ECE244 Lab 3: A Resistor Network Program

1 Objectives

The objective of this assignment is to provide you with a practical introduction to programming with C++ objects. This assignment requires you to complete the implementation of two classes and their associated member functions, as well as performing basic I/O operations with formatting.

2 Problem Statement

Your task is to implement a program for storing resistors in a circuit. The circuit, or network, is defined by a number of *nodes*, each node being connected to one or more *resistors*. Each resistor has certain values associated with it, including: resistance value, text name, and the IDs of the two nodes (endpoints) it connects. The diagram below (Fig 1) illustrates an example. Here R1 connects nodes 1 and 2, R2 does the same, and R3 connects nodes 1 and 3. The nodes could be strips on a breadboard for example, with resistors plugged in to some nodes.

The program you will write is similar to the "input-and-store-the-network" portion of real programs used to simulate electric circuits, and programs that control the robots that automatically insert resistors connecting the appropriate points (nodes) on circuit boards.

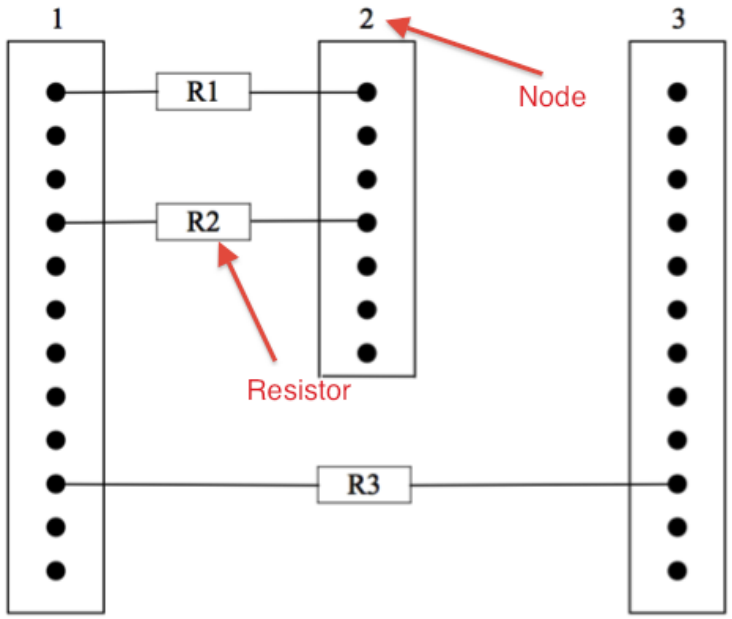

Figure 1: Example circuit

Circuits are input using a text-based user interface, which will require you to extend the command parser you developed in lab2. Commands are defined to allow the user to insert, modify, delete and print resistors, and to set the maximum size of the network. Based on the commands given, you will need to maintain two arrays, one of Resistors and one of Nodes, to which you can add and modify elements.

3 Preparation & Background

The key concepts in this lab are the creation and use of classes. You should review the textbook and lecture notes pertaining to the following topics:

- Dynamic memory allocation and arrays
- Constructors and destructors
- Class member variables and functions
- Public/private access types

4 Specifications

The commands your program must accept are those specified in Table 2 of lab 2, along with one new command to set the maximum size of the network. However, while in lab2 you had only to check that the commands were syntactically valid, in this lab you will actually store the resistors and their connectivity in two arrays (of Resistors and of Nodes) and output the correct data in response to the `print` commands.

The first line of input to your program will set the maximum allowed node number and the maximum number of resistors:

`maxVal` *maxNodeNumber maxResistors*

where both *maxNodeNumber* and *maxResistors* are positive integers. You should store these two values in variables in your program and also use them to specify the appropriate size for the resistor and node arrays your program will require. Use the `new[]` operator to allocate the node and resistor arrays. If your program is sent several `maxVal` commands during one run, you will have to ensure that for any `maxVal` command after the first one you `delete` any already allocated Node and Resistor arrays, and then allocate the arrays to the specified new sizes. The node and resistor arrays and any other necessary variables should be initialized to the empty network state (no resistors stored) after any `maxVal` command.

The *resistor index* shall be the position of the resistor in the resistor array, starting at 0. The program shall be able to accommodate exactly *maxResistors*, as specified in the `maxVal` command. The nodes to which these resistors can be attached are identified by an integer ID, which corresponds to their index in the `node` array. You must be able to store data for node indices from 0 to *maxNodeNumber*, inclusive, where *maxNodeNumber* is set by the `maxVal` command. Each node can accomodate only a limited number of resistors attaching to it (think of the number of holes in a breadboard strip); this is a `#define` constant`MAX_RESISTORS_PER_NODE` which is set to 5 for this lab. The node and resistor arrays shall be sized according to the `maxVal` command, start empty and be modified based on user commands as defined below in Sec 4.2. The command parser must check its input for validity, catching and reporting the errors defined in Sec 4.3. Lastly, successful commands must update the data structures and produce output conforming to Sec 4.2. Additional restrictions on how the program is to be written and organized are detailed below.

4.1 Coding Requirements

1. You shall create classes `Node` and `Resistor` to store your network, starting with the provided defintions.

2. The provided header files `Node.h` and `Resistor.h` shall contain only the class definitions for `Node` and `Resistor` respectively. They may be modified to add additional member data and functions. The access type (public/private), arguments, and return types of the existing functions shall not be changed.

3. You shall create source files `Node.cpp` and `Resistor.cpp` to contain the function definitions for the two classes. Non-class-member functions (parsing, printing, main, etc.) should be stored in separate files.

4. Any class member variables that are added shall be of private access type. You may provide public accessor functions to set and get values where necessary.

5. All I/O shall be from `cin`/`cout` using the operators `<<` and `>>`. You may use `peek()`, `getline()`, and `stringstream`s as in the previous lab.

6. You should reuse your parser from lab2. If you did not detect all invalid input in lab 2 all is not lost – the grading for this lab will not check many parsing errors from lab 2. The only errors you need to detect are those specified in Sec 4.3.

4.1.1 Class `Resistor`

Each object of the `Resistor` class holds information about one resistor in the network. You must create a file `Resistor.cpp` containing implementations for the functions indicated in `Resistor.h` below (Fig 2). Each resistor object has a string (`name`), a resistance value in Ohms (`resistance`), and the IDs (indices in the node array) of the two nodes to which it is connected (`endpointNodeIDs[2]`). There are also constructor and destructor functions to initialize the class, a function to print the values, and functions to set/get the data members of the class. You may add data members and methods, but must adhere to the restrictions outlined in Sec 4.1 above.

The program shall store all resistors in an array, starting at element 0 for the first resistor added and incrementing from there. When adding a `Resistor` to a `Node`, the resistor shall be referred to by its position in the resistor array, also known as its `resistorID`.

4.1.2 Class `Node`

The starter code for class `Node` is shown below. The variable `numRes` is intended to store the number of resistors currently attached to the node. It should start at zero, and increment each time a resistor is added.

4.2 Input and Output

Input shall be accepted as specified in lab 2, except you do not have to handle the *deleteR name* command (but you do have to handle *deleteR all*). The meaning of the commands shall be as indicated in Table 1.

For each valid line of input (ie. each line not causing an error as defined in Section 4.3), one or more lines of output shall be produced as described in Table 1 and below. The values in italics must be replaced by either the value given in the command, or the value already stored (eg. *resistance_old* in the `modifyR` output defined in Table 1). Strings must be reproduced exactly as entered. An example session is provided in Sec 6 to illustrate this.

```
/*
 * File:   Resistor.h
 * Author: JC and VB
 *
 * Created on October 6, 2013, 1:01 PM
 */

#ifndef RESISTOR_H
#define RESISTOR_H

#include <string>
#include <iostream>
using namespace std;

class Resistor
{
private:
   double resistance; // resistance (in Ohms)
   string name; // C++ string holding the label
   int endpointNodeIDs[2]; // IDs of nodes it attaches to

public:
   Resistor(int rIndex_,string name_,double resistance_,int endpoints_[2]);
   // rIndex_ is the index of this resistor in the resistor array
   // endpoints_ holds the node indices to which this resistor connects

   ~Resistor();

   string getName() const; // returns the name
   double getResistance() const; // returns the resistance

   void setResistance (double resistance_);

   // you *may* create either of the below to print your resistor
   void print ();
   friend ostream& operator<<(ostream&,const Resistor&);
};

ostream& operator<<(ostream&,const Resistor&);

#endif        /* RESISTOR_H */

```

Figure 2: Starter defintion for class `Resistor` (you may add to this, but not modify what is given)

4.2.1 Printing resistance information (`printR` command)

Resistor information shall be printed as:

name resistance Ohms *nodeid* -> *nodeid*

The two nodes shall appear in the order in which they were presented in the `insertR` command that created the resistor. The name field shall be **20 characters wide, aligned on the left side and padded with spaces**. The resistance field shall be eight characters wide, show two digits after the decimal place, and be aligned on the right side[1]. An example is shown below. The underscore characters are intended to show padding spaces that are part of the column widths

[1] See the `setw` and `left` IO manipulators

```
/*
 * File:   Node.h
 * Author: JC and VB
 *
 * Created on October 6, 2013, 12:58 PM
 */

#ifndef NODE_H
#define NODE_H

#define MAX_RESISTORS_PER_NODE 5


class Node
{
private:

   int numRes; // number of resistors currently connected
   int resIDArray[MAX_RESISTORS_PER_NODE]; // stores the index of each resistor connected

public:
   Node();
   ~Node();

   // Updates resIDArray to show the resistor in position rIndex in
   // the resistor array is now connected to this node.
   // Returns true if successful
   bool addResistor (int rIndex);

   // prints the whole node
   // nodeIndex is the position of this node in the node array.
   void print (int nodeIndex);

};

#endif        /* NODE_H */

```

Figure 3: Starter defintion for class `Node` (you may add to this, but not modify what is given)

defined above. All columns shall be separated by a single space.

```
R20_________________ __150.00 Ohms 1 -> 3
```

When using the `printR` command, the output above is preceded by a single line of output, `Print:`

```
Print:
R20_________________ __150.00 Ohms 1 -> 3
```

For `printR all` the output is the same, except there are multiple resistances printed, one per line. Resistors should be printed in the same order in which they were entered.

4.2.2 Output from `printNode` command

The output from the `printNode` command is shown below, assuming that `printNode 1` was entered.

Command	Arguments	Output if valid	Action if valid
maxVal	maxNodeNumber maxResistors	New network: max node number is *maxNodeNumber*; max resistors is *maxResistors*	Node array created with new[] to store nodes from 0 to *maxNodeNumber* and Resistor array created to store up to *maxResistors*; the network is initialized to empty (no resistors).
insertR	name resistance nodeid nodeid	Inserted: resistor *name resistance* Ohms *nodeid* -> *nodeid*	Adds 1 resistor to Resistor array and updates 2 entries in Node array
modifyR	name resistance	Modified: resistor *name* from *resistance_old* Ohms to *resistance* Ohms	Updates one entry in Resistor array
printR	name	Print: *resistor info (see below)*	No data changed
printR	`all`	Print: *resistor info (see below)*	No data changed
printNode	nodeid	Print: *node info (see below)*	No data changed
printNode	`all`	Print: *node info (see below)*	No data changed
deleteR	`all`	Deleted: all resistors	All resistors cleared and Node array updated so we have an empty network

Table 1: Valid commands and arguments

```
Print:
Connections at node 1: 2 resistor(s)
  R20_________________ __150.00 Ohms 1 -> 3
  Rfb_________________ ___20.13 Ohms 1 -> 4
```

For each node, a header line shall be printed indicating the number of attached resistors. Following that line, all connected resistors shall be printed in the order in which they were added, formatted as in Sec 4.2.1, but with each resistor line indented by two spaces. If no resistors are attached, no resistor lines are printed. The header line shall be printed as (note there are no double-spaces anywhere despite appearances):

`Connections at node` ***`nodeid`***`:  ***count*** `resistor(s)`

The output from `printNode all` is the same, except data for each node is printed, from nodeid (index) 0 to *maxNodeNumber* inclusive. Note that each resistor will appear twice, once per node to which it is connected. For example:

```
Print:
Connections at node 0: 0 resistor(s)
Connections at node 1: 2 resistor(s)
  R20_________________ __150.00 Ohms 1 -> 3
  Rfb_________________ ___20.13 Ohms 1 -> 4
Connections at node 2: 0 resistor(s)
Connections at node 3: 1 resistor(s)
```

```
  R20_________________ __150.00 Ohms 1 -> 3
Connections at node 4: 1 resistor(s)
  Rfb_________________ ___20.13 Ohms 1 -> 4
```

4.3 Error Checking

The program must check that the input is valid. Your program must check for the error conditions listed in the table below and output the specified message. It is fine if your program checks for the other error conditions listed in lab 2, but those errors will not be tested in this lab, as they were already graded in lab 2. If a command has any error (e.g. does not parse correctly, or the `insertR` command fails due to inadequate space in either the resistor or node array), the network (arrays and related variables) shall remain unchanged. As in lab 2, at most one error message shall be printed per line of input. In the messages, italicized values such as R should be replaced by the value causing the error.

Error message output must comply exactly (content, case, and spacing) with the table below to receive credit. There are no trailing spaces following the text.

Error message	Cause
Error: maxVal arguments must be greater than 0	*maxNodeNumber* or *maxResistors* was specified to be 0 or negative
Error: resistor *name* not found	When searching for a resistor by name (eg. in modifyR, printR), a resistor with the given name was not found
Error: node *value* is out of permitted range *lower_bound-upper_bound*	An integer nodeid value has been provided that is outside the legal 0 to *maxNodeNumber* range
Error: resistor array is full	There are no more spaces in the resistor array to accommodate the resistor being added
Error: node is full	One of the endpoint nodes is already full (has MAX_RESISTORS_PER_NODE resistors already attached)

Table 2: Errors to be reported in this lab.

The program is not required to deal with errors other than those listed in the table above. If more than one error message is applicable for a single line of input, the error message listed first in Table 2 should be printed (i.e. the errors in Table 2 are listed in descending priority order).

5 Helpful Hints

- You may (and are encouraged to) recycle the parser from the previous lab. You will need to make a few minor changes and add the maxVal command.
- Before submitting, remember to use `exercise` to test your program. Don't forget that `exercise` covers only some possible input; when graded your program will be tested with

some of the test cases in `exercise` and some additional test cases. Accordingly, you should read the specification above carefully and create some additional test cases of your own.

- You will need to store one array of `Node`s and one of `Resistor`s - global scope is acceptable for a program of this scale. You will also need a variable such as `resCount` to count the number of resistors already added, and will need variables to store `maxNodeNumber` and `maxResistors`.
- For output formatting (padding, left/right justify, precision control) you should look at the `<iomanip>` header file[2]. Functions like `left`, `right`, `setfill`, and `setw` will be of great help.
- Add features one-by-one. Even if you cannot complete every command (or detect every error), you should still submit your program since it may pass some test cases.
- Your program should `delete` all the memory it allocates with `new` before it exits; this is good practice and ensures there are no memory leaks. The autotester will check if your program deletes all the memory it should and you will lose marks if you do not. A good way to check if you have deleted all the memory you allocated with new is to run the `valgrind` memory checking program. A tutorial on `valgrind` will be posted to the portal, and you are encouraged to learn and use this tool.

6 Example Session

```
> maxVal 0 1
Error: maxVal arguments must be greater than 0
> maxVal 6 7
New network: max node number is 6; max resistors is 7
> insertR R0 100.1 0 1
Inserted: resistor R0 100.10 Ohms 0 -> 1
> insertR R1 110.001 0 2
Inserted: resistor R1 110.00 Ohms 0 -> 2
> insertR R2 120 0 3
Inserted: resistor R2 120.00 Ohms 0 -> 3
> insertR R3 130 0 4
Inserted: resistor R3 130.00 Ohms 0 -> 4
> insertR R4 140 0 5
Inserted: resistor R4 140.00 Ohms 0 -> 5
> insertR R5 150 0 6
Error: node is full
> insertR R6 160 1 7
Error: node 7 is out of permitted range 0-6
> insertR R7 170 1 5
Inserted: resistor R7 170.00 Ohms 1 -> 5
> insertR R8 180 4 3
Inserted: resistor R8 180.00 Ohms 4 -> 3
> insertR R9 190 2 4
Error: resistor array is full
> printR R8
Print:
R8                    180.00 Ohms 4 -> 3
> printR R9
Error: resistor R9 not found
> printR all
Print:
```

[2] www.cplusplus.com/reference/iostream is a good reference

```
R0                    100.10 Ohms 0 -> 1
R1                    110.00 Ohms 0 -> 2
R2                    120.00 Ohms 0 -> 3
R3                    130.00 Ohms 0 -> 4
R4                    140.00 Ohms 0 -> 5
R7                    170.00 Ohms 1 -> 5
R8                    180.00 Ohms 4 -> 3
> printNode 1
Print:
Connections at node 1: 2 resistor(s)
  R0                    100.10 Ohms 0 -> 1
  R7                    170.00 Ohms 1 -> 5
> printNode all
Print:
Connections at node 0: 5 resistor(s)
  R0                    100.10 Ohms 0 -> 1
  R1                    110.00 Ohms 0 -> 2
  R2                    120.00 Ohms 0 -> 3
  R3                    130.00 Ohms 0 -> 4
  R4                    140.00 Ohms 0 -> 5
Connections at node 1: 2 resistor(s)
  R0                    100.10 Ohms 0 -> 1
  R7                    170.00 Ohms 1 -> 5
Connections at node 2: 1 resistor(s)
  R1                    110.00 Ohms 0 -> 2
Connections at node 3: 2 resistor(s)
  R2                    120.00 Ohms 0 -> 3
  R8                    180.00 Ohms 4 -> 3
Connections at node 4: 2 resistor(s)
  R3                    130.00 Ohms 0 -> 4
  R8                    180.00 Ohms 4 -> 3
Connections at node 5: 2 resistor(s)
  R4                    140.00 Ohms 0 -> 5
  R7                    170.00 Ohms 1 -> 5
Connections at node 6: 0 resistor(s)
> modifyR R3 2.5
Modified: resistor R3 from 130.00 Ohms to 2.50 Ohms
> printR R3
Print:
R3                    2.50 Ohms 0 -> 4
> maxVal 3 4
New network: max node number is 3; max resistors is 4
> printNode all
Print:
Connections at node 0: 0 resistor(s)
Connections at node 1: 0 resistor(s)
Connections at node 2: 0 resistor(s)
Connections at node 3: 0 resistor(s)
> insertR bigres 1000 2 3
Inserted: resistor bigres 1000.00 Ohms 2 -> 3
> insertR smallres 0.05 2 4
Error: node 4 is out of permitted range 0-3
> printR all
Print:
bigres                1000.00 Ohms 2 -> 3
> deleteR all
Deleted: all resistors
> printR all
Print:
```

```
>
```

7 Procedure

Create a sub-directory in your `ece244` directory, and set its permissions so no one else can read it. Create a Makefile or NetBeans project to build a program called `rnet` starting with the provided files. Write and test the program to conform to the specifications laid out in Sec 4.

The hints in Sec 5 may help get you started, and the example session in Sec 6 may be used for testing.

8 Deliverables

You must submit all source files to permit your project to compile. They should be:

- Main.cpp
- Rparser.cpp (a modified version of Parser.cpp from previous lab)
- Rparser.h (containing function prototypes for the parser)
- Resistor.h
- Resistor.cpp
- Node.h
- Node.cpp

As before, it is essential that your program produce output exactly as specified: no additional or missing parts.

Submit the files using `~ece244i/public/submit 3`.