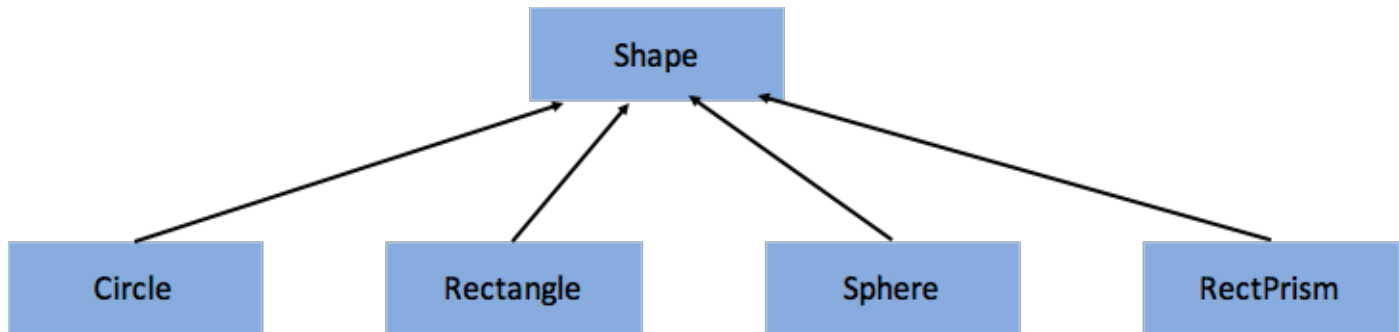


MP10

MP 10 - Introduction to C++

Introduction

Your task in MP10 is to implement the Shape hierarchy shown here:



Shape class is the base class. Circle, Rectangle, Sphere, RectPrism are derived classes. The base class contains function `getName()` which return the name (Circle, Rectangle, Sphere, RectPrism) of an object. For each derived classes, you need to implement `getArea()`, `getVolume()`, and overloading operator `+` and operator `-`. Your program should read the shapes defined in `test.txt`, and create a list of Shape pointers that points to objects of each input shape. Different constructor should be called according to the input shape. For example, if the input shape is a rectangle, class Rectangle's constructor should be called to initialize the length, width, and name of the rectangle object. After reading the file, given an array of shape pointers, you have to implement `MaxArea()` and `MaxVolume()` which return the max area and max volume respectively.

The format of the test cases is as follows:

```
<#objects>
<name0> <var1> <var2> ..
<name1> <var1>
```

where `<#objects>` gives the number of objects defined in the test case. Starting from the second line, each line defines a shape object. `<name>` gives the name of the object. `<var1> <var2>...` are the parameter required to initialize an object. For circle and sphere, the only parameter is radius. For rectangle, the parameters are `<width> <length>`. For rectangle prism, the parameters are `<width> <length> <height>`.

Below is a sample input-output pair.

Input:

```
4
Circle 2
Rectangle 2 3
Sphere 2
RectPrism 1 2 3
```

Output:

```
max area = 50.2655
max volume = 33.5103
```

The Pieces

In this MP, you are given a set of files:

- `main.cpp` - The source file that contains the main function.
- `shape.hpp` - The header file of the shape hierarchy. You should write your code here.
- `verify.cpp`, `check.cpp` - The source files of the verification program.
- `check.hpp` – The header file of the verification program.
- `gshape.hpp` – The header file of the gold version.
- `gshape.o` – The object file of the gold version.

Details

Shape hierarchy

The header file of the shape hierarchy is given. Your task is to implement the member functions of the class.

```
//shape.hpp
class Shape{
public:
    Shape(string name);
    string getName();
    virtual double getArea() = 0;
    virtual double getVolume() = 0;
private:
    string name_;
};
```

Iterators and lists

Iterators are a feature of C++ that allows us to traverse a data structure, similar to the


```
for (int i = 0; i < length; i++)
```

form of traversing an array you have used throughout the course.

Iterators provide an interface to traverse data structures.


In this MP, you need to construct and traverse a linked list using C++'s **list** implementation.

Similar to **vectors**, there are a variety of functions that can help you manipulate their elements.

For detailed documentation on these functions, see here <https://www.cplusplus.com/reference/list/list/>  [\(https://www.cplusplus.com/reference/list/list/\)](https://www.cplusplus.com/reference/list/list/)

Iterators for different levels of functionality depending on the structure they iterate over.

In the case of a C++ list, you cannot randomly access elements with an index (such as using `list[3]`), and need to traverse the list with an iterator.

For a list of allowed operations for iterators, see here: <https://www.cplusplus.com/reference/iterator/>  [\(https://www.cplusplus.com/reference/iterator/\)](https://www.cplusplus.com/reference/iterator/)

To get an iterator to the first element of a list, used

```
list<int>::iterator it = somelist.begin();
```

Likewise, to get an iterator to the end of a list (meaning, 1 beyond the last element), use

```
somelist.end();
```

Iterators can be incremented to move them to the next item in the list

```
it++;
```

Like pointers, iterators can be dereferenced to access the element they reference.

```
*it = 3;  
  
printf("%d ", *it);
```

You can also compare iterators using `==` and `!=`. For example, the code below will compare an iterator to the end of a list, i.e. determining if the iterator is within the bounds of the list.

Note that for list iterators, you cannot use `<`, `<=`, `>` or `>=` to compare them.

```
it != somelist.end();
```

Combining all these, you can create a loop to iterate through and print all the elements of a list:

```
for (list<int>::iterator it =somelist.begin(); it != somelist.end(); it++) {  
    printf("%d ", *it);  
}
```


Read Input File

You have to implement `CreateShapes()` to read the input file and initialize corresponding objects. `CreateShapes()` return a list of pointers (list) point to the objects initialized according to the input data.

```
list<Shape*> CreateShapes(char* file_name);
```

For example, to initialize a Circle object with radius 2, you may use the following codes:

```
Shape* shape_ptr = new Circle(2);
```

To read the input file, you can use ifstream. You can find more information about ifstream on the website <http://www.cplusplus.com/reference/fstream/ifstream/>  (<http://www.cplusplus.com/reference/fstream/ifstream/>)

An example of using ifstream to read input file is as follows:

```
//test.txt  
circle 2
```

```
//in main()  
String name  
int r  
ifstream ifs ("test.txt", std::ifstream::in);  
ifs >> name >> r;  
ifs.close();
```

By executing the code piece above, "circle" is stored in name and 2 is stored in r

MaxVolume() and MaxArea()

Given a list of object pointers, you have to implement `MaxArea()` and `MaxVolume()`. `MaxArea()` and `MaxVolume()` compute each objects area and volume, and return the maximum area and maximum volume respectively.

```
double MaxArea(list<Shape*> shapes);  
double MaxVolume(list<Shape*> shapes);
```

Operator Overloading

For the purposes of this MP, we have defined the addition and subtraction of 2 Rectangles/Circles/Spheres/RectPrisms to be as follows:

Rectangles

Given **$R3 = R1 + R2$** : it follows that

length $R3 = \text{length } R1 + \text{length } R2$

width $R3 = \text{width } R1 + \text{width } R2$

Given **$R3 = R1 - R2$** : it follows that

length $R3 = \max(0, \text{length } R1 - \text{length } R2)$

width $R3 = \max(0, \text{width } R1 - \text{width } R2)$

Circles

Given **$C3 = C1 + C2$** : it follows that

radius $C3 = \text{radius } C1 + \text{radius } C2$

Given **$C3 = C1 - C2$** : it follows that

radius $C3 = \max(0, \text{radius } C1 - \text{radius } C2)$

Rectangular Prisms

Given **$RP3 = RP1 + RP2$** : it follows that

length $RP3 = \text{length } RP1 + \text{length } RP2$

width $RP3 = \text{width } RP1 + \text{width } RP2$

height $RP3 = \text{height } RP1 + \text{height } RP2$

Given **$RP3 = RP1 - RP2$** : it follows that

length $RP3 = \max(0, \text{length } RP1 - \text{length } RP2)$

width $RP3 = \max(0, \text{width } RP1 - \text{width } RP2)$

height $RP3 = \max(0, \text{height } RP1 - \text{height } RP2)$

Sphere

Given **$S3 = S1 + S2$** : it follows that

radius $S3 = \text{radius } S1 + \text{radius } S2$

Given $S3 = S1 - S2$: it follows that

radius $S3 = \max(0, \text{radius } S1 - \text{radius } S2)$

Computing Area and Volume

The fomulas for computing areas and volumes are listed as below for your reference.

Rectangle R

area of R = length * width

volume of R = 0

Circle C

area of C = $\text{radius}^2 * \text{PI}$

volume of C = 0

Rectangular Prism RP

area of RP = 2 (*length* width + length *height* + width height)

volume of RP = length *width* height

Sphere S

area of S = 4 *PI* radius^2

volume of S = $(4.0 / 3.0) \text{radius}^3 \text{PI}$

Rectangles and Templates

You may notice that, unlike the other shapes, Rectangles have a line before their definition.

```
<template class T>
```

Templates are another function of C++ that allows us to work with generic types.

Note that in the provided skeleton, width *and* length are of type T. Using templates, we can have this T be different data types, without needing to rewrite the class for each data type.

Essentially, T becomes a placeholder for whatever type we want our rectangle to use.

For instance, by creating a constructor with T,

```
Rectangle<T>(T width, T length):Shape("Rectangle") {
    // some code here
```

```
}
```

we can construct a Rectangle that uses integers for its width and length, or one that uses floats, or doubles.

```
Rectangle<int>(2, 4);
```

```
Rectangle<float>(2.2, 4.31);
```

```
Rectangle<double>(2.2, 4.31);
```

You have already encountered templates - the C++ vector and list implementations are templated based on what we are storing in them. (ex. list)

Note that getVolume and getArea have a return type of double, even for Rectangles.

For Rectangle you will need to cast your results to double before returning them.

```
return (double) area;
```

You may also need to cast when using functions like max, which expects the two inputs to be of the same type.

```
T num1;  
int num2;  
max(num1, (T) num2);
```

Building and Testing

To compile your program, type the following command:

```
make
```

To execute your program, type :

```
./mp10 test1.txt
```

After you finish your implementation, you could verify your program by using a provided test program. To Verify your program, use the following command

```
make verify  
./verify_mp10 test1.txt
```

The program will invoke a hidden checker to test your program and display the verification results.

```
----- Begin Verifying MP10 -----  
getName() 6/6
```

```
Rectangle: 16/16
Circle: 16/16
Sphere: 16/16
RectPrism: 16/16
MaxArea(): 10/10
MaxVolume(): 10/10
CreateShape() 10/10
Your total Score for MP10: 100/100
----- End Verifying MP10 -----
```

Grading Rubric

Functionality (100%)

- getName() (6%)
- Class Rectangle (16%)
 - 4% for each function
- class Circle (16%)
 - 4% for each function
- class Sphere (16%)
 - 4% for each function
- class RectPrism (16%)
 - 4% for each function
- MaxArea() (10%)
- MaxVolume() (10%)
- CreateShape() (10%)
- **If your code does not compile, you will get 0.**