Lab 4 : Linked Lists

1 Objectives

The main objective of this assignment is to give you practice in creating and manipulating linked lists in C++. In the process, the assignment re-enforces earlier concepts of writing classes with constructors and the destructor. It also serves as an exercise on dynamic allocation and de-allocation of data without memory leaks. This is done by writing a program (called "the program" hereafter) extending your previous lab assignment to represent the `Shapes` database as a set of linked lists, but with a simplified set of commands in which there are (mostly) no errors in input.

2 Problem Statement

The assignment consists of two parts. In the first part, you will write a command parser similar to that of the previous lab. The parser provides a textual input interface to your program. In contrast to the previous lab, you need not check for errors in the input (with some exceptions). These commands create shapes, create *groups* of shapes, move shapes across groups and delete shapes/groups.

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, and print responses. The command parser loops, processing input as long as input is available.

In the second part of the assignment, you will implement a linked list based data structure to serve as a simple "database" of objects that store the created shapes. To do so, you will implement several classes: `Shape`, `ShapeNode`, `ShapeList`, `GroupNode` and `GroupList`.

3 Specifications

Similar to the previous lab, 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 in Section 4 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.

3.1 Coding Requirements

1. The code you will write shall be contained in only the source files named `Parser.cpp`, `Shape.cpp`, `ShapeNode.cpp`, `ShapeList.cpp`, `GroupNode.cpp` and `GroupList.cpp`. Skele-

tons of these files are released with the assignment's `zip` file. The `zip` file also contains corresponding `.h` files: `globals.h`, `Shape.h`, `ShapeNode.h`. `ShapeList.h`, `GroupNode.h` and `GroupList.h`. These `.h` files contain comments that describe the classes defined in them and their methods. Please note that the `.h` files are **NOT** to be modified in any way. Modifying these files almost always results in a mark of 0 for the assignment.

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 `.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.
6. The Standard Template Library (STL) shall not be used. The point of this assignment is for you to create your own linked list implementation.
7. Your program shall not leak memory. Unlike the previous assignment, you shall be penalized for leaking memory.

3.2 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 4 for an example. Input shall always be accepted one line at a time, with each line terminated by a newline character[1].

Unlike the previous lab, you shall assume that input values are error-free (i.e., commands are always valid, integers are always valid, no arguments are missing, no extra arguments, etc.). Nonetheless, there will be some errors you must check for and print an appropriate error message (see Section 3.3). In this case, the input 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[2].

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.

The valid commands, their arguments, and their output if the command and its arguments are all legal are shown below in Table 2. In the case of the `draw` command, which takes no arguments,

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

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

[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.

Argument	Description, type, and range
name	a string consisting of any non-whitespace characters[3]; except strings that represent commands, shape types or reserved words (e.g., `pool`).
type	a string that represents the type of a shape and must be one of: **`ellipse`**, **`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 a shape in either the x or y dimension

Table 1: Acceptable input arguments

the program prints not only the message shown in the table, but also all the groups and shapes in the database (see the example in Section 4).

Command	Arguments	Output if Command is Valid
shape	name type loc loc size size	*name*: *type loc loc size size*
group	name	*name*: group
draw		drawing:
move	name1 name2	moved shape *name1* to group *name2*
delete	name	*name*: deleted

Table 2: Valid commands, arguments and their output

If there is an error, a message shall be displayed as described in Section 3.3. 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.

3.3 Error Checking

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. Please note:

Error message	Cause
invalid name	The name used for a shape or a group is a reserved word (e.g., a command word, a shape type or `pool`)
name exists	A shape/group with the name *name* exists in the database, i.e., has once been created and has not been deleted
name not found	A shape/group with the name *name* specified in a command does not exist

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

1. The commands and the shape types are case sensitive. Thus, while a shape or a group cannot be named pool or triangle, it can be named Pool 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). ***For an empty line of input (nothing or just whitespace) your program should just output the command prompt***.

3. You are guaranteed that autotester will not use test cases that have any errors other than the ones listed above. However, You may reuse your code from lab assignment 3 along with its error checking, if you wish. This part of your code will not be exercised or tested.

3.4 Program Use

Users of the program use the command line to create and delete shapes, similar to the previous assignment. In addition, users can create and delete shape *groups*, where a group is a collection of already created shapes. Finally, users can "draw" the groups and shapes, meaning to print them to the standard output.

A database is used to organize the shapes and the groups. When the database is initially created, there is only one group called pool. A shape command creates a new Shape object and this object is by default a member of the pool group. The group command is used to create a new group, initially empty. The move command is used to move a shape from the group it is currently in to a new group (which can be the one it is currently in). *A shape cannot belong to more than one group.*

The delete command can be used to delete shapes or groups. If used to delete a shape, only the shape specified in the command is deleted from the group it currently in. If the command is used to delete a group, then all the shapes in the group are moved to the pool group and the group is deleted. The pool group cannot be deleted (pool is a reserved word that cannot be used in commands or as shape/group names).

The examples in Section 4 further illustrate the above commands, the actions they take and the output they produce.

3.5 The Database

The program shall keep track of all shapes and groups using a linked list based data structure. An example of which is shown in Figure 1. There is a linked list of type GroupList s that represents the list of groups created during program execution. Each node in the list is a GroupNode, which in turn has a pointer to a ShapeList that represents the shapes belonging to the group. A ShapeList is a linked list of ShapeNodes, where each ShapeNode has a pointer to the Shape object it represents.

At the onset of execution, a GroupList object is created to represent a linked list of GroupNodes. The list contains exactly one GroupNode with the name pool. The ShapeList of the pool is initially empty. As shapes are created, they are added the pool group by inserting them at the *end* of its ShapeList. When new groups are created, they are added to the *end* of the GroupList. Thus, the order of the groups on the GroupList is that of the order in which they are created.

When a shape is deleted, it is removed from the ShapeList of the group it currently belongs to. When a group is deleted, all the ShapeNodes in it ShapeList are added at the end of the pool's ShapeList, keeping their order, and the corresponding GroupNode is deleted from the GroupList.

It is critical that your program deletes all dynamic data it allocates so as no memory leaks occur. In this assignment, memory leaks will be checked for and reported by exercise and the autotester, and *there is penalty for having memory leaks.* It is highly recommended that you use

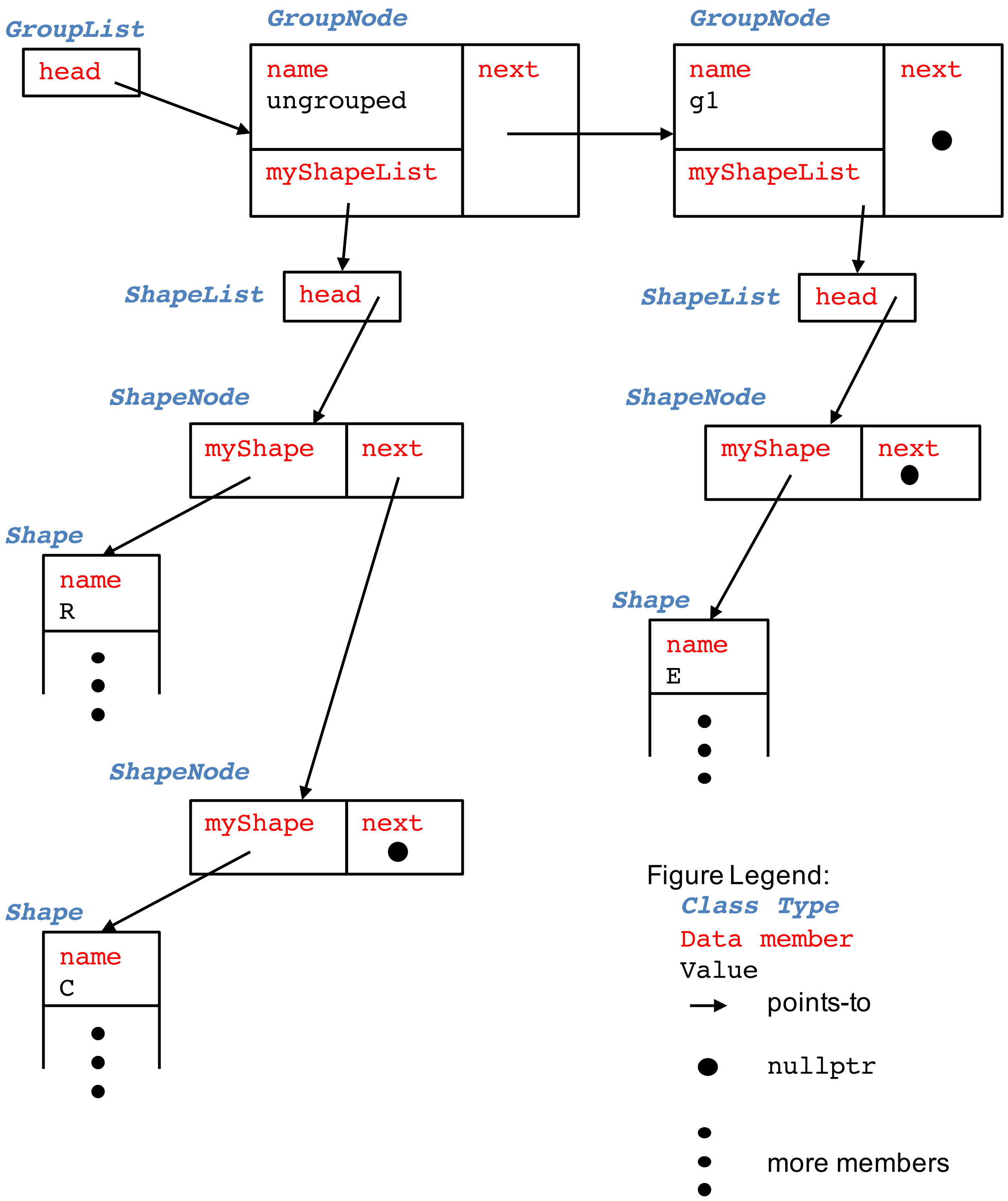

Figure 1: A depiction of the database

the `valgrind` memory checking program. A tutorial on `valgrind` as released with the previous assignment. Please learn and use this tool. It is used by `exercise` to check for memory leaks in your code.

3.6 Program Classes and Files

3.6.1 The Shape Class

The Shape class holds the properties of a shape, including its name, type, location, and size. It is mostly identical to the Shape class in the previous assignment. The definition of the class appears in Shape.h. 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.

3.6.2 The ShapeNode Class

This class defines a node in a linked list of ShapeNodes. It has two data members: myShape, which points to the Shape object associated with the node, and next, which is a pointer to the next ShapeNode in a list. It is defined in the file ShapeNode.h and you must implement this class in the file ShapeNode.cpp. Read through the comments in ShapeNode.h to find out what the methods of the class do.

3.6.3 The GroupNode Class

This class defines a node in a linked list of GroupNodes. It has three data members: name, which stores the name of the group, myShapeList, which points to the ShapeList object associated with the group, and next, which is a pointer to the next GroupNode in a list. It is defined in the file GroupNode.h and you must implement this class in the file GroupNode.cpp. Again, read through the comments in GroupNode.h to find out what the methods of the class do.

3.6.4 The ShapeList Class

This class defines a linked list of ShapeNodes. It contains a single data member, head, which points to the first ShapeNode in the list. It is defined in the file ShapeList.h and the files contains comments that describe what the methods of the class do. You must implement this class in ShapeList.cpp.

3.6.5 The GroupList Class

This class defined a linked list of GroupNodess. It contains one data member called head, which is a pointer to the first GroupNode in the list. The class is defined in the file GroupList.h and you must implement it in the file GroupList.cpp. The comments in the GroupList.h indicate what the methods of the class should do.

3.6.6 The Command Line Parser

A suggested (but not mandatory) structure for your command line parser appears in the skeleton Parser.cpp file released within the assignment's zip file. It is identical to that of the previous assignment and it is recommended that you reuse your code from that assignment.

4 Examples

4.1 An Annotated Example

The program when first started, is ready to receive input:

```
>
```

Now the user types a command (ending with `Enter`) to create a new rectangle called `sqr` located at `x` and `y` positions of `10` and `10` and with a `x` and `y` sizes of `30` and `40`:

```
> shape sqr rectangle 10 10 30 40
```

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

```
sqr: rectangle 10 10 30 40
```

The user then creates a new ellipse called `circ`:

```
> shape circ ellipse 30 40  10 10
```

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

```
circ: ellipse 30 40 10 10
```

The user then creates a new triangle called `trig`:

```
> shape trig triangle 60 80  30 50
```

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

```
trig: triangle 60 80 30 50
```

The user issues a `draw` command and the following is printed:

```
drawing:
pool:
sqr: rectangle 10 10 30 40
circ: ellipse 30 40 10 10
trig: triangle 60 80 30 50
```

Observe how the shapes belong to the `pool` group and are printed in the order in which they were created. Now the user creates a new group called `no_corners`:

```
group no_corners
```

The program should respond by printing:

```
no_corners: group
```

The user issues a `draw` command and the following is printed:

```
drawing:
pool:
sqr: rectangle 10 10 30 40
circ: ellipse 30 40 10 10
trig: triangle 60 80 30 50
no_corners:
```

Observe how the `no_corners` group is printed after the `pool` group and is empty. Now, the user moves the shape `circ` to the group `no_corners`:

```
move circ no_corners
```

The program should respond:

```
moved circ to no_corners
```

The user issues a `draw` command and the following is printed:

```
drawing:
pool:
sqr: rectangle 10 10 30 40
trig: triangle 60 80 30 50
no_corners:
circ: ellipse 30 40 10 10
```

Observe how the `circ` is now in the `no_corners` group and is removed from the `pool` group. The user deletes the `no_corners` group:

```
delete no_corners
```

The program should respond as:

```
no_corners: deleted
```

Finally, the user issues a `draw` command and the following is printed:

```
drawing:
pool:
sqr: rectangle 10 10 30 40
trig: triangle 60 80 30 50
circ: ellipse 30 40 10 10
```

Observe how the group `no_corners` is no longer printed since it is deleted. Also observe how `circ` is back *at the end* of the `pool` group.

4.2 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. Further observe how the

```
> shape my_circle ellipse 50 65 20 20
my_circle: ellipse 50 65 20 20
> shape my_square rectangle 100 150 60 60
my_square: rectangle 100 150 60 60
> shape a_circle ellipse 120 200 40 40
a_circle: ellipse 120 200 40 40
> shape my_triangle triangle 40 75 90 90
my_triangle: triangle 40 75 90 90
> draw
drawing:
pool:
my_circle: ellipse 50 65 20 20
my_square: rectangle 100 150 60 60
a_circle: ellipse 120 200 40 40
my_triangle: triangle 40 75 90 90
> group my_first_group
my_first_group: group
> shape my_rectangle rectangle 100 275 90 180
my_rectangle: rectangle 100 275 90 180
> shape my_rectangle triangle 70 50 10 5
error: name my_rectangle exists
> shape second_triangle triangle 70 50 10 5
second_triangle: triangle 70 50 10 5
> draw
drawing:
pool:
my_circle: ellipse 50 65 20 20
my_square: rectangle 100 150 60 60
a_circle: ellipse 120 200 40 40
my_triangle: triangle 40 75 90 90
my_rectangle: rectangle 100 275 90 180
second_triangle: triangle 70 50 10 5
my_first_group:
> move my_triangle my_first_group
moved my_triangle to my_first_group
> move my_circle my_first_group
moved my_circle to my_first_group
> draw
drawing:
pool:
my_square: rectangle 100 150 60 60
a_circle: ellipse 120 200 40 40
my_rectangle: rectangle 100 275 90 180
```

```
second_triangle: triangle 70 50 10 5
my_first_group:
my_triangle: triangle 40 75 90 90
my_circle: ellipse 50 65 20 20
> delete my_first_group
my_first_group: deleted
> draw
drawing:
pool:
my_square: rectangle 100 150 60 60
a_circle: ellipse 120 200 40 40
my_rectangle: rectangle 100 275 90 180
second_triangle: triangle 70 50 10 5
my_triangle: triangle 40 75 90 90
my_circle: ellipse 50 65 20 20
> delete pool
error: invalid name
> delete abc
error: shape abc not found
>
```

5 Procedure

Create a sub-directory called lab4 in your ece244 directory, and set its permissions so no one else can read it. Download the lab4.zip file, un-zip it and place the resulting files in the lab4 directory.

In the release, there are 6 .h files: globals.h, Shape.h, ShapeNode.h, ShapeList.h, GroupNode.h and GroupList.h. The first has a number of global definitions and the remaining ones define the various classes, as described in Section 3.6. **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 the release, there are also 6 .cpp files in which you will add your code: Parser.cpp, Shape.cpp, ShapeNode.cpp, ShapeList.cpp, GroupNode.cpp and GroupList.cpp. In the first file, you will implement the command parser code. In the rest you will implement the various classes described in Section 3.6.

Finally, 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 3. The example sessions in Section 4 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/exercise 4 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.

6 Deliverables

Submit the Parser.cpp, Shape.cpp, ShapeNode.cpp, ShapeList.cpp, GroupNode.cpp and GroupList.cpp files as lab 4 using the command

```
~ece244i/public/submit 4
```