and never see problems with the data. A picture of the data instantly reveals the structure. There are issues that went undetected with all the task data files on the website. The ‘clean’ data files are not clean.

Graphviz is open source software used to automatically generate graphs with a reasonable looking layout. The package is available from **graphviz.org.** It is very popular and it is used almost everywhere by many programs whenever an automatically layed out graph image is required.

Write a pair of member function that generates a **graphviz** graph description of the task data.

These pair of function is very similar to the pair of **print** functtions. They write the task data to a file in **graphviz** format.

Copy your working print member function to a new member function. Call it something like ‘**graph**’. You decide on the name.

Here is how:

A graph consists of two sets.

The first set is a set called 'nodes' or 'vertices'. (Vertices is plural of vertex.)

The second set is called 'edges' or 'arcs'.

Edges are pairs of nodes.

A graph where the order, or direction, of an edge from Node i to Node j is significant is a directed graph.

Directed graphs are commonly referred to as a **digraph** (directed graph).

Digraph edges have directions. For example head to tail, source to destination, start to finish, etc. Digraph edges are usually drawn as arrows.

Consider a twitter social graph (user A follows user B). A twitter graph is a digraph. Tom can follow Sally. Sally need not follow Tom.

A node in a digraph is called a source if it impossible to reach this node from other nodes.

A node in a digraph is called a sink if it impossible to reach another node from this node.

The graphviz software can be downloaded from **http://www.graphviz.org**. Please download and install the stable release, version 2.38. The new releases my work but I have not tested them.

There are run-time versions for Linux, Windows, and Mac. Download, install, and test graphviz for your platform.

I installed and tested **graphviz-2.38.msi** for windows 10. It works.

Linux knows about graphviz. Ubuntu users can run "s**udo apt-get install graphviz**" from a bash shell window. Redhat users can run "**yum install graphviz**".

Here are the available packages for Mac OSX users.

OSX Release	current stable release
mountainlion	graphviz-2.36.0.pkg
lion	graphviz-2.40.1.pkg
snowleopard	graphviz-2.38.0.pkg
leopard	graphviz-2.28.0.pkg

Graphviz consists of series of programs.

Each program reads an ASCII description of a graph (nodes and arcs), automatically generates a layout for the given graph, and writes a graph picture to file stream cout.

The graph description language is trivial. Here are some examples:

A graph starts off with the word 'graph' or 'digraph'.

Nodes can be declared by the keyword 'node', a list of names and a semicolon.

Single nodes can be specified without the 'node' keyword.

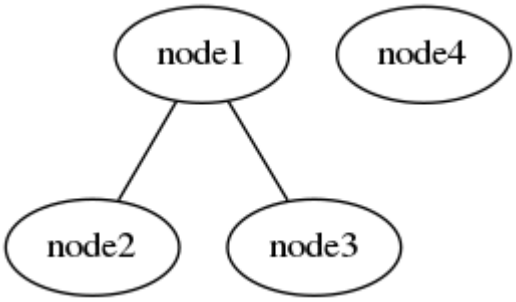
Edges are declared by a node-name , a '--' for graphs or '-> for digraphs, a node-name, and a ';'.

```
graph G {
  node1 -- node2;  # an edge from node 1 to node 2
  node1 -- node3;  # an edge from node 1 to node 3

  node4;           # an isolated node or a destination only node: no edges starting from this node.
  // node node4;   # using the optional node keyword <-- not required
}
```

The first line says it is to be graph and the graph name is 'G'. The name can be anything.

Comments can start with a '#', a C++ style '//' or C style /* ... */.

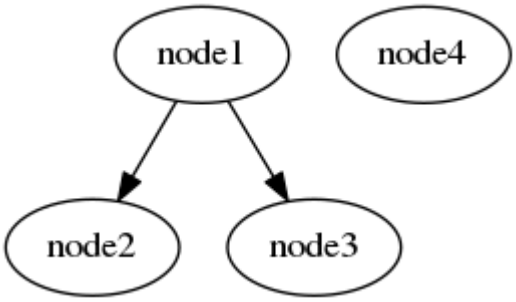


A digraph layout is similar:

```
digraph DG {
  node1 -> node2;  # a directed edge from node 1 to node 2
  node1 -> node3;  # a directed edge from node 1 to node 3

  node4;           # an isolated node or a destination only node: no edges starting from this node.
}
```

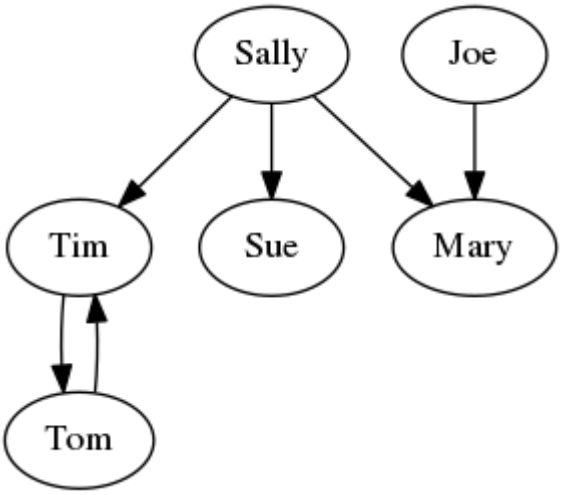
The first line says it is to be digraph and the graph name is 'DG'.



Consider a twitter graph where an edge denotes 'follows':

twitter.gv:

```
digraph twitter {
  Tim     ->    Tom;
  Tom     ->    Tim;
  Sally   ->    Mary;
  Joe     ->    Mary;
  Sally   ->    Tim;
  Sally   ->    Sue;
}
```



Graphviz can figure out node names from the edge information.

Attributes can be assigned to nodes or edges:

Here is an example of running graphviz program dot that reads a .gv file spec and produces a .png output file:

dot –Tpng twitter.gv > twitter.gv.png

See http://graphviz.org/Documentation.php for more details on how to run dot.

(Note: The graphiz package is installed on matrix. The package includes graphviz programs 'dot', 'neato', 'circo', ...)

The path to graphviz program **dot.exe** for Windows is often something like "C:\PROGRAM FILE (X86)\Graphviz2.38\bin\dot.exe".

The graphviz program '**dot**' reads a graph description file and produces an picture of the graph on file stream cout (FILE stdout, or file descriptor 1).

The command '**dot –Tpng myGraphDescription.gv > myGraphDescription.gv.png**' produces a .png picture of the graph in the .gv description.

One can execute (run) a program from your C++ program by using the system command:

NAME

system - execute a shell command

SYNOPSIS

```
#include <stdlib.h>

int system(const char *command);
```

Notice the argument to function **system** is **char***. Recall to access the **char*** data in a **std::string**, use the **c_str()** string member function.

On Unix, iOS, Android, Linux, this code suffices:

```
string cmd = "dot -Tpng myGraphDescription.gv > myGraphDescription.gv.png";
system(cmd.c_str());
```

On Windows, something like this works:

```
string cmd = "C:\\Program Files (886)\\Graphviz2.38\\bin\\dot.exe -Tpng
myGraphDescription.gv > myGraphDescription.gv.png";
system(cmd.c_str());
```

The only difference is the operating system dependent location and file suffix for program dot.

Colors and shapes can be assigned to nodes and edges. See the following example or the graphviz.org documentation for details.

A .gv graph for TaskList_Clean.dat

```
Power Supply|4|Motherboard
Motherboard|3|CPU|Remove CPU
Remove CPU|1|CPU
CPU|5|Memory|Remove CPU
Remove CPU|1|CPU
Memory|4|SSD|Remove Memory
Remove Memory|1|Memory
SSD|4|GPU|Remove SSD
Remove SSD|1|SSD
GPU|3|Test
Test|4|Approve|Repair
Approve
Repair
```

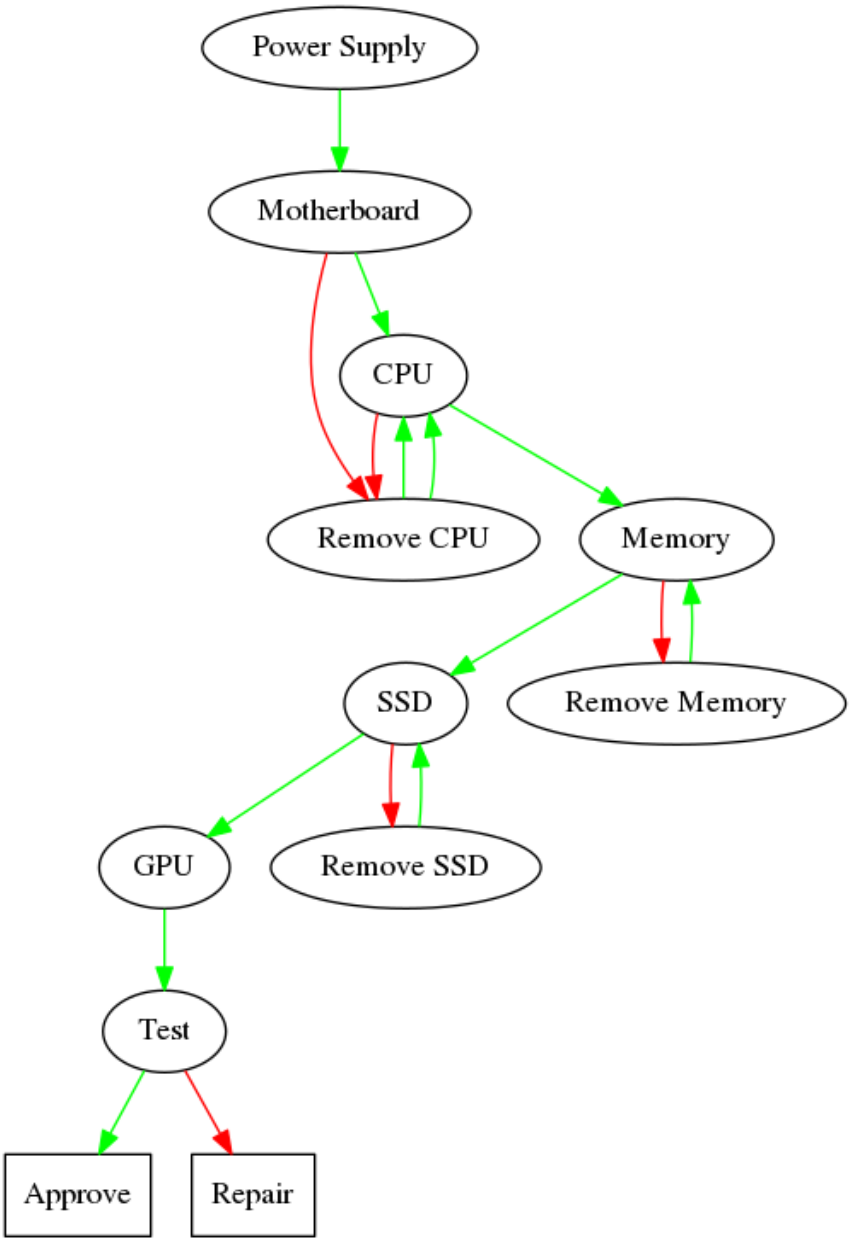
file TaskList_Clean.dat.gv looks like this:

```
digraph task {
  "Power Supply"->"Motherboard" [color=green];
  "Motherboard"->"CPU" [color=green];
  "Motherboard"->"Remove CPU" [color=red];
  "Remove CPU"->"CPU" [color=green];
  "CPU"->"Memory" [color=green];
  "CPU"->"Remove CPU" [color=red];
  "Remove CPU"->"CPU" [color=green];
  "Memory"->"SSD" [color=green];
  "Memory"->"Remove Memory" [color=red];
  "Remove Memory"->"Memory" [color=green];
  "SSD"->"GPU" [color=green];
  "SSD"->"Remove SSD" [color=red];
  "Remove SSD"->"SSD" [color=green];
  "GPU"->"Test" [color=green];
  "Test"->"Approve" [color=green];
  "Test"->"Repair" [color=red];
  "Repair" [shape=box];
  "Approve" [shape=box];
}
```

Since task node names can have embedded spaces, double quotes are required.

Note the last two lines. They are not edges. They are nodes. One can follow an edge to these nodes but there is no edge leaving these nodes.
It should be apparent the task data files are graphs.

Here is the graph:



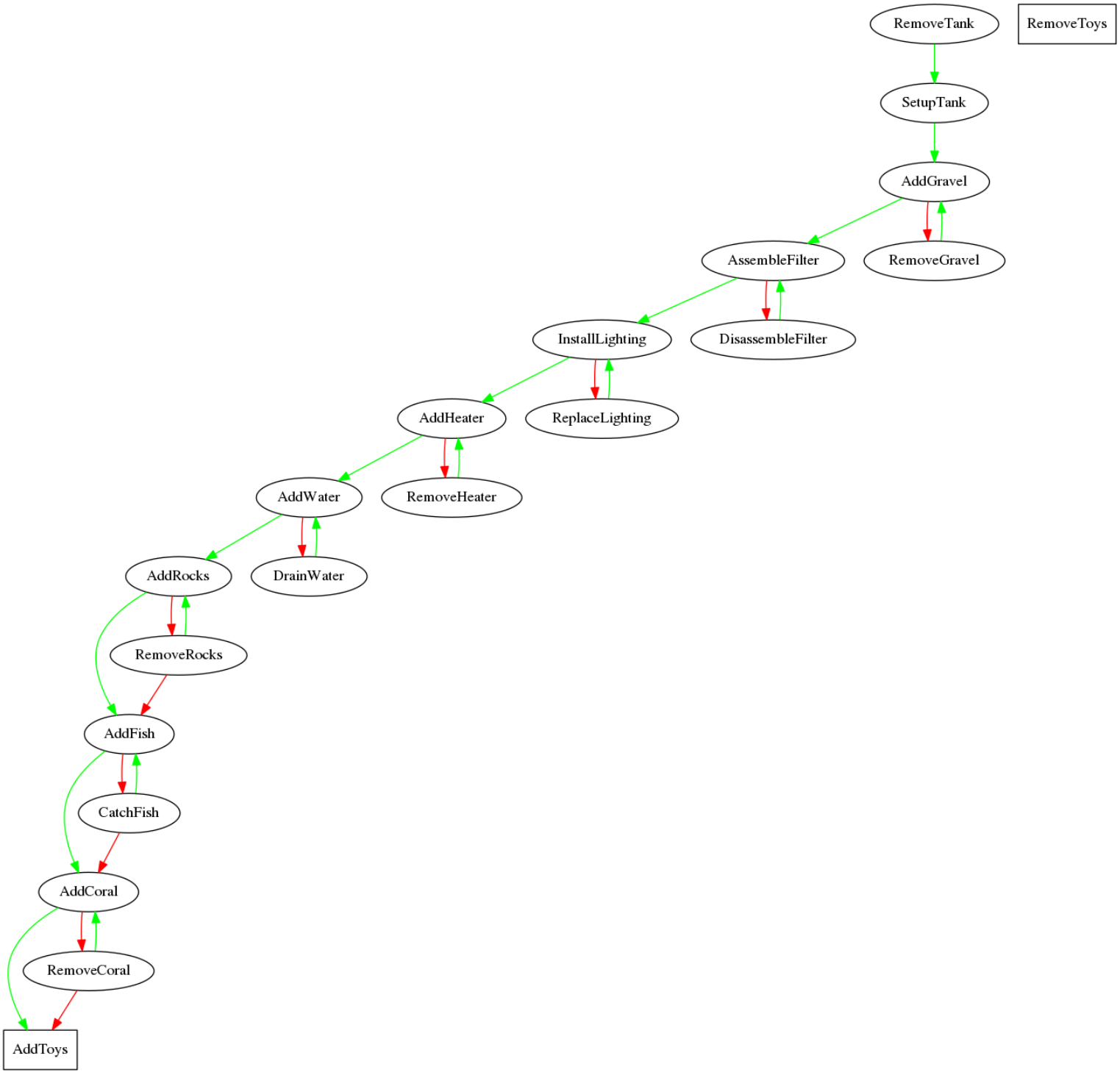
Similarly the Fishtank.dat file, .gv file, and graph are:

```
FishTankTasks.dat
      SetupTank, 2, AddGravel
      RemoveTank, 1, SetupTank
      AddGravel, 5, AssembleFilter, RemoveGravel
      RemoveGravel, 1, AddGravel
      AssembleFilter, 5, InstallLighting, DisassembleFilter
      DisassembleFilter, 1, AssembleFilter
      InstallLighting, 2, AddHeater, ReplaceLighting
      ReplaceLighting, 1, InstallLighting
      AddHeater, 5, AddWater, RemoveHeater
      RemoveHeater, 1, AddHeater
      AddWater, 5, AddRocks, DrainWater
      DrainWater, 1, AddWater
```

```
      AddRocks, 8, AddFish, RemoveRocks
      RemoveRocks, 1, AddRocks, AddFish
      AddFish, 10, AddCoral, CatchFish
      CatchFish, 1, AddFish, AddCoral
      AddCoral, 5, AddToys, RemoveCoral
      RemoveCoral, 1, AddCoral, AddToys
      AddToys, 4
      RemoveToys, 1

FishTankTasks.dat.gv
      digraph task {
        "SetupTank"->"AddGravel" [color=green];
        "RemoveTank"->"SetupTank" [color=green];
        "AddGravel"->"AssembleFilter" [color=green];
        "AddGravel"->"RemoveGravel" [color=red];
        "RemoveGravel"->"AddGravel" [color=green];
        "AssembleFilter"->"InstallLighting" [color=green];
        "AssembleFilter"->"DisassembleFilter" [color=red];
        "DisassembleFilter"->"AssembleFilter" [color=green];
        "InstallLighting"->"AddHeater" [color=green];
        "InstallLighting"->"ReplaceLighting" [color=red];
        "ReplaceLighting"->"InstallLighting" [color=green];
        "AddHeater"->"AddWater" [color=green];
        "AddHeater"->"RemoveHeater" [color=red];
        "RemoveHeater"->"AddHeater" [color=green];
        "AddWater"->"AddRocks" [color=green];
        "AddWater"->"DrainWater" [color=red];
        "DrainWater"->"AddWater" [color=green];
        "AddRocks"->"AddFish" [color=green];
        "AddRocks"->"RemoveRocks" [color=red];
        "RemoveRocks"->"AddRocks" [color=green];
        "RemoveRocks"->"AddFish" [color=red];
        "AddFish"->"AddCoral" [color=green];
        "AddFish"->"CatchFish" [color=red];
        "CatchFish"->"AddFish" [color=green];
        "CatchFish"->"AddCoral" [color=red];
        "AddCoral"->"AddToys" [color=green];
        "AddCoral"->"RemoveCoral" [color=red];
        "RemoveCoral"->"AddCoral" [color=green];
        "RemoveCoral"->"AddToys" [color=red];
        "AddToys" [shape=box];
        "RemoveToys" [shape=box];
```

}


RemoveTank
RemoveToys
SetupTank
AddGravel
AssembleFilter
RemoveGravel
InstallLighting
DisassembleFilter
AddHeater
ReplaceLighting
AddWater
RemoveHeater
AddRocks
DrainWater
RemoveRocks
AddFish
CatchFish
AddCoral
RemoveCoral
AddToys

Simplify Testing

You are welcome to generate meaningful data files with referential integrity for testing.

For example:

SimpleTask.data:

Install Power Supply|4|Install Motherboard|Remove Power Supply

```
Remove Power Supply|2|Install Power Supply
Install Motherboard|3|Install CPU|Remove Motherboard
Remove Motherboard|1|Install Motherboard
Install CPU|5|Install Memory|Remove CPU
Remove CPU|1|Install CPU
Install Memory|4|SSD|Remove Memory
Remove Memory|1|Install Memory
Install SSD|4|Install GPU|Remove SSD
Remove SSD|1|Install SSD
Install GPU|3|Test
Remove GPU|1|Install GPU
Test|4|Approve|Repair
Approve
Repair
```

A Special Note for Visual Studio Users

MSVS IDE WOES

The default Visual Studio project set up is for building a single .exe program file. It is possible to configure a Visual Studio project that builds multiple .exe programs. A quick search of msdn.microsoft.com does not readily show how to do this. Companies such as AMD don't bother coercing VS to support multiple .exe targets. They have hundreds of source files. Everything changes breaking the build procedure whenever they upgrade the compiler. They use another build system, scons (scons.org), to manage running the Microsoft Visual Studio compiler where multiple .exe targets are required. You can experiment with scons, but there is a far easier solution that meets our needs.

MSVS IDE WOES SOLUTION – USE THE COMMAND LINE COMPILER

It is trivial to build multiple programs running the compiler from the command line. Run the compiler for each program you need to build. The Visual Studio compiler is a "compile and link" program called **CL.EXE**. Visual Studio uses **CL.EXE** to compile code. You can also call it from a **CL.EXE** command line window. A few things need to be set up before it will work.

The program needs to know the location of the include files (for example <vector> or <iostream>) and the location of the library files which contain the compiled code for the c++ library functions.

There is a file in the **C:\Program Files (x86)\Microsoft Studio xx.x\VC** folder tree called something like **vcvars.bat, vcvars32.bat, vcvarsx86.bat** or some similar name. The batch file name changes from release-to-release or patch-to-patch. Run this batch file from a **CMD.EXE** window. It will set up the environment so **CL.EXE** knows the location of the include and library files required to compile and link your program.

I installed the 2016-Nov-23 version of the free MSVC IDE system on an up-to-date Windows 10 machine. What an ordeal. It ran all night installing 22 Gbytes of bloat. Further more, once installed, it refused to create a new project, citing a setup error. The installation process ran error free. What was I thinking?

So the 2016-Nov-23 version of free MSVC IDE project system is broken. The IDE is not important to us. We want to bypass it and call the compiler directly from the

command line to build multiple targets, one target at a time.

Where is **VCVARSxxx.BAT**? Running UNIX find on the '**C:/Program Files (x86)**' folder we see:

```
C:/Program Files (x86)/Microsoft Visual Studio 14.0/Common7/Tools/VCVarsPhoneQueryRegistry.bat
C:/Program Files (x86)/Microsoft Visual Studio 14.0/Common7/Tools/vcvarsqueryregistry.bat
C:/Program Files (x86)/Microsoft Visual Studio 14.0/VC/bin/amd64/vcvars64.bat
C:/Program Files (x86)/Microsoft Visual Studio 14.0/VC/bin/amd64_arm/vcvarsamd64_arm.bat
C:/Program Files (x86)/Microsoft Visual Studio 14.0/VC/bin/amd64_x86/vcvarsamd64_x86.bat
C:/Program Files (x86)/Microsoft Visual Studio 14.0/VC/bin/vcvars32.bat
C:/Program Files (x86)/Microsoft Visual Studio 14.0/VC/bin/vcvarsphoneall.bat
C:/Program Files (x86)/Microsoft Visual Studio 14.0/VC/bin/vcvarsphonex86.bat
C:/Program Files (x86)/Microsoft Visual Studio 14.0/VC/bin/x86_amd64/vcvarsx86_amd64.bat
C:/Program Files (x86)/Microsoft Visual Studio 14.0/VC/bin/x86_arm/vcvarsphonex86_arm.bat
C:/Program Files (x86)/Microsoft Visual Studio 14.0/VC/bin/x86_arm/vcvarsx86_arm.bat
C:/Program Files (x86)/Microsoft Visual Studio 14.0/VC/vcvarsall.bat
```

File '`C:/Program Files (x86)/Microsoft Visual Studio 14.0/VC/vcvarsall.bat`' looks promising.

To compile your code, this worked on my Windows machine:

> Start up a CMD.EXE window.
>
> Suppose it starts up in drive D, folder \. Visual Studio 14.0 is my latest Microsoft Visual Studio install.

```
D:\> C:
C:\> CD \Program Files (x86)\Visual Studio 14.0\VC
C:\Program Files (x86)\Visual Studio 14.0\VC> vcvarsall.bat
C:\Program Files (x86)\Visual Studio 14.0\VC> CD \Users\<your-name>\<your 345 project files>
C:\Users\<your-name>\<your 345 project files>> CL csvreader.cpp
C:\Users\<your-name>\<your 345 project files>> CL csvdump.cpp util.cpp
C:\Users\<your-name>\<your 345 project files>> CL task.cpp util.cpp
C:\Users\<your-name>\<your 345 project files>> CL item.cpp util.cpp
C:\Users\<your-name>\<your 345 project files>> CL order.cpp util.cpp
C:\Users\<your-name>\<your 345 project files>> CL t-test.cpp t.cpp util.cpp
C:\Users\<your-name>\<your 345 project files>> CL i-test.cpp i.cpp util.cpp
C:\Users\<your-name>\<your 345 project files>> CL o-test.cpp o.cpp util.cpp
C:\Users\<your-name>\<your 345 project files>> CL iot.cpp i.cpp o.cpp t.cpp util.cpp
C:\Users\<your-name>\<your 345 project files>> ...
```

One more thing. If using try-throw-catch, you need to add a compiler flag. The compiler will tell you the flag to use if it detects try-throw-catch code while compiling without the flag. Add the flag. It works.

Running Windows dot.exe from C++.

First install graphviz for windows from graphviz.org, http://graphviz.org/Download.php. I used the .MSI installer. It created the graphviz family of programs in folder **C:/Program Files (x86)/Graphviz2.38/**. Program **dot.exe** is in sub-folder '**bin**'.

This O/S agnostic Windows/UNIX code runs dot on each of its command line arguments.

```
// File dot-runner.cpp
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char*argv[])
{
```

```
        #ifdef __unix
          string dot = "dot";
        #else
          string dot = "C:/\"Program Files (x86)\"/Graphviz2.38/bin/dot.exe";
        #endif

        for(int arg = 1; arg < argc; arg++) {

          string file(argv[arg]);

          cmd = dot + " -Tpng " + file + " > " + file + ".png";

          cout << "Running command -->'" << cmd << "'\n";

          cout << "The operating system says dot returned '"
               << system(cmd.c_str())
               << "' (0 is good --> dot executed successfully)\n";
        }
      }
```

I tested this program on Windows 10 and Ubuntu 16.04. It works.

So there you go. The magic line for '**dot.exe**' is
"C:/\"Program Files (x86)\"/Graphviz2.38/bin/dot.exe".