# ECE244 Lab 2

## 1 Objectives

The objective of this assignment is to provide you with practice on the use of C++ I/O facilities and string operations. It also provides an introduction to parsing of terminal input and text files, an important skill for almost all programs. You will modify and use this command parser to exercise several of the subsequent assignments.

## 2 Problem Statement

You will write a command parser that provides a textual input interface to your program, a circuit database program which contains a number of resistors. We do not implement the database here, just the front-end that accepts commands. The parser should take a sequence of commands as input. Each command consists of an operation followed by its arguments. The command and the arguments are separated by one or more spaces. The program should take input from the terminal or a file, parse it, verify that it is correct, and print a response or error message. It will loop, processing input according to the specification laid out below as long as input is available.

## 3 Preparation & Background

Parsing of input from the user and files is critical for most programs. Since content of user input is variable and may or may not be correct, correct program operation requires checking of user input to ensure it conforms to the expected format ("makes sense"). If it does not, an error message should be issued so the program doesn't "bomb" or run and produce nonsense results. This lab starts with the most basic C++ constructs for reading from a *stream* which is an object that represents a flow of characters in or out of the program. Streams are a useful abstraction for things like files and the terminal,[1] and provide a number of useful functions for extracting values from the underlying source. To start, please read the following background material so you start with some understanding of streams and parsing:

- Textbook Chapter 6
- Lecture notes on IO and parsing
- The use of `cin` and `cout` with the IO operators `<<` and `>>`
- Documentation on the `iostream` and `stringstream` classes[2]

Also read the spec below (Sec 4) carefully to be sure you understand what you are supposed to do.

[1] They also include other things like hardware devices, network connections, pipes between programs, etc - anything that can produce a sequence of characters

[2] www.cplusplus.com has a good reference

# 4 Specifications

To receive credit for this lab, you must follow the specifications below carefully. Where the spec says *shall* or *must* (or their negatives), following the instruction is **required** to receive credit. Any such requirement may be tested by the marking script or by examining the code. If instead is 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. Your code will be marked subjectively for style, so it's best to take the recommendations unless you have a good reason not to.

Example input and output for the program are provided in Sec 6 for your convenience. They do not cover all parts of the specification. You are responsible for making sure your program meets the spec by reading and applying the information below.

## 4.1 Coding Requirements

1. The entire program shall be contained in a single source file named `Parser.cpp`. It should make use of functions to split up the code for readability, and to make it easier to re-use parts of the code in the future.

2. Input and output must be done using the C++ standard library streams `cin` and `cout`.

3. The stream input operator `>>` and associated functions such as getline 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.

6. Maximum values shall be stored as `#define` constants, including `MAX_NODE_NUMBER`. For this lab, the `MAX_NODE_NUMBER` is 5000.

## 4.2 Input

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 a greater-than sign followed by a single space (`> `) - see Sec 6 for an example. Input shall always be accepted one line at a time, with each line terminated by a newline character[3]. If there is an error encountered when parsing a line, the program shall print an error message (see Sec 4.3), 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.[4]

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.

[3] A newline character is input by pressing Enter

[4] EOF is automatically provided when input is redirected from a file or pipe. It can also be entered at the keyboard by pressing Ctrl-D.

| Argument | Description, type, and range |
|---|---|
| name | A string consisting of any non-whitespace characters, except the string *all* which we reserve for special use[5] characters |
| nodeid | Node ID, an integer ranging from 0 to `MAX_NODE_NUMBER` |
| 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. Notice that the last three commands can be run in two ways (depending on their argument): with a specific node or resistor, or with the keyword `all`. 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 Sec 4.3. 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 *nodeid*s 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.

| Command | Arguments | Output if Command is Valid |
|---|---|---|
| insertR | name resistance nodeid nodeid | Inserted: resistor *name resistance* Ohms *nodeid* -> *nodeid* |
| modifyR | name resistance | Modified: resistor *name* to *resistance* Ohms |
| printR | name | Print: resistor *name* |
| printR | `all` | Print: all resistors |
| printNode | nodeid | Print: node *nodeid* |
| printNode | `all` | Print: all nodes |
| deleteR | name | Deleted: resistor *name* |
| deleteR | `all` | Deleted: all resistors |

Table 2: Valid commands and arguments and their output

### 4.3 Error Checking

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. If more than one error could be reported for a single argument in the line, only the error occuring first in Table reftbl:errors should be reported.

Errors shall cause a message to be printed to `cout`, consisting of the text "`Error:` " followed by a single space and the error message from the table below. In the messages, italicized values such as *value* 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.

The program is not required to deal with errors other than those listed in Table 3.

| Error message | Cause |
|---|---|
| invalid command | The first word entered does not match one of the valid commands |
| 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) |
| node *value* is out of permitted range *lower_bound-upper_bound* | An integer nodeid value has been provided that is out of range |
| resistor name cannot be the keyword "all" | A resistor name of `all` was specified to insertR or modifyR |
| both terminals of resistor connect to node *value* | The two nodes to which a resistor connects cannot be the same |
| too many arguments | More arguments were given than expected for a command |
| too few arguments | Fewer arguments were given than expected for a command |

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

# 5 Helpful Hints

- You can check a stream for end-of-file status using the `eof` member function.
- The `ignore` member function in `iostream` may be useful to you if you need to ignore the remainder of a line.
- 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 NetBeans editor). If your file is called `test.txt`, you can then send it to your program by typing `Parser < test.txt`. Building a good suite of test cases is important when developing software.
- 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 subseequent calls to `peek()` will return 'e'.
- When interacting with your program from the keyboard, Ctrl-D will send an End-Of-File (EOF) marker.
- 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.
- Remember you can use the debugger to pause the program, step through it, and view variables (including strings).
- IO manipulators (see header file `<iomanip>`) can be used to control the appearance of output. You may need at least `setprec`.

- Double-quotes can be printed to the terminal either as single characters (`cout << '"'`) or by *escaping* them with a slash inside a string (`cout << "text \"in quotes \""`).

A suggested (but not mandatory) structure for your code would be:

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


int parser() {

    string line, command;

    // May have some setup code here

    getline (cin, line);  // Get a line from standard input

    while ( !cin.eof () ) {

        // Put the line in a stringstream for parsing
        // Making a new stringstream for each line so flags etc. are in a known state

        stringstream lineStream (line);
        lineStream >> command;

        if (command == "insertR") {
            // parse an insertR command
        }
        else if ( ... ) {
            ...
        }

        getline (cin, line);
    }  // End input loop until EOF.

    return 0;
}
```

# 6 Examples

## 6.1 Getting started

The program when first started, ready to receive input:

```
>
```

Now the user types a command (ending with Enter) to add a new resistor between nodes 1 and 5, with label of R2 and a resistance of 47.0 Ohms.

```
> insertR R2 47.0 1 5
```

To which the program should respond with the message for a successful addition:

```
> insertR R2 47.0 1 5
Inserted: resistor R2 47.00 Ohms 1 -> 5
```

### 6.2 Full session

The following is an example session. Note that the text from the prompt (`> `) up to the end of the line is typed by the user, whereas the prompt and line without a prompt are program output.

```
> insertR R0 100 0 1
Inserted: resistor R0 100.00 Ohms 0 -> 1
> insertR smallres 1.033 3 5
Inserted: resistor smallres 1.03 Ohms 3 -> 5
> insertR newres 3 2 2
Error: both terminals of resistor connect to node 2
> modifyR R0 12
Modified: resistor R0 to 12.00 Ohms
> modifyR R0 -1.5
Error: negative resistance
> deleteR R0
Deleted: resistor R0
> deleteR all
Deleted: all resistors
> printNode 12
Print: node 12
> printNode 8000
Error: node 8000 is out of permitted range 0-5000
> printR alpha
Print: resistor alpha
> printR
Error: too few arguments
> printR alpha beta
Error: too many arguments
> Help me!!
Error: invalid command
> modifyR all 33
Error: resistor name cannot be the keyword "all"
>
```

## 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 `Parser` from a single source file

`Parser.cpp`. 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 sessions in Sec 6 may be used for testing. The `~ece244i/public/exercise` command will also be helpful in testing your program, and 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, so you should create additional test cases yourself and ensure you fully meet the specification listed above.

## 8 Deliverables

Submit the `Parser.cpp` file as lab 2 using the command

```
~ece244i/public/submit 2
```

The programming style (structure, descriptive variable names, useful comments, consistent indentation, use of functions to avoid repeated code and general readability) shown in your code will be examined and graded as well.