Lab 3 : A Command Line Parser

1 Objectives

The objectives of this assignment are for you to practice: (1) the use of C++ I/O streams, including error handling, (2) writing a simple class with constructors, accessors and mutators, and (3) dynamic allocation and de-allocation of one-dimensional arrays. You will do so through the design of a program that parses drawing commands from the standard input, displaying appropriate error messages if necessary, and by creating and maintaining objects that represent drawn shapes.

2 Problem Statement

The assignment consists of two main parts. In the first part, you will write a command parser that provides a textual input interface to your program. The parser takes a sequence of commands as input. The commands create, delete modify and display *shapes* to be drawn on the screen. Each command consists of an operation keyword followed by arguments. The command and the arguments are separated by one or more spaces. Thus, the code you will write for this part of the assignment should take input from the standard input, parse it, verify that it is correct, print a response or error message and either create, delete or modify `Shape` objects that represent the shapes specified in the command. The command parser loops, processing input as long as input is available.

In the second part of the assignment, you will implement a simple "database" of objects that stores the created shapes. To do so, you will implement a class called `Shape` that is used to store the shape properties. Further, you will create and maintain a dynamically allocated array of pointers to `Shape` objects to keep track of `Shape` objects created and deleted.

In this assignment, you will not actually draw the shapes to the screen, just process the commands and maintain the database of the `Shape` objects.

3 String Streams

In the lectures, you were introduced to the standard input stream (handled by `cin`) as well as user-created file streams (handled by `ifstream` objects that you create). There is one more useful type of user-defined streams, namely *string streams*. These streams come handy when processing input one line at a time, as you will do in this assignment. The rest of this section introduces you to string streams and how to use them. You will find that they are not that much different than using `cin` or `ifstream`.

String streams allows the extraction of input from a string, as opposed to from the keyboard (`cin`) or a file (`ifstream`)[1]. The following example demonstrates how this may be done.

[1] String streams can be also used as output streams, but in this section, only their use for input is described.

```
1  #include <iostream>
2  using namespace std;
3  #include <string>
4  #include <sstream>
5
6  int main () {
7    int anInteger;
8    string inputLine = "204 113 ten";
9
10   stringstream sin (inputLine);
11
12   sin >> anInteger; // Extracts 204
13   if (sin.fail()) return (-1);
14   sin >> anInteger; // Extracts 113
15   if (sin.fail()) return (-1);
16   sin >> anInteger; // Extraction fails
17   if (sin.fail()) return (-1);
18
19   return (0);
20 }
```

The `#include <sstream>` on line 4 imports the definition of string streams, allowing it to be used in the example. The `main` function creates a `string` variable called `inputLine` on line 8 and initializes it to `"204 113 ten"`. The declaration on line 10 creates a new string stream handler called `sin` (ala `cin`, but you can give it any other name). This stream is initialized from the `inputLine` string variables created and initialized earlier[2].

Once this is done, we can use the `sin` handler in the same way we use `cin`. We can extract an integer from the stream, as shown on line 12. The handler `sin` has the same set of flags that `cin` has. Thus, we can check if the extraction operation failed by invoking the method `sin.fail()`. In the example, the extraction succeeds and the value 204 is placed in `anInteger`. The same happens for the second extraction on line 14, which extracts 113. In contrast, the third extraction on line 16 fails, the value of `anInteger` is not affected and `main` returns with exit code -1.

The above example is not very interesting because it extracts input from a string initialized by the program and has the foreknowledge that three integers are expected. More interesting is when we wish to extract input from a string provided by the user and we have no knowledge of how many extractions we have to do. The example below illustrates how to do this.

In the example, the function `getline()` is used to read the input the user provides through the keyboard and places the entire stream received by `cin` into the string `inputLine`. This includes all the white spaces in the stream (see your lecture notes for details). It also appends an `eof` to the stream. Thus, while a `cin` stream may end with an `Enter` or an end-of-file (`eof`) a string stream always ends with an `eof`. In the example, we assume the user entered `204 113 10` as input. Thus, `inputLine` contains `"204 113 10"`.

The string `inputLine` is then used to build the string stream handler called `sin` on line 11. Subsequently, the `while` loop iterates until there is no more integers in a line. `sin` is used to extract

[2] A copy of the string variable is made inside the string stream. Thus, if `inputLine` changes after the string stream is created, the stream in `sin` does not change.

an integer from the input (the string), as shown on line 16. The first extraction succeeds and thus the `fail` and `eof` flags of `sin` are false. The integer (204 in our example) is printed to the standard output and the `moreInput` flag remains true. The next two iterations of the `while` loop extract the next two integers, 113 and 10, and prints them to the output.

On the next iteration of the `while` loop, the extraction fails because the `eof` is encountered. Both the `fail` and the `eof` flags are set to true. The code checks for these flags as shown on lines 17 and 18. Since the `eof` flag is true, the `moreInputs` flag is set to false (line 19) causing the `while` loop to exit and the program to terminate. Since the program immediately exits after checking the `eof` flag, there is really no need to clear the flags of `sin`, as the comment indicates in the code on line 20. Thus, it is safe to remove this line from the code.

```
1  #include <iostream>
2  using namespace std;
3  #include <string>
4  #include <sstream>
5
6  int main () {
7    int anInteger;
8    string inputLine;
9
10   getline(cin, inputLine);
11   stringstream sin(inputLine);
12
13   bool moreInput=true;
14   while (moreInput) {
15
16       sin >> anInteger;
17       if (sin.fail()) {
18          if (sin.eof()) {
19             moreInput = false;
20             sin.clear();            // Not necessary
21          }
22          else {
23             cout << "Bad input\n";
24             sin.clear();            // Not necessary
25             sin.ignore(10000,'\n'); // Again not necessary
26          }
27       }
28       else  cout << "The integer read is: " << anInteger << endl;
29   }
30
31   return (0);
32 }
```

Now, let's assume that the user provides `" 204 113 ten"` as input. The first two extraction succeed as above. However, the third extraction fails because of the `ten`. The fail flag is set to true but the `eof` flag is set to false. The code then checks if the reason of failure is the `eof` (line 18), and this is not the case. A message is printed to instruct the user that the input is bad. When using

`cin` the flags **must** be cleared (using `cin.clear()`) and the input stream **must** be flushed using `cin.ignore()`. However, with string streams, this is unnecessary since we can simply discard the input by reading another line from `cin` using `getline()` and then rebuilding the stream handler using the new input (not shown in the example). This automatically resets the flags and flushes the old input, replacing it by the new one. Thus, the two calls on lines 24 and 25 are not really necessary and can also be removed from the code.

The use of string streams is helpful when input must be processed one line at a time and the user is not allowed to break input across multiple lines of input, separated by `Enter`s. String streams allows your program to get the entire line of input, analyze it and decide if the line is valid or not. While this can be done using `cin`, it is more difficult since `cin` allows user input to be split into multiple lines. Indeed, this is the case for this assignment and the skeleton of the `main` program (included with the lab release) shows a modified version of the above example.

4 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. 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 Section 6 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.

4.1 Coding Requirements

1. The code you will write shall be contained in only two source files named `parser.cpp` and `Shape.cpp`. Skeletons of the two files are released with the assignment's `zip` file. The `zip` file also contains two `.h` files: `globals.h` and Shape.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 two `.cpp` files. That is, you must not add any new `.h` or `.cpp` 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.

Argument	Description, type, and range
name	a string consisting of any non-whitespace characters[3]; except strings that represent commands, shape types or the reserved word `all`.
type	a string that represents the type of a shape and must be one of: `ellipse`, `circle`, `rectangle` or `triangle`
loc	a positive integer (0 or larger) that represents the location of the shape in either the x or y dimension
size	a positive integer (0 or larger) that represents the size of the a shape in either the x or y dimension
value	a positive integer (0 or larger) that represents the maximum number of shapes in the database
angle	a positive integer between 0 and 360 that represents angle of rotation of a shape

Table 1: Acceptable input arguments

4.2 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 a greater-than sign followed by a single space (`> `); see Section 6 for an example. Input shall always be accepted one line at a time, with each line terminated by a newline character[4]. If there is an error encountered when parsing a line, the program shall print an error message (see Section 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[5].

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.

Command	Arguments	Output if Command is Valid
maxShapes	value	New database: max shapes is *value*
create	name type loc loc size size	Created *name*: *type loc loc size size*
move	name loc loc	Moved *name* to *loc loc*
rotate	name angle	Rotated *name* by *angle* degrees
draw	name	Drew *name*: *type loc loc size size*
draw	`all`	Drew all shapes
delete	name	Deleted shape *name*
delete	`all`	Deleted: all shapes

Table 2: Valid commands and arguments and their output

The valid commands, their arguments, and their output if the command and its arguments are

[3] Whitespace characters are tab, space, newline, and related characters which insert "white space"; they mark the boundaries between values read in by `operator<<`. All other characters (digits, letters, underscore, symbols, etc.) are non-whitespace characters.

[4] A newline character is input by pressing `Enter`.

[5] `eof` is automatically provided when input is redirected from a file. It can also be entered at the keyboard by pressing Ctrl-D.

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 or a string may have been entered instead of an integer where an integer is expected.
invalid shape name	The name used for a shape is a reserved word (e.g., a command name or a shape type)
shape name exists	A shape with the name *name* exists in the database, i.e., has once been created and has not been deleted
shape name not found	A shape with the name *name* specified in a command does not exist
invalid shape type	The type used for a shape is not one of the allowed types
invalid value	The value specified in a command is invalid. For example, a less than 0 value for a loc argument, a rotation angle not between 0 and 360, or the two size arguments of the `circle` shape are not equal.
too many arguments	More arguments were given than expected for a command
too few arguments	Fewer arguments were given than expected for a command
shape array is full	An attempt to create more shapes than the argument given to the maxShapes command

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

all legal are shown below in Table 2. Notice that the last two commands (`draw` and `delete`) can be run in two ways (depending on their argument): with a specific shape name, or with the keyword `all`. In the case of the `draw all` command, the program prints not only the message shown in the table, but also all the shapes in the database (see the example in Section 6).

Also notice that for the `circle` shape, the two size arguments must be equal, or an error message is printed, as described in Section 4.3.

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. The handling of command names shall be case-sensitive.

The first line of input to your program will always be the `maxShapes` command. It creates a new (empty) database of shapes with the specified maximum allowed number of shapes. You shall assume that this command will not have any errors in it. The `maxShapes` command may be given multiple times. A subsequent `maxShapes` command shall create a new database with the new maximum allowed number of shapes (see Section 4.5 for more details).

If there is an error, a message shall be displayed as described in Section 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 *loc*s or *size*s are printed, they shall appear on the order entered in the command.

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 3 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 Table 3. In the messages, italicized values such as *name* 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. The following are some clarification on the errors.

1. The commands and the shape types are case sensitive. Thus, while a shape cannot be named `all`, `draw` or `triangle`, it can be named `All`, `Draw` or `triAngle`.

2. For every line of input, your program must output something. Either a message indicating success (Table 2) or an error (Table 3). So for an empty line of input (nothing or just whitespace) your program should print `Error:  invalid command`.

3. Only the first error from the left should be reported per line of input. In the case of missing/extra arguments, these are errors in the arguments that are missing/extra and should be reported only if the preceding arguments were valid.

4. You should let the extraction operator (`>>`) do the work for you. Recall that the operator stops when the next character cannot be converted to the destination type. This may or may not be a white space. Learn how to use the `cin.peek()` method explained in Section 5 below.

4.4 The `shape` Class

The `shape` class holds the properties of a shape, including its name, type, location, size and rotation. The definition of the class appears in `Shape.h`, which is re-produced in Figure 1. Examine the file and read through the comments to understand the variables and methods of the class. You must implement this class in the file `Shape.cpp`.

4.5 The Database

The program shall keep track of all shapes using an array whose elements are pointers to `Shape` objects. The array should be dynamically allocated after the first line of the input to have a size that matches exactly the argument given to the `maxShapes` command. The array elements shall all be initialized to `NULL`. This array is declared in `parser.cpp`. An integer variable `shapeCount` is used to track the actual number of `Shape` objects stored in the database. Figure 2 depicts what the database may look like during program execution.

All shapes shall be stored in the array (i.e., by having the pointer element of the array point to a `Shape` object) starting at element 0 for the first shape added and incrementing from there. When a Shape object is deleted, the memory allocated to the object must be freed and the element of the array that used to point to the Shape object must be assigned the value `NULL` When a new Shape object is added after another one is deleted, it must be added at location `shapeCount`. Thus, you must not "pack" the array after deletions or reuse "deleted" locations.

When the `maxShapes` command is issued after a database has been created with an earlier `maxShapes` command, all the shapes created so far must be deleted. Further, the existing dynamically allocated array must be de-allocated. Only then can a new array, with the new `shapeCount` value be created. It is critical that your program deletes `Shape` objects and the database array so as no memory leaks occur. Indeed, it should delete all the memory it allocates with `new` before it exits. In this assignment, memory leaks will be checked for and reported by `exercise` and the

```
//
//  Shape.h
//  lab3
//
//  Modified by Tarek Abdelrahman on 2020-10-04.
//  Created by Tarek Abdelrahman on 2018-08-25.
//  Copyright  2018-2020 Tarek Abdelrahman.
//
//  Permission is hereby granted to use this code in ECE244 at
//  the University of Toronto. It is prohibited to distribute
//  this code, either publicly or to third parties.

// ***********  ECE244 Student: DO NOT MODIFY THIS FILE  ***********

#ifndef Shape_h
#define Shape_h

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

class Shape {
private:
    string name;                  // The name of the shape
    string type;                  // The type of the shape (see globals.h)
    int x_location;               // The location of the shape on the x-axis
    int y_location;               // The location of the shape on the y-axis
    int x_size;                   // The size of the shape in the x-dimension
    int y_size;                   // The size of the shape in the y-dimension
    int rotation = 0;             // The rotations of the shape (integer degrees)
public:
    // Build a Shape object with its properties
    Shape(string n, string t, int x_loc, int x_sz, int y_loc, int y_sz);

    // Accessors
    string getType();             // Returns the type
    string getName();             // Returns the name of the shape
    int getXlocation();           // Returns location of the shape on the x-axis
    int getYlocation();           // Returns location of the shape on the y-axis
    int getXsize();               // Returns the size of the shape in the x-dimension
    int getYsize();               // Returns the size of the shape in the y-dimension

    // Mutators
    void setType(string t);       // Sets the type (see globals.h)
                                  // No error checking done inside the method
                                  // The calling program must ensure the type
                                  // is correct
    void setName(string n);       // Sets the name of the shape
    void setXlocation(int x_loc); // Sets location of the shape on the x-axis
    void setYlocation(int y_loc); // Sets location of the shape on the y-axis
    void setXsize(int x_sz);      // Sets size of the shape in the x-dimension
    void setYsize(int y_sz);      // Sets size of the shape in the y-dimension

    void setRotate(int angle);    // sets the rotation of the shape

    // Utility methods
    void draw();                  // Draws a shape; for this assignment it
                                  // only prints the information of the shape
};

#endif /* Shape_h */
```


Figure 1: Defintion of the class `shape`

`autotester`, but there is no penalty for having memory leaks. In future assignments there will be penalties for leaking memory.

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` is released with this assignment. You are encouraged to learn and use this tool. It is used by `exercise` to check for memory leaks in your code.

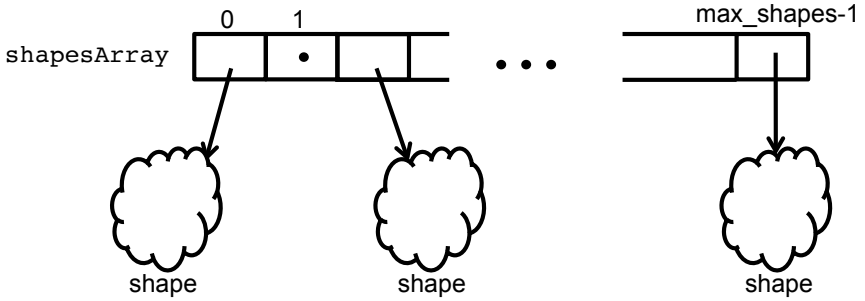

Figure 2: A depction of the database

5 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 `main < 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 subsequent 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).
- 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.

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

6 Examples

6.1 A Short Example

The program when first started, ready to receive input:

```
>
```

Now the user types a command (ending with `Enter`) to create a new database of size 100.

```
> maxShapes 100
```

To which the program should respond with the message indicating the successful creation of new database with 100 entries.

```
New database: max shapes is 100
```

The user then creates a new ellipse called `my_circle` located at `x` and `y` positions of `30` and `40` and with a `x` and `y` sizes of `10` and `10`.

```
> create my_circle ellipse 30 40  10 10
```

To which the program should respond with the message for a successful creation of a shape:

```
Created my_circle: ellipse 30 40 10 10
```

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.

```
> maxShapes 4
New database: max shapes is 4
> create my_circle ellipse 50 65  20 20
Created my_circle: ellipse 50 65 20 20
> create my_square rectangle 100 150 60 60
Created my_square: rectangle 100 150 60 60
> create a_circle circle 120 200 30 40
Error: invalid value
> create my_triangle triangle 40 75 -90 90
Error: invalid value
> create my_rectangle rectangle 100 275 90 180
Created my_rectangle: rectangle 100 275 90 180
> create ellipse rectangle 100 275 90 180
Error: invalid shape name
> create my_rectangle triangle 70 50 10 5
Error: shape my_rectangle exists
> create second_triangle triangle 70 50 10 5
Created second_triangle: triangle 70 50 10 5
> move my_circle
```

```
Error: too few arguments
> mve my_circle
Error: invalid command
> move my_circle 70 90
Moved my_circle to 70 90
> rotate my_rectangle 90 100
Error: too many arguments
> rotate my_rectangle 100
Rotated my_rectangle by 100 degrees
> rotate my_rectangle 400
Error: invalid value
> draw my_trinagle
Error: shape my_trinagle not found
> draw my_circle
Drew my_circle: ellipse 70 90 20 20
> draw all
Drew all shapes
my_circle: ellipse 70 90 20 20
my_square: rectangle 100 150 60 60
my_rectangle: rectangle 100 275 90 180
second_triangle: triangle 70 50 10 5
> delete my_square
Deleted shape my_square
> draw all
Drew all shapes
my_circle: ellipse 70 90 20 20
my_rectangle: rectangle 100 275 90 180
second_triangle: triangle 70 50 10 5
> delete all
Deleted: all shapes
> draw all
Drew all shapes
>
```

7 Procedure

Create a sub-directory called `lab3` in your `ece244` 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 two `.cpp` files in which you will add your code. The first is `parser.cpp` in which you will write the command parser code. The second file is `Shape.cpp` in which you will implement the class `Shape`. Both files are in the directory `parser`. You must not rename these files or add more files. There is also a NetBeans project to help you get started with NetBeans.

The release also contains two include files `globals.h` and `Shape.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` that is used by `NetBeans` to separately compile your project. Do not modify this file either.

Write and test the program to conform to the specifications laid out in Section 4. The hints in

Section 5 may help get you started, and the example sessions in Section 6 may be used for testing.

The `~ece244i/public/exercise` command will also be helpful in testing your program. You should exercise the **executable**, i.e., `parser.exe`, using the command:

```
~ece244i/public/exercie 3 parser.exe
```

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, so you should create additional test cases yourself and ensure you fully meet the specification listed above.

8 Deliverables

Submit the `parser.cpp` and `Shape.cpp` files as lab 3 using the command

```
~ece244i/public/submit 3
```