Lab: Object-Oriented Programming

Topics and References

- Inheritance and run-time polymorphism
- C++ standard library: file I/O, std::string and std::vector

Learning Outcomes

- Inheritance
- Run-time polymorphism
- Practice using C++ standard library
- Practice file I/O
- Practice C++ standard library std::string and std::vector classes
- Practice C++ standard library std::vector class
- Gain practice in reading and understanding code.

Introduction

This lab aims to give you an introduction at the key components of object-oriented programming (OOP) - inheritance and dynamic binding (or run time polymorphism). We consider the problem of storing and handling of a list of shapes that consists of a mix of such different elements as ellipses and polygons. Each ellipse in the list has fixed size structure that describes it. In contrast, polygons are defined with varying number of vertices. Despite of the difference both ellipses and polygons as geometrical shapes have such properties as area, center of mass, perimeter of the border and so on that can be calculated based on their descriptions using relatively simple formulas and algorithms. As you'll see, this problem specification is common to most real-world programming problems and lends itself to an object-oriented solution. Hopefully, implementing this solution will expose you to the language features that C++ offers to support OOP.

Inheritance hierarchy

In our shapes storing problem, we know that ellipses and polygons have some attributes that are common for both shapes and even other shapes - all shapes have an id, color of border, color of internal area inside the border. Shapes can have additional properties, for example, a flag that indicate the shape is selected (so it is visually highlighted on the screen) or indication that the shape is part or a complex shape and so on. Ellipses and polygons use different algorithms for computing the area or center of mass. Such contexts are natural places for inheritance. The basic idea is that we can often think of one class as being just like another, except for some extensions. In this problem, the common attributes of both ellipse and polygon shapes are represented by an abstract base class `Shape`.

> An abstract base class cannot be directly used to create objects. Instead it is used to define an interface to derived classes. A class is made abstract by having a pure virtual function or a protected constructor.

Class `Shape` (defined in `Shape.hpp`) captures the common attributes of any shape. Class `Ellipse` (defined in `ellipse.hpp`) will derive from base class `Shape` to capture the extra requirements for ellipse shapes in the list. Similarly, class `Polygon` (defined in `polygon.hpp`) will derive from base class `Shape` to capture the extra requirements for polygon shapes in the list. Because `Polygon` inherits from `Shape`, every member of `Shape` is also a member of `Polygon` -

except for the constructors and destructor. Likewise, for derived class `Ellipse`. Both derived classes can add members of their own, as we do here with data member `vertices` for class `Polygon` and data members `center`, `a`, and `b` for class `Ellipse`.

Task

In this lab, you'll be doing the following:

1. Read a comma-separated file `shapes.txt` containing details about both ellipse and polygon shapes.
2. Use operator `new` to dynamically allocate memory for each shape object (of type `Ellipse` or `Polygon`) specified in `shapes.txt`; initialize the allocated object using appropriate constructor; and store the pointer to the object in a container `std::vector<Shape*>`. The application requires tasks to be implemented on all shapes, only on ellipse, or only on polygon shapes. To avoid down casting a `Shape*` to a `Polygon*` or a `Ellipse*` during run time, the application uses three separate containers of type `vector<Shape*>` : one to store pointers to all shapes, the second to store pointers to objects of type `Polygon`, and the third to store pointers to objects of type `Ellipse`. Remember only a single object is dynamically allocated for each shape. However, the pointer to this shape object is stored in two containers - the container of pointers to all shape objects irrespective of their type and a second container of pointers to only `Ellipse` shapes or only `Polygon` shapes.
3. Print counts of shapes followed by details of each shape including the shape's id, colors, and parameters. Then, process containers of type `std::vector<Shape*>` to print some statistics related to ellipse, polygon, and all shapes.

How to complete lab

The code for this lab is split across files out of which the following files are provided and should not be modified: `main.cpp`, `shape.hpp`, `ellipse.hpp`, `polygon.hpp`, `process.hpp`. A partially implemented file `process.cpp` is provided. You'll to author source files `shape.cpp`, `ellipse.cpp`, and `polygon.cpp` that will contain the definitions of static data members and member functions of classes defined in the corresponding header files.

We highly recommend that you follow these steps to complete the lab:

1. Begin by reading the source code in function main in `main.cpp` - this step is very important to get started on this lab as it provides you not only the overall program flow but also an idea of the inputs required and corresponding output generated by the program.
2. Read `shape.hpp` to understand the definition of abstract base class `Shape`. Create file `shape.cpp` and add definitions of static data member `Shape::count` and member functions of class `Shape`. For now, provide a skeleton definition of constructor `Shape(std::string&)`. Notice that this class contains a protected member function that must be defined - other derived classes will rely on this function. Ensure `shape.cpp` compiles (use -c option and for now disable -Werror option of g++) before proceeding to the next step.
3. Read `ellipse.hpp` to understand the definition of derived class `Ellipse`. Create file `ellipse.cpp` and add definitions of static data member `Ellipse::count` and member functions of class `Ellipse`. For now, provide a skeleton definition of constructor `Ellipse(std::string&)`. Ensure `ellipse.cpp` compiles before proceeding to the next step.

4. Similarly, read `polygon.hpp` to understand the definition of derived class `Polygon`. Create file `polygon.cpp` and add definitions of static data member `Polygon::count` and member functions of class `Polygon`. For now, provide a skeleton definition of constructor `Polygon(std::string&)`. Ensure `polygon.cpp` compiles before proceeding to the next step.
5. Dummy functions are already defined in `process.cpp`. Later, you'll complete the dummy definitions functions that process containers of type `std::vector<Shape*>`.
6. The next step is the hardest part of the programming task and involves the completion of the dummy definitions of constructors `Shape::Shape(std::string&)`, `Ellipse::Ellipse(std::string&)`, and `Polygon::Polygon(std::string&)`. Begin by examining file `shapes.txt` that contains one record per line with a record describing details of a polygon or ellipse shape. The first few lines of `shapes.txt` look like this:

   > E black white 89,65 31 45
   > P grey none 0,0 10,0 10,10 0,10

   Lines starting with E describe an ellipse shape's record while lines starting with P describe a polygon shape's record.

   The attributes common to all shapes consists of the *shapes' border* and *fill* colors. For example, the first record describes an ellipse with black border and white internal space. Parsing the common attributes in a line (represented by a `std::string`) will be implemented by Shape constructor:

   `Shape::Shape(std::string&);`.

   The extra requirements for ellipse shapes are the position of the central point and lengths of the major (a) and minor (b) radii. For example, the first record specifies the shape's center at point (89, 65) and a, b radii as 31 and 45 accordingly. Constructor `Ellipse::Ellipse(std::string&)` will use base class constructor `Shape::Shape(std::string&)` to initialize the `Shape` part of an `Ellipse` object.

   The extra requirements for polygon shapes is the list of an unknown number of vertices. Unknown means that different polygons may have been defined with different number of vertices. For example, the second record specifies 4 vertices at (0, 0), (10, 0), (10,10), and (0, 10), that define a square with size 10. The Polygon constructor `Polygon::Polygon(std::string&);` will use base class constructor `Shape::Shape(std::string&)` constructor to initialize the Shape part of a Polygon object.
7. The process of parsing `shapes.txt` begins in function `parse_file` in file `process.cpp` . The function uses the file stream parameter (the file has been opened in function `main` !!!) to read each line from the file. After identifying whether a line specifies a polygon or ellipse shape record, memory for the corresponding (`Polygon` or `Ellipse`) object is allocated and initialized on the free store. The pointer returned by operator `new` is inserted to a `std::vector<Shape*>`. The application requires tasks to be implemented only on ellipse and on polygon shapes. To avoid down casting a `Shape*` to a `Polygon*` or a `Ellipse*` during run time, the application uses three separate containers of type `vector<Shape*>`: one to store pointers to all shapes, the second to store pointers to objects of type `Polygon`, and the third to store pointers to objects of type `Ellipse`. The code of this function looks like this:

   ```
   std::string line;
   while (std::getline(ifs, line))
   {
   ```

```
        char c = line[0];
        line = line.substr(2);
        if (c=='E')
        {
          Shape *p = new Ellipse(line);
          vs.push_back(p);  // vs will be used when all shapes
                           //   are to be processed
          ves.push_back(p); // ves will be used when only ellipses
                            //   are to be processed
        }
        else if (c=='P')
        {
          Shape *p = new Polygon(line);
          vs.push_back(p);  // vs will be used when all shapes
                           //   are to be processed
          vps.push_back(p); // vps will be used when only polygons
                            //   are to be processed
        }
        else
          break;
    }
```

8. In file `process.cpp`, complete the definition of function `parse_file` (described above).
9. In file `process.cpp`, implement function `print_records` to print the details of all the shapes that the container (passed as a reference) points to. Note that this function uses dynamic binding (or run-time polymorphism). That is, the function doesn't care about the type (Polygon or Ellipse) of the object that each container element (of type Shape*) points to.
10. In file `process.cpp`, implement function `print_stats` that computes and prints the following:
    1. Number of shapes specified in the parameter of type `std::vector<Shape*>` (this means each element of the container could point to a Ellipse or Polygon object).
    2. The mean (or average) of the area of all the shapes specified in the parameter.
11. A sorted list of shapes based on their areas in descending order. Since the parameter is a reference to a read-only container, you'll have to make a copy of the input container and use std::sort to sort the new copy along with a function object or function that specifies the descending order sorting criterion.
12. A sample output of this function for ellipse shapes is shown below:

> Number of shapes = 7
> The mean of the areas = 7031.78
> The sorted list of shapes (id,center,area) in ascending order of areas:
> 5,39,65,4187.74
> 1,89,65,4382.52
> 12,80,61,6126.11
> 3,19,67,6842.39
> 10,81,65,8243.54
> 8,86,65,9591.28
> 9,49,65,9848.89

13. Note that this function uses dynamic binding (or run-time polymorphism). That is, the function doesn't care about the type (`Polygon` or `Ellipse`) of the object that each container element (of type `Shape*`) points to.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Header file

No header file will be submitted.

Source file

You will be submitting the following files: `shape.cpp`, `ellipse.cpp`, `polygon.cpp`, and `process.cpp`.

Compiling, executing, and testing

Download `vpl-driver.cpp`, `process.hpp`, `process.cpp` (partially complete), `shape.hpp`, `ellipse.hpp`, `polygon.hpp`, input file `shapes.txt`, and following output files: `output-ellipse.txt` (when input to program is and output is 1 only concerned with ellipse shapes), `output-polygon.txt` (when input to program is 2 and the output is only concerned with polygon shapes), and `output-all.txt` (when input to program is 3 and the output is about all shapes in input file). Follow the steps from the previous lab to refactor a makefile and test your program.

Documentation

This module will use Doxygen to tag source and header files for generating html-based documentation. Every source and header file must begin with file-level documentation block. Every function that you declare and define and submit for assessment must contain *function-level* documentation. This documentation should consist of a description of the function, the inputs, and
return value.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit `shape.cpp`, `ellipse.cpp`, `polygon.cpp`, and `process.cpp`.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - *F* grade if your submission doesn't compile with the full suite of `g++` options.

- *F* grade if your submission doesn't link to create an executable.
    - Your implementation's output doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). The auto grader will provide a proportional grade based on how many incorrect results were generated by your submission. *A+* grade if output of function matches correct output of auto grader.
- A maximum of `D` grade if Valgrind detects even a single memory leak or error. A teaching assistance will check you submission for such errors.
    - A deduction of one letter grade for each missing documentation block in a source file. Your submission must have **one** file-level documentation block and function-level

documentation blocks for ***every function you're defining***. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an *A+* grade and one documentation block is missing, your grade will be later reduced from *A+* to *B+*. Another example: if the automatic grade gave your submission a `C` grade and the two documentation blocks are missing, your grade will be later reduced from *C* to *E*.