```
Script started on 2024-05-10 09:40:27-05:00 [TERM="xterm" TTY="/dev/pts/0" COLUMNS=
ee43254@ares:~$ pwd
/home/students/ee43254
ee43254@ares:~$ cat shape.info
Name: Kyle Enkhzul

Class: CSC122-W01



Activity: That's About the Shape of It.

Level: 6, 4 (base program), 2 (counting shape menu option)



Description:



This program allows user to create shapes using polymorphic hiearchy to

create objects that can be different types of shapes and specialization.

It is broken down into something like this:



                                                            Shape

                                |

            --------------------------------------------------

            |                   |                            |

           1D                  2D                           3D

            |                   |                            |

        ----------         -------------            -------------------

        |        |         |           |            |        |        |

       Line    Curve     Circle     Rectangle      Sphere    Cube     Cone



Where the user can specify the type of Shape they want to create and its

certain parameters depending on the type of shape. It then stores it into a

dynamic array which is flexible and allows deletion.
```

```
ee43254@ares:~$ show-code shape.cpp


shape.cpp:


#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include "Point.h"
#include "Shape.h"
#include "OneD.h"
#include "TwoD.h"
#include "ThreeD.h"
#include "Line.h"
#include "Circle.h"
#include "Rectangle.h"
#include "Sphere.h"
#include "Cube.h"
#include "Cone.h"

using namespace std;

// Count the number of each kind of Shape in the current list
void countShapes(const vector<Shape*>& shapes) {
 short lineCount = 0, circleCount = 0, rectangleCount = 0,
  sphereCount = 0, cubeCount = 0, coneCount = 0;

 for (const auto& shape : shapes) {
  if (dynamic_cast<Line*>(shape)) {
        lineCount++;
        } else if (dynamic_cast<Circle*>(shape)) {
         circleCount++;
        } else if (dynamic_cast<Rectangle*>(shape)) {
         rectangleCount++;
        } else if (dynamic_cast<Sphere*>(shape)) {
         sphereCount++;
        } else if (dynamic_cast<Cube*>(shape)) {
         cubeCount++;
        } else if (dynamic_cast<Cone*>(shape)) {
         coneCount++;
        }
}

 cout << "\nNumber of each kind of Shape in the list:\n"
          << "Lines: " << lineCount << "\n"
      << "Circles: " << circleCount << "\n"
          << "Rectangles: " << rectangleCount << "\n"
          << "Spheres: " << sphereCount << "\n"
          << "Cubes: " << cubeCount << "\n"
          << "Cones: " << coneCount << "\n";
}

```

```cpp
int main() {
 vector<Shape*> shapes;

 short choice;
do {
  cout << "\nMenu:\n"
   << "1. Create Shape\n"
   << "2. Print Shape Names\n"
   << "3. Print Shape Information\n"
   << "4. Remove Shape\n"
   << "5. Count Shapes\n"
   << "0. Exit\n"
   << "Enter your choice: ";
  cin >> choice;

switch(choice) {
 case 1: {
 // Create shape based on user input
 string name;
 short shapeType;
 double x, y, z, radius, length, width, height;
 Point point;

  cout << "Enter shape name: ";
  cin >> name;
  cout << "Enter point coordinates (x, y, z): ";
  cin >> x >> y >> z;
  point = Point(x, y, z);

        cout << "Select shape type:\n"
     << "1. Line\n"
     << "2. Circle\n"
         << "3. Rectangle\n"
         << "4. Sphere\n"
         << "5. Cube\n"
         << "6. Cone\n"
         << "Enter choice: ";
        cin >> shapeType;

        // Create shape based on user input and add it to the container
        switch (shapeType) {
         case 1: {
          double x2, y2, z2;
          cout << "Enter end point coordinates (x, y, z): ";
                cin >> x2 >> y2 >> z2;
                Point endPoint = Point(x2, y2, z2);
                shapes.push_back(new Line(name, point, endPoint));
        break;
        }
         case 2: {
          cout << "Enter radius: ";
          cin >> radius;
          shapes.push_back(new Circle(name, point, radius));
        break;
        }
      case 3: {
           cout << "Enter length and width: ";
           cin >> length >> width;
           shapes.push_back(new Rectangle(name, point, length, width));
        break;
        }
         case 4: {
          cout << "Enter radius: ";
          cin >> radius;
          shapes.push_back(new Sphere(name, point, radius));
          break;
        }
         case 5: {
          cout << "Enter side length: ";
          cin >> length;
          shapes.push_back(new Cube(name, point, length));
        break;
     }
         case 6: {
          cout << "Enter radius and height: ";
          cin >> radius >> height;
          shapes.push_back(new Cone(name, point, radius, height));
        break;
        }
        default: {
          cout << "Invalid shape type.\n";
        break;
     }
    }
    break;
  }
  case 2: {
        // Print shape names
        cout << "\nShape Names:\n";
          for (const auto& shape : shapes) {
                shape->print();
          }
        break;
}
  case 3: {
        // Print shape information
        cout << "\nShape Information:\n";
        for (const auto& shape : shapes) {
         shape->print();
         cout << "Coordinates: (" << shape->getPoint().x << ", "
         << shape->getPoint().y << ", " << shape->getPoint().z << ")\n";
        }
        break;
  }
  case 4: {
  // Remove shape based on user input
    if (shapes.empty()) {
        cout << "\nNo shapes to remove.\n";
```

```
    }
    else{
         string name;
         cout << "Enter the name of the shape to remove: ";
         cin >> name;

         auto it = shapes.begin();
         while (it != shapes.end()) {
          if ((*it)->getName() == name) {
           delete *it;
           it = shapes.erase(it);
           cout << "\nShape \"" << name << "\" removed successfully.\n";
          } else {
            ++it;
          }
         }

         if (it == shapes.end()) {
          cout << "\nShape \"" << name << "\" not found.\n";
         }
    }
   break;
   case 5: {
    countShapes(shapes);
   break;
   }
  }
   case 0: {
     cout << "Exiting...\n";
     break;
   }
   default: {
     cout << "Invalid choice. Please try again.\n";
      break;
   }
   }
  } while (choice != 0);

  // Clean up memory
   for (auto& shape : shapes) {
         delete shape;
   }

   return 0;
  }
```

ee43254@ares:~$ show-code Point.h

Point.h:

```
#ifndef POINT_H
#define POINT_H

class Point {
public:
    double x, y, z;
    Point() : x(0), y(0), z(0) {}
    Point(double _x, double _y, double _z) : x(_x), y(_y), z(_z) {}
};

#endif
```

ee43254@ares:~$ show-code Shape.h

Shape.h:

```
#ifndef SHAPE_H
#define SHAPE_H

#include "Point.h"
#include <string>

class Shape {
protected:
    std::string name;
    Point point;
public:
    Shape(const std::string& _name, const Point& _point)
                : name(_name), point(_point) {}
    virtual ~Shape() {}
    virtual void draw() const = 0;
    virtual void print() const {
        std::cout << "Shape: " << name << std::endl;
    }
    const std::string& getName() const { return name; }
    const Point& getPoint() const { return point; }
};

#endif
```

ee43254@ares:~$ show-code OneD.h

OneD.h:

```
#ifndef ONED_H
#define ONED_H

#include "Shape.h"

class OneD : public Shape {
protected:
    OneD(const std::string& _name, const Point& _point)
                : Shape(_name, _point) {}
};

```

```
    12  #endif
ee43254@ares:~$ show-code TwoD.h
```

TwoD.h:

```
     1  #ifndef TWOD_H
     2  #define TWOD_H
     3
     4  #include "Shape.h"
     5
     6  class TwoD : public Shape {
     7  protected:
     8      TwoD(const std::string& _name, const Point& _point)
     9                  : Shape(_name, _point) {}
    10  };
    11
    12  #endif
ee43254@ares:~$ show-code ThreeD.h
```

ThreeD.h:

```
     1  #ifndef THREED_H
     2  #define THREED_H
     3
     4  #include "Shape.h"
     5
     6  class ThreeD : public Shape {
     7  protected:
     8      ThreeD(const std::string& _name, const Point& _point)
     9                  : Shape(_name, _point) {}
    10  };
    11
    12  #endif
ee43254@ares:~$ show-code Line.h
```

Line.h:

```
     1  #ifndef LINE_H
     2  #define LINE_H
     3
     4  #include "OneD.h"
     5
     6  class Line : public OneD {
     7      Point end;
     8  public:
     9      Line(const std::string& _name, const Point& _start, const Point& _end)
    10          : OneD(_name, _start), end(_end) {}
    11      void draw() const override {}
    12  };
    13
    14  #endif
ee43254@ares:~$ show-code Circle.h
```

Circle.h:

```
     1  #ifndef CIRCLE_H
     2  #define CIRCLE_H
     3
     4  #include "TwoD.h"
     5
     6  class Circle : public TwoD {
     7      double radius;
     8  public:
     9      Circle(const std::string& _name, const Point& _center, double _radius)
    10          : TwoD(_name, _center), radius(_radius) {}
    11      void draw() const override {}
    12  };
    13
    14  #endif
ee43254@ares:~$ show-code Rectangle.h
```

Rectangle.h:

```
     1  #ifndef RECTANGLE_H
     2  #define RECTANGLE_H
     3
     4  #include "TwoD.h"
     5
     6  class Rectangle : public TwoD {
     7      double length, width;
     8  public:
     9          Rectangle(const std::string& _name, const Point& _topLeft, double _
    10                  double _width)
    11          : TwoD(_name, _topLeft), length(_length), width(_width) {}
    12      void draw() const override {}
    13  };
    14
    15  #endif
ee43254@ares:~$ show-code Sphere.h
```

Sphere.h:

```
     1  #ifndef SPHERE_H
     2  #define SPHERE_H
     3
     4  #include "ThreeD.h"
```

```
     5
     6  class Sphere : public ThreeD {
     7      double radius;
     8  public:
     9      Sphere(const std::string& _name, const Point& _center, double _radius)
    10          : ThreeD(_name, _center), radius(_radius) {}
    11      void draw() const override {}
    12  };
    13
    14  #endif
ee43254@ares:~$ show-code Cube.h


Cube.h:


     1  #ifndef CUBE_H
     2  #define CUBE_H
     3
     4  #include "ThreeD.h"
     5
     6  class Cube : public ThreeD {
     7      double side;
     8  public:
     9      Cube(const std::string& _name, const Point& _topLeftBack, double _side
    10          : ThreeD(_name, _topLeftBack), side(_side) {}
    11      void draw() const override {}
    12  };
    13
    14  #endif
ee43254@ares:~$ show-code Cone.h


Cone.h:


     1  #ifndef CONE_H
     2  #define CONE_H
     3
     4  #include "ThreeD.h"
     5
     6  class Cone : public ThreeD {
     7      double radius, height;
     8  public:
     9      Cone(const std::string& _name, const Point& _center, double _radius,
    10                  double _height)
    11          : ThreeD(_name, _center), radius(_radius), height(_height) {}
    12      void draw() const override {}
    13  };
    14
    15  #endif
ee43254@ares:~$ CPP shape
shape.cpp***
```

```
ee43254@ares:~$ ./shape.out

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 1
Enter shape name: A
Enter point coordinates (x, y, z): 1 2 3
Select shape type:
1. Line
2. Circle
3. Rectangle
4. Sphere
5. Cube
6. Cone
Enter choice: 1
Enter end point coordinates (x, y, z): 3 4 5

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 1
Enter shape name: B
Enter point coordinates (x, y, z): 2 1 3
Select shape type:
1. Line
2. Circle
3. Rectangle
4. Sphere
5. Cube
6. Cone
Enter choice: 2
Enter radius: 2

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 1
Enter shape name: C
Enter point coordinates (x, y, z): 3 2 1
Select shape type:
```

```
1. Line
2. Circle
3. Rectangle
4. Sphere
5. Cube
6. Cone
Enter choice: 3
Enter length and width: 4 3

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 1
Enter shape name: D
Enter point coordinates (x, y, z): 5 4 3
Select shape type:
1. Line
2. Circle
3. Rectangle
4. Sphere
5. Cube
6. Cone
Enter choice: 4
Enter radius: 5

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 1
Enter shape name: E
Enter point coordinates (x, y, z): 4 5 6
Select shape type:
1. Line
2. Circle
3. Rectangle
4. Sphere
5. Cube
6. Cone
Enter choice: 5
Enter side length: 3

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 1
Enter shape name: F
Enter point coordinates (x, y, z): 2 3 1
Select shape type:
1. Line
2. Circle
3. Rectangle
4. Sphere
5. Cube
6. Cone
Enter choice: 6
Enter radius and height: 2 3

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 2

Shape Names:
Shape: A
Shape: B
Shape: C
Shape: D
Shape: E
Shape: F

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 3

Shape Information:
Shape: A
Coordinates: (1, 2, 3)
Shape: B
Coordinates: (2, 1, 3)
Shape: C
Coordinates: (3, 2, 1)
Shape: D
Coordinates: (5, 4, 3)
Shape: E
Coordinates: (4, 5, 6)
Shape: F
Coordinates: (2, 3, 1)
```

```
Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 5

Number of each kind of Shape in the list:
Lines: 1
Circles: 1
Rectangles: 1
Spheres: 1
Cubes: 1
Cones: 1

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 1
Enter shape name: G
Enter point coordinates (x, y, z): 6 7 8
Select shape type:
1. Line
2. Circle
3. Rectangle
4. Sphere
5. Cube
6. Cone
Enter choice: 3
Enter length and width: 3 2

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 2

Shape Names:
Shape: A
Shape: B
Shape: C
Shape: D
Shape: E
Shape: F
Shape: G

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 3

Shape Information:
Shape: A
Coordinates: (1, 2, 3)
Shape: B
Coordinates: (2, 1, 3)
Shape: C
Coordinates: (3, 2, 1)
Shape: D
Coordinates: (5, 4, 3)
Shape: E
Coordinates: (4, 5, 6)
Shape: F
Coordinates: (2, 3, 1)
Shape: G
Coordinates: (6, 7, 8)

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 5

Number of each kind of Shape in the list:
Lines: 1
Circles: 1
Rectangles: 2
Spheres: 1
Cubes: 1
Cones: 1

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 4
Enter the name of the shape to remove: A
```

```
Shape "A" removed successfully.

Shape "A" not found.

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 4
Enter the name of the shape to remove: B

Shape "B" removed successfully.

Shape "B" not found.

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 4
Enter the name of the shape to remove: C

Shape "C" removed successfully.

Shape "C" not found.

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 4
Enter the name of the shape to remove: LOL

Shape "LOL" not found.

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 5

Number of each kind of Shape in the list:
Lines: 0
Circles: 0
Rectangles: 1
Spheres: 1
Cubes: 1
Cones: 1

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 2

Shape Names:
Shape: D
Shape: E
Shape: F
Shape: G

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 3

Shape Information:
Shape: D
Coordinates: (5, 4, 3)
Shape: E
Coordinates: (4, 5, 6)
Shape: F
Coordinates: (2, 3, 1)
Shape: G
Coordinates: (6, 7, 8)

Menu:
1. Create Shape
2. Print Shape Names
3. Print Shape Information
4. Remove Shape
5. Count Shapes
0. Exit
Enter your choice: 0
Exiting...
ee43254@ares:~$ cat shape.tpq
1.How many libraries did you create for your hierarchy? Do all of them have

both interface and implementation files?
```

I have created 11 libraries for each kind of object. I used one implementation file as that is easier for me and Jason James to grade. I got away with not using implementation files for each library by including a default constructor in each library file as well as linking them all together with #include.

2. How can you store information about so many different classes in a single container?

I could store pointers to objects of different classes in a single container such as a vector of pointers. Since they are all derivedf from 'Shape', I can use polymorphism to store different derived class objects.

3. What does that new keyword virtual have to do with any of this?

The new keyword virtual is used to declare a member function in the base class that can be overridden in derived classes. It is super helpful when derived classes have the same function but have more specific uses. It is pretty much the thing that allows polymorphism.

4. Will you ever need/want to create an object of type Shape, OneD, TwoD, or ThreeD? How can you assure that this won't happen?
No, they are abstract classes and are meant to be base classes for other classes such as Circle, Rectangle, Cube, and can not be instantiated. I can assure this by making their constructors protected.

5. What other methods/operators might prove useful in an application for drawing shapes? What if the application were more of a computer-aided instruction in geometry? Is there a need to limit your classes?
(Note: You don't have to implement these, I'm just looking for descriptive responses.)

In a drawing application, additional methods/operators that might prove useful include:

Transformations: Methods to translate, rotate, scale, or skew shapes.
Color and Style: Methods to set colors, line thickness, fill patterns, etc.
Selection and Editing: Methods to select and manipulate individual shapes.
Grouping and Layering: Methods to group shapes together and manage layers.

If the application were more focused on computer-aided instruction in geometry, you might need additional classes to represent geometric concepts such as points, lines, angles, polygons, etc. There may also be a need for classes to perform geometric calculations and validations. The need to limit your classes depends on the specific requirements of your application.

6. What kind of container should you use to store the Shapes: dynamic array, static array, templated dynamic Arrayclass, vector, ...? Since this lab has nothing to do with array management, what would be the most appropriate/easiest choice?

The most appropriate and easiest choice for storing Shape objects would be a std::vector. std::vector provides dynamic resizing, efficient element access, and supports polymorphism through storing pointers to the base class.

Additionally, it manages memory automatically, reducing the risk of memory leaks and simplifying memory management. Since this lab doesn't focus on array management, using std::vector is a convenient and efficient option.

MORE TPQS

1. Which methodology is more general/portable?

Both RTTI and dynamic_cast are general and portable methods for determining the types of objects at runtime. However, dynamic_cast provides more control and safety in downcasting, making it preferable in scenarios where downcasting is required.

2. Which embodies the motto: "Work smarter, not harder"?
Both of them embody the motto, however they work just slightly different depending on the need of the programmer. If one needs to downcast, dynamic_cast is the way to go. If one needs a more general use without the need of third party or external libraries, RTTI is the way to go.

3. Other than to count them, what other reason might you have to verify a type at run-time?

There could be several reasons needed to verify a type at run time but one case is polymorphic behavior. When working with polymorphic class hierarchies, you may need to verify the type of an object at runtime to determine its behavior. Depending on the type, you may execute different methods or algorithms. ee43254@ares:~$ exit

exit

Script done on 2024-05-10 09:44:31-05:00 [COMMAND_EXIT_CODE="0"]