Lab 3: The Resistor Network Program

Objectives

The objectives of this assignment are for you to practice: (1) the use of C++ I/O streams, including error handling, (2) dynamic allocation and de-allocation of one-dimensional arrays, (3) solve a problem using object-oriented programming. You will do so through the design of a program that parses circuit building commands from the standard input, displaying appropriate error messages if necessary, and by creating and maintaining resistors in a circuit. You will be asked to calculate the voltage at certain nodes.

Problem Statement

Your task is to implement a program for storing resistors in a circuit. The circuit, or network, is definec 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.

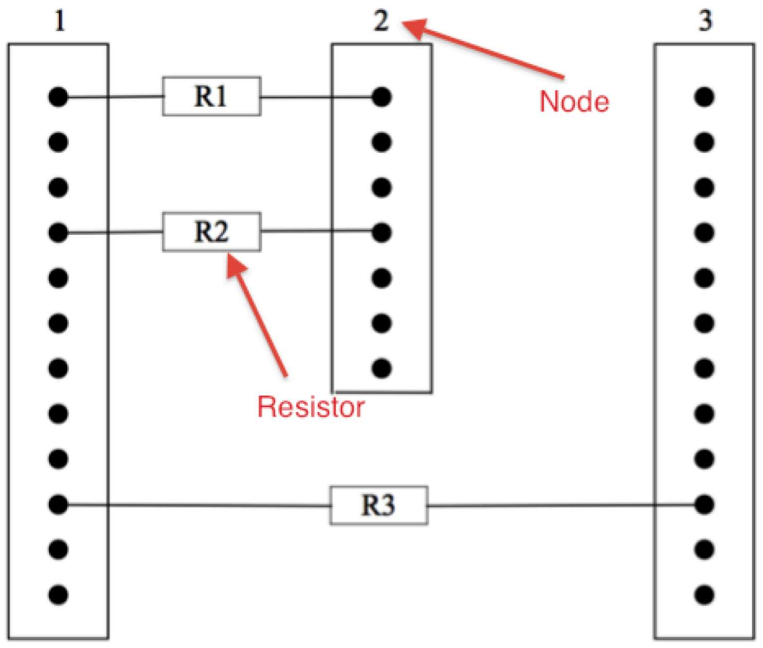

Fig. 1: Example of a circuit

Circuits are input using a text-based user interface. 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.

Specifications

It is important that you follow the specifications below carefully. Where the specification says **shall** or **must** (or their negatives), following the instruction is **required** to receive credit for the assignment. If instead it says *may* or *can*, these are optional suggestions. The use of *should* indicates a recommendation; compliance is not specifically required. However, some of the recommendations may hint at a known-good way to do something or pertain to good programming style.
Example input and output for the program are provided at the end of this document for your convenience. They do not cover all parts of the specification. You are responsible for making sure your program meets the specification by reading and applying the description below.

Coding Requirements

1. The code you will write shall be contained in only three source files named `main.cpp`, `Resistor.cpp` and `Node.cpp`. Skeletons of the files are released with the assignment's zip file. The zip file also contains two header files: `Resistor.h`, `Node.h`. These files are **NOT** to be modified in any way. Modifying these files often results in a mark of 0 for the assignment. However, you may make use of helper functions to split up the code for readability and to make it easier to re-use. These functions (and their prototypes) **must** be in one of the aforementioned source files. That is, you must not add any new header or source files.
2. Input and output must be done **only** using the C++ standard library streams `cin` and `cout`.
3. The stream input operator `>>` and associated functions such as `fail()` and `eof()` shall be used for all input. C-style IO such as `printf` and `scanf` shall not be used.
4. Strings shall be stored using the C++ library type `string`, and operations shall be done using its class members, not C-style strings.
5. C-library string-to-integer conversions (including but not limited to `atoi`, `strtol`, etc) shall not be used.

The commands your program must accept are those specified in Table 2. In addition to checking the commands are syntactically valid, in this lab you will 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 - 1`, inclusive, where `maxNodeNumber` is set by the `maxVal` command. Nodes entered by the user range from 1 to `maxNodeNumber`, but are transferred to indices by subtracting 1. Each node can accommodate 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. The command parser must check its input for validity, catching and reporting the errors. Lastly, successful commands must update the data structures and produce output as discussed below. Additional restrictions on how the program is to be written and organized are detailed below.

Command Line Input

Input will be given one command on one line at a time. The entire command must appear on one line. All input must be read using the C++ standard input `cin`.
The program shall indicate that it is ready to receive user input by prompting with three greater-than sign followed by a single space (`>>>`); see end of this page for an example.
Input shall always be accepted one line at a time, with each line terminated by a newline character (A newline character is input by pressing `Enter`. If there is an error encountered when parsing a line, the program shall print an error message (see ~~Section~~~??? Table 3), the line shall be discarded, and processing shall resume at the next line. The program shall continue to accept and process input until an end-of-file (`eof`) condition is received `eof` is automatically provided when input is redirected from a file. It can also be entered at the keyboard by pressing Ctrl-D. Each line of valid input shall start with a command name, followed by zero or more arguments, each separated by one or more space characters. The number and type of arguments accepted depend on the command. The arguments and their permissible types/ranges are shown below in Table 1.

Argument	**Description, type, and range**
name	A string consisting of any non-whitespace characters. Whitespace characters are tab, space, newline, and related characters which insert "white space"; they mark the boundaries between values read in by operator<<, except the string `all` which we reserve for special use characters.
nodeid	Node ID, an integer ranging from 1 to `maxNodeNumber`
resistance	Positive resistance value (type `double`)

Table 1: Acceptable input arguments

The valid commands, their arguments, and their output if the command and its arguments are all legal are shown below in Table 2. The program shall verify that the command and arguments are correctly formatted and within range, and that a command is followed by the correct number of arguments, on the same line. The handling of command names shall be case-sensitive. If there is an error, a message shall be displayed as described in Table 3. Otherwise, a successful command produces a single line of output on the C++ standard output, `cout`, as shown in Table 2. The values in italics in Table 2 must be replaced with the values given by the command argument. Strings must be reproduced exactly as entered. Where *nodeids* are printed, they shall appear on the order entered in the command. Resistance values shall be printed in fixed point format with exactly two decimal places, regardless of the number entered.

The valid commands, their arguments, and their output if the command and its arguments are all legal are shown below in Table 2.

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). Print function is implemented in Resistor.cpp already.	No data changed
deleteR	all	Deleted: all resistors	All resistors cleared and Node array updated so we have an empty network
setV	nodeid voltage	Set: node *nodeid* to *voltage* Volts	Updates the specified node voltage data member. This corresponds to connecting this node to a voltage source.
solve		Solve: node voltage info (see below) for bonus	Runs a numerical solution technique to determine the

		marks.	voltage of all nodes that do not have their voltages set for bonus marks.

Table 2: Summary of Commands and Actions

Printing resistance information

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 resistance field shall show two digits after the decimal place (use `setprecision` function in `iomanip` library). `printR` command requires you to call the `print` function implemented in Resistor.cpp.

Error Checking

Error message	Cause
invalid command	The first word entered does not match one of the valid commands in Table 2
too few arguments	Fewer arguments were given than expected for a command
invalid argument	The argument is not of the correct type. For example, a floating point number may have been entered instead of an integer nodeid or a string other than `all` may have been entered where a nodeid or `all` is expected.
negative resistance	The resistance value is strictly less than zero (*0.0* is permitted)
resistor name cannot be the keyword "all"	A resistor name of `all` was specified to insertR or modifyR
resistor name already exists	A resistor name of same name exists at insertR
resistor *name* not found	The resistor name entered at modifyR or printR is not found
node value is out of permitted range 1 - *maxNodeNumber*	An integer nodeid value has been provided that is out of range
both terminals of resistor connect to same node	The two nodes to which a resistor connects cannot be the same

Table 3: List of errors to be reported, in priority order

The program must check that the input is valid. It must be able to identify and notify the user of the following input errors~~, in order of priority~~. Where multiple errors exist on one input line, only one should be reported: the one that occurs first as the line is read from left to right. Errors shall cause a message to be printed to `cout`, consisting of the text "Error:" followed by a single space and the erro message from Table 3. 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.
The program is not required to deal with errors other than those listed in Table 3.

The `Resistor` Class

Each object of the `Resistor` class holds information about one resistor in the network. In Resistor.cpp, you must implement all the functions indicated in Resistor.h. 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 is ~~are~~ also a constructor ~~and destructor~~ function to initialize the class, a function to print the values, and functions to set/get the data members of the class. 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. Be mindful of what is an index and what is the node number. The Resistor.h file has multiple comments to help you implement the member functions.

The `Node` Class

The starter code for class `Node` is in Node.h and Node.cpp. 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. The Node.h file has multiple comments to help you implement the member functions in Node.cpp.

Bonus: The `solve` command

For 1% of the course grade bonus, the solve command first determines the voltage of every node, and then it prints the voltage of every node in ascending node order. To find the voltage at every node in a network, we can follow the iterative procedure below. The iterative procedure we are using to solve for the node voltages is called the Gauss-Seidel method. It is a very general method of solving linear systems of equations, and is used to solve circuits and other systems of equations with thousands or even millions of unknowns.

```
Initialize the voltage of all nodes without a specified (setV) voltage to 0.

while (some node’s voltage has changed by more than MIN_ITERATION_CHANGE) {

    for (all nodes without a set voltage) {

        set voltage of node according to Eq. 3

    }
}
```

The voltage of a node is computed from the voltages of its neighbours according to Kirchoff's current equation, which states that the total current entering or leaving a node must be 0. Consider node_0 below, which is surrounded by 3 resistors to 3 other nodes, as shown in Fig. 2. The total current entering node_0 must be:

$$I_a + I_b + I_c = 0$$

We can rewrite this using the fact that the current through each resistor is simply the voltage across i divided by its resistance.

$$\frac{V_1 - V_0}{R_a} + \frac{V_2 - V_0}{R_b} + \frac{V_3 - V_0}{R_c} = 0$$

Rearranging and solving for the voltage at node_0, V_0, yields:

$$V_0 = \frac{1}{\frac{1}{R_a} + \frac{1}{R_b} + \frac{1}{R_c}} \times \left(\frac{V_1}{R_a} + \frac{V_2}{R_b} + \frac{V_3}{R_c} \right)$$

You can safely assume that at two nodes will have their voltage set before typing the `solve` command.

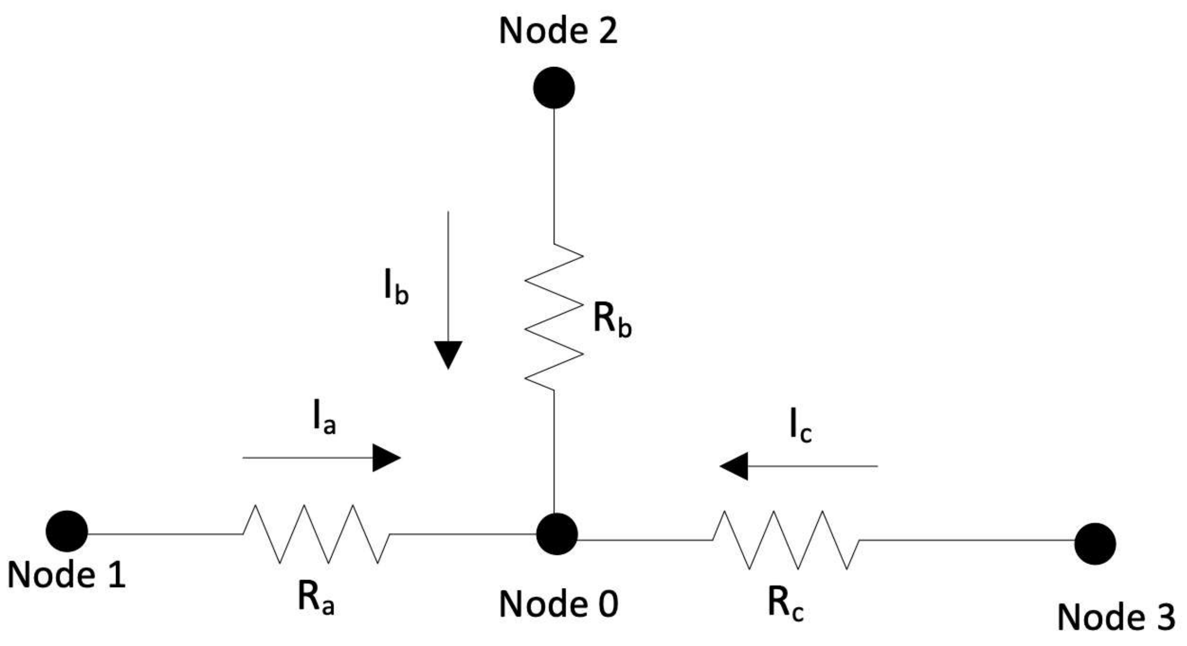

Fig 2: A portion of a resistor network. Node 0 is connected to three other nodes as shown.

The iterative procedure above determines what voltage a node must have for it to balance the curren coming through all the resistors connected to it. Once we update the voltage of some node (say node #0), however, the voltage of other nodes (e.g., node #1) connected to that node may have to change to ensure the current flowing into them is in balance. Hence we iterate through all the nodes, repeatedly updating the voltages of those that do not have a fixed (`setV`) voltage according to Eq. 3 until no voltage changes much --- at that point we have *converged* to a solution.

For this lab, you should iterate until no node changes by more than `MIN_ITERATION_CHANGE`, which you should define to be 0.0001.

Once your solver has converged, print (to `cout`) the voltage of every node that has at least one resistor connected to it, in ascending node order. For the example shown in Fig. 3, the solve command would print:

```
Solve:
Node 2: 0.30 V
Node 3: -0.02 V
Node 4: -0.60 V
Node 5: 0.52 V
Node 6: 1.00 V
```

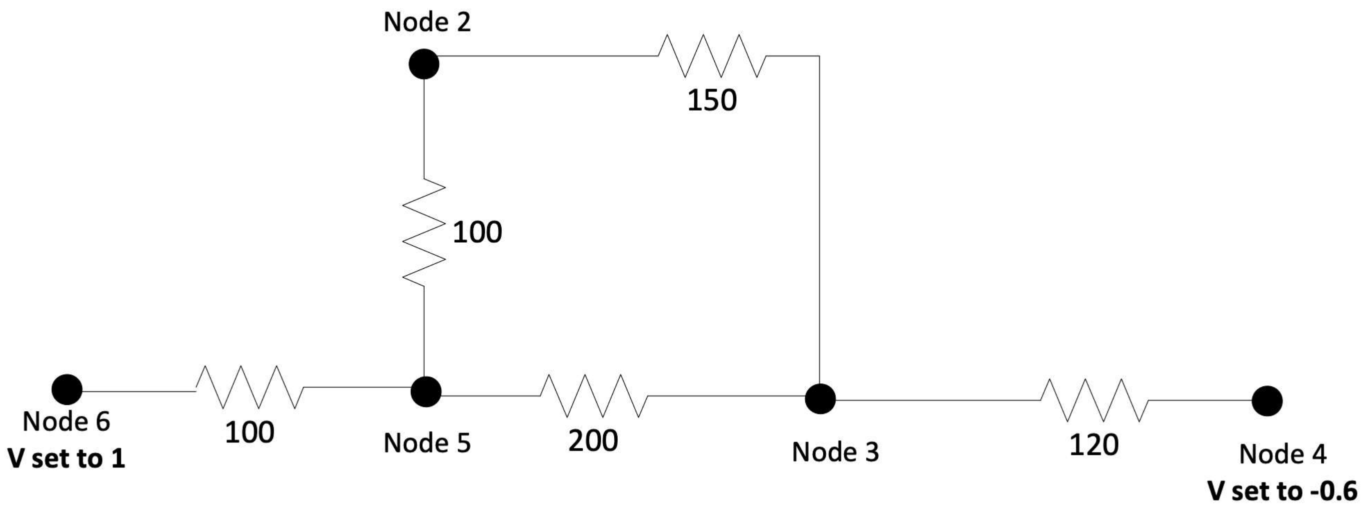

Fig 3: A more complex resistor network. Two nodes have been connected to voltage sources with setV, and our program will solve for the other 3 voltages.

Hints

1. You can check a stream for end-of-file status using the `eof` member function.
2. To save typing, you can create one or more test files and pipe them to your program. You can create a text file using a text editor (try `gedit`, `gvim`, or the VSCode editor). If your file is called `test`, you can then send it to your program by typing `circuit < test`. Building a good suite of test cases is important when developing software.
3. If you want to look ahead ("peek") at what character would be read next without actually reading it, `peek()` does that. For instance, if you type "Hello" then each time you run `peek()` you will get 'H'. If you read a single character, it will return 'H' but then subsequent calls to peek() will return 'e'.
4. When interacting with your program from the keyboard, Ctrl-D will send an end-of-file (eof) marker.
5. Reading from `cin` removes leading whitespace. When reading strings, it discards all whitespace characters up to the first non-whitespace character, then returns all non-whitespace characters until it finds another whitespace. For integers (numbers), it skips whitespace and reads to the first non-digit (0-9) character.

6. Remember you can use the debugger to pause the program, step through it, and view variables (including strings).
7. If you decide to pass the string stream you created to a function, remember that string streams (and other types of streams for that matter) can only be passed by reference, not by value.
8. IO manipulators (see header file `<iomanip>`) can be used to control the appearance of output. You may need at least `setprecision`.
9. You will need to store one array of `Nodes` and one of `Resistors` - global scope is acceptable for a program of this scale. You will also need a variable such as `resistorsCount` to count the number o resistors already added, and will need variables to store `maxNodeNumber` and `maxResistors`.
10. For output formatting (padding, left/right justify, precision control) you should look at the <iomanip> header file (**www.cplusplus.com/reference/iostream (http://www.cplusplus.com/reference/iostream)** is a good reference). Functions like left, right, setfill and setw will be of great help.
11. 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.
12. 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 Quercus, and you are encouraged to learn and use this tool.

A suggested (but not mandatory) structure for your code appears in the skeleton main.cpp file released within the assignment's zip file.

Procedure

Create a sub-directory called lab3 in your ece244-labs directory, and set its permissions so no one else can read it. Download the lab3_release.zip file, un-zip it and place the resulting files in the lab3 directory. There are three source files in which you will add your code. The first is main.cpp in which you will write the command parser code. The second and third files are Node.cpp and Resistor.cpp ir which you will implement the classes `Node` and `Resistor` respectively. All files are in the directory *lab3_release*. You must not rename these files or add more files.
The release also contains two include files Node.h and Resistor.h. **You may NOT modify these files** to add to or delete from their content. Modifying the files commonly results in a mark of zero for the assignment. In addition, there is a Makefile to separately compile your project. Run `make` to compile your code. Do not modify this file either.
The exercise command will also be helpful in testing your program. You should exercise the executable, i.e., `circuit`, using the command:

```
~ece244i/public/exercise 3 circuit
```

As with previous assignments, some of the `exercise` test cases will be used by the `autotester` during marking of your assignment. We will not provide all the autotester test cases in exercise, however, sc you should create additional test cases yourself and ensure you fully meet the specification listed above.
Input to public test cases can be found in files named `1`, `2`, `3` and `4` in `tests.zip` file. Test case 1 2, 3 don't test the solve command, but test case 4 does and counts for partial marks for the bonus in this lab.

Reference Executable

To run the reference/solution executable, you may run the following command:

`~ece244i/public/circuit-ref`

Deliverables

Submit the main.cpp, Node.cpp and Resistor.cpp files as lab 3 using the command
`~ece244i/public/submit 3`

Example Output

```
>>> maxVal 1 1
New network: max node number is 1; max resistors is 1
>>> maxVal 6 7
New network: max node number is 6; max resistors is 7
>>> insertR R1 100 6 5
Inserted: resistor R1 100.00 Ohms 6 -> 5
>>> insertR R2 100 5 2
Inserted: resistor R2 100.00 Ohms 5 -> 2
>>> insertR R3 200 5 3
Inserted: resistor R3 200.00 Ohms 5 -> 3
>>> insertR R4 150 2 3
Inserted: resistor R4 150.00 Ohms 2 -> 3
>>> insertR R4 140 2 3
Error: resistor R4 already exists
>>> insertR R5 120 3 3
Error: both terminals of resistor connect to same node
>>> setV 6 1
Set: node 6 to 1.00 Volts
>>> setV 4 -0.6
Set: node 4 to -0.60 Volts
>>> printR R0
Error: resistor R0 not found
>>> print R1
Error :invalid command
>>> printR R1
Print:
R1                    100.00 Ohms 6 -> 5
>>> insertR all 2
Error: resistor name cannot be keyword "all"
>>> deleteR all
Deleted: all resistors
>>>
```