12_shapes: Vererbung, Wertobjekte und vieles mehr

Dipl.-Ing. Dr. Günter Kolousek

1 Allgemeines

- **Es gelten die gleichen Richtlinien wie beim ersten Beispiel!!!**

Please apologize for switching to English but this excercise is a recycled one...

2 Aufgabenstellung

The purpose of this exercise is to train the capabilities of writing classes, understanding of inheritance and polymorphism, and, particularly, to improve the comprehension of classes and objects.

Assumption: We want to develop the next (and later on: the only) CAD (computer-aided design) program to build buildings and all other constructible things. We don't want to be modest ;-)

But to be a bit more realistic, we have to start at the very beginning. Ok, what are the base geometrical shapes?

Here are some ones: point, line, square, rectangle, triangle, circle, ellipse, cuboid,... to name only a few.

To simplify the domain a bit further, we restrict ourself to two-dimensional geometry.

3 Anleitung

1. Let's start with the simplest one – the point.

   Write a class `Point` inside a header-only modul `shapes` that represent a point in the euclidian plane. That sounds simple!

   But what instance variables should we implement? Which visibility will we attach? Which methods are necessary? Which methods make sense?

   a) First things first, the instance variables are really obvious: `x` and `y` and the datatype should be `double`. What else?

      Which visibility? Object-orientated programming heavily relies on encapsulation. That means the access of the instance variables from outside should be restricted as much as possible. Hence, the visibility modifier has to be...

      Did you remember that the private part of the class should be placed near the closing curly brace of the the class definition? Why? Because it's an implementation detail that should not be visible at first sight.

b) Next, we have to make a decision: Do we want to make the objects of this class mutable or immutable? That means: Should it be possible to change the values of the instance variables (the state of this object) or the instance variables do not change at all.

   Trust me, we will make this class immutable. Therefore, no setter methods will be implemented.

   Ok, then implement getters and name it `get_x` and `get_y`. Don't forget about `const`!

c) Fine, but how to fill a concrete instance with values? We have to implement a constructor. Go!

   That's easy, but does it work? Who knows? We must see it... Up to the next item!

d) A nice class is one which is able to generate a nice string representation of itself. A nice representation for an instance of `Point` would look like `Point(3.0, 4.0)`. Hence, implement an appropriate overloaded operator `operator<<`!

e) So, it's testing time. Implement a `main` - method that creates an instance of `Point` with the coordinates $x = 3$, $y = 4$ and outputs it. Then, compile your program and run it.

f) So far, so good. We implemented a getter method as we would do it e.g. in Java. But remember we implemented an immutable class, i.e. there will be no changes at all! Wouldn't it be better to eliminate the getter methods?

   What are the consequences of this?

   - The instance variables `x` and `y` have to be public.
   - So that there are no modifications possible they have to be constant over time.
   - You have to adapt your operator `operator«` and...

g) Now we want to move a point somewhere by Δx and Δy. Ok, easy, implement a method `move` which moves the point by `dx` and `dy`.

h) Really that easy? What does your method looks like? Did you changed `x` and `y`? Yes? Fine!

i) No, not fine at all! Do you remember that our class should be immutable? That `move` has to return a new point! Therefore, implement the following prototype: `Point move(double dx, double dy) const`. Test it!

   You already know why we specify `const` inside this particular prototype, right?

j) It looks like the Point behaves like it should. Sometimes it would be necessary to instantiate a `Point` with default values. Add the following statement to your test code inside `main()`:

```
vector<Point> points(10);
for (auto& p : points)
    cout << p << endl;
```

   - What should it do?
   - Why does it fail to compile?
   - How does the correct fix looks like?

   Do you know the answers?

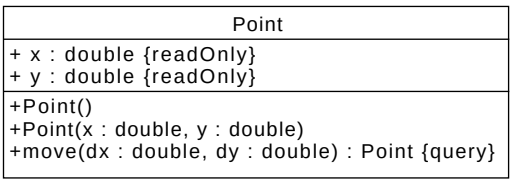
   - Yes? Fine!
   - No?
     - At least the first answer should be obvious...
     - Why does it fail? → No default constructor!

  - How to fix? → `Point() = default;` will do the trick!
    * Does it compile? No? Maybe you forgot to initialize our neat but constant data members? Oh, I love the curly braces `;)`

  Fix it!

  Now your class `Point` should look like in the following class diagram:

  An adequate class diagram will look so:



k) So we tackled the "problem" of inserting instances of `Point` into a `vector`. What about inserting a `Point` into a `set`. Let's try by appending the following statements to `main()`:

```
set<Point> points2;
points2.insert(Point{});
```

  You got really plenty of error messages? Yes, that's `C++`. That's because of the template mechanism of `C++` and the absence of relevant type information inside the template specification and, therefore, the instantiating of templates failes miserably. `C++20` will get us – so called – "concepts" that will reduce the count of compiler messages significantly but up to now...

  What do these compiler messages mean? Take a look please!

  Ok, I hope you take my advice seriousely but if you not figured it out on your own: We have to implement the `operator<`! Why? Because `set` is defined that way and usually implemented by an balanced tree!

  I see, but how to order 2D points? There is no such thing of a natural ordering! Take it easy and do it the same way as e.g. tuples are sorted in Python and so forth.

  The prototype of this member functions is `bool operator<(const Point& o) const`.

  So, go ahead and implement the operator < accordingly.

l) Fine, now we are able to sort a collection of `Point` instances using `std::sort` and insert it into collections based on the operator <. Next, we try to do the following inside `main()`:

```
unordered_set<Point> points3;
points3.insert(Point{});
```

  More compiler errors than ever?!

  To insert objects in an unordered collection like `unordered_set` the requirements for those objects are that there must be an appropriate

  - operator ==
  - structure `std::hash<Point>` with an overloaded operator `()`

  defined!

  - The operator == is also a member function of the class and is really easy to implement. Implement it!
  - The structure `hash` should be defined like shown in the following code snippet:

```
template<>
struct std::hash<Point> {
    size_t operator()([[maybe_unused]]const Point& p) const {
        return std::hash<double>()(p.x)...;
    }
};
```

Replace `...` so that the return value of the operator `()` is constituted of the "exclusive or" of the hash value of `p.x` and `p.y`. Fine, now try to understand the whole snippet. Btw, `template<>` means that the template is a specialization of another template and needs no further formal type parameter. Notice, the whole structure has to be belong to the namespace `std`.

Did you recognize how elegantly a call operator `()` could be defined?

Btw, if you don't want to pollute the namespace `std`, you must define two user defined structs (both with an appropriate overloaded operator `()`) and provide both of them when instantiating the `unordered_set` as further template arguments. But this little info is only for the ambitious coders...

Now you be happy (or the compiler whoever) since the code will compile again!

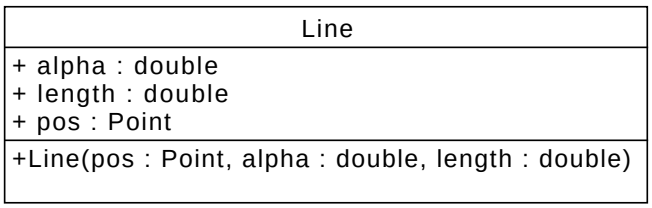
2. Next, we want to represent a line which starts and ends by a point. A line consists of two connected points.

   a) Again, do not implement a command line user interface, it's not really necessary. But do not forget about some testing statements inside `main`. Yes, unit testing would be definitely better but who cares? `;-)`

   b) In contrast to `Point` we decide that the class should be *mutable*. Should we use setter and getter methods for accessing the point, the angle α, and the length? Hmm, this is a difficult question. On the one hand this enforces encapsulation and reduces dependencies, on the other hand it makes the interface more complex.

   Hmm, what to do? We consult the `C++` Core Guidelines and refer to the specific guideline C.131: Avoid trivial getters and setters. Go!

   Hence, the design is laid down in the following class diagram:

   c) If you look at the class diagram carefully you will see that a line is represented in our design by the starting point (`pos`, the position), the length of the line, and the angle `alpha` (angle to the x-axis, counter-clockwise). Later on, you will see that this will ease our implementation very much.

   But there are some drawbacks, too. You *have to* check against a "negative" length (to be more concrete: we will consider only values greater than zero as valid values) and you have to calculate the modulo of the passed `alpha` because there is no such thing as an angle greater than 2π (or 360°)! Be aware that the operator `%` is only defined for integers... The appropriate function which calculates the modulus of floating point numbers is called `fmod` defined within the header `cmath`.

You see, there we have a little problem: There are some constraints on setting the data fields! Therefore, our access to these fields need to be guarded. We need setters and getters!

Line
-alpha : double -length : double { length > 0 } +pos : Point
+Line(pos : Point, alpha : double, length : double) +Line(start : Point, end : Point) +set_length(length : double) +get_length() : double { query } +set_alpha(alpha : double) +set_alpha_degree(alpha : double) +get_alpha() : double { query }

- If the user provides a "negative" value for the length of the line raise an exception of type `invalid_argument`.
- Do you remember, that computers love "radians" and don't consider "degrees" as first-class citizens? Therefore, the functions `sin`, `cos`, and `tan` expect a radian as argument. Furthermore, `asin`, `acos`, and `atan` returns the angle in radians! So, you have to convert...

  It's up to you if you store the angle in radians or degrees. But the user interface has to provide both possibilities for convenience and easy use. Anyway, the value has to be in the range of $[0, 2\pi)$ resp. $[0, 360)$! This has to be adjusted accordingly inside the setter method.

  Clearly, it would be better that we can differentiate between radians and degrees syntactically but this could be treated later on.

  Just in case: Don't forget about DRY!!!
- Alternatively, we could represent a line by a start point and an end point. For such use cases, there is an additional constructor necessary that takes two points and calculates the angle `alpha` as well as the length `length`.
- There are two getters. Do you know if the compiler is happy about the following snippet?

  ```
  Line l1{{0, 0}, 1, 0};
  l1.get_alpha();
  ```

  Of course, but is it meaningful to query the angle and ignore it afterwards? In this particular case there is little or nothing in it, isn't it? So, we learn about another attribute `[[nodiscard]]`:

  ```
  [[nodiscard]]
  double get_alpha()...
  ```

  Adapt it and test it!

  Next, add it to the other getter.

d) Implement a method `move` which moves the whole line by the given parameters `dx` and `dy`. The return type should be `Line&` so that we can implement "method chaining" that looks like in the following:

```
line.move(3, 4).move(2, 3);
```

Of course, this particular example makes no sense at all but if we would implement another operation like `rotate`...

Try to implement the requested behaviour!

Please try it!

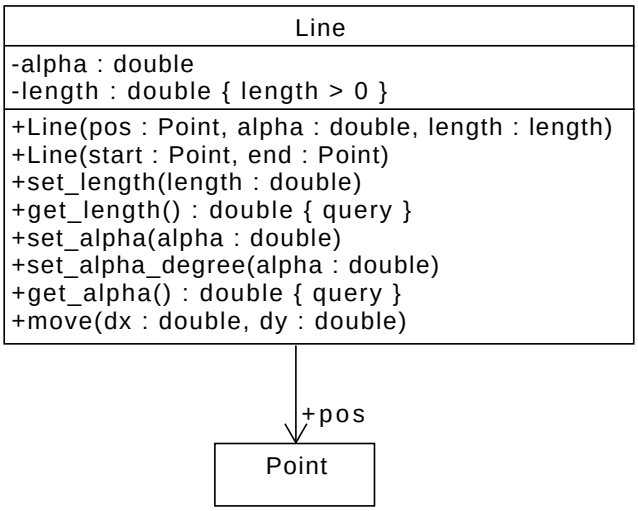
Ok, no chance! Why?

The position is a immutable object and, therefore, it couldn't be changed. Hmm, it seems there is still a design flaw…

So, let's change it so that we have to *replace* the point in question by changing the representation from physical containment of `pos` to a weaker form of logical containment by storing a pointer to an instance of `Point`. It's clear, no raw pointer!!!

Also, I hope it does not come to your mind to do such nonsense like `make_unique<Point>(&p)`...

Even if this could not be clearly represented in a class diagram we will represent it in the following way:

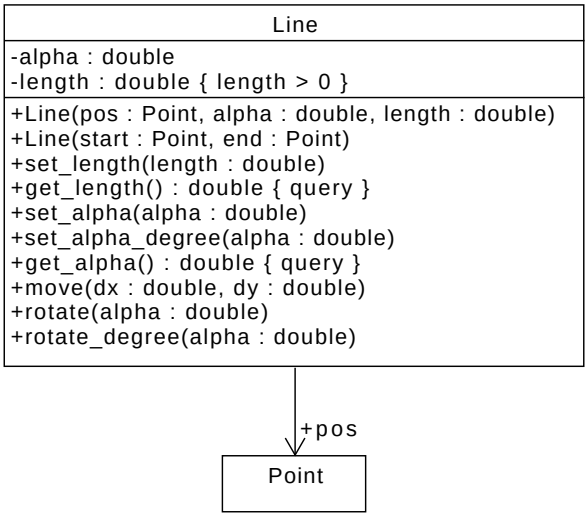


In fact, there is no semantic difference to the previous class diagram as far as it is concerned to the UML but it should express our intent to store a pointer. Remember, there is absolutely *no* semantic difference!

If you argue that this a shortcoming of UML than you have to consider that the UML class diagram should merely and mainly represent the logical view. In case you insist to represent such scenarios you are free to use the extension possibilities of UML. But again, this is another story.

e) Implement the methods `rotate` and `rotate_degree` which rotates the line by the given angle `alpha` to the position of the line counter-clockwise. Again, it should return a reference to itself.

Help! How to rotate a line by a given angle about the position of the line… counter-clockwise? That's really easy because of our chosen representation of a line… just adding angles (modulo!)



3. Next, we want to implement a square.

   a) First we have to decide about an appropriate signature of the constructor.

      - Four points: Simple, but then we have to verify that it is really a square. Not that this is too complicated…
      - Four lines: Same as above.
      - One point (left bottom), length of a side, angle between the side (next to the given point counter-clockwise) to the x-axis. Here, we do not have to verify the constraints! Fine, we take this approach.

      So, it is obvious which instance variables we will define.

   b) Then, implement the relevant getter and setter methods as before.

   c) You didn't forget about a nice `operator«`, did you?

   d) Next, implement the methods `void move(double dx, double dy)`,`void rotate(double alpha)` (rotates about the left bottom point of the square!), and `rotate_degree(double alpha)`. Look, how easy it is with this representation (i.e. with our chosen data representation).

4. Ok, now for something completely different. No, not really, just kidding! We want to implement a class which represents a rectangle.

   a) What is a rectangle? Something like a square, but with potentially different sides. Hmm, so it would be possible to implement a class which inherit from the class `Square` and add another side. This sounds well. But it violates the liskov substitution principle. Hence no solution at all.

   b) Ok, then we could try it the other way around to inherit `Square` from `Rectangle`. Yes, a `Square` is-a `Rectangle` but it violates the liskov substitution priciple as well. This, too, is no solution to the problem. As we already know from the ellipse-circle-problem there is no inheritance relationship between these two classes.

   c) So, implement `Rectangle` on its own. Now! `Square` could be a pattern for you!

5. If we inspect our already written lines of code, we see that we implemented `move`, `rotate`, `set_alpha`,… more than once!

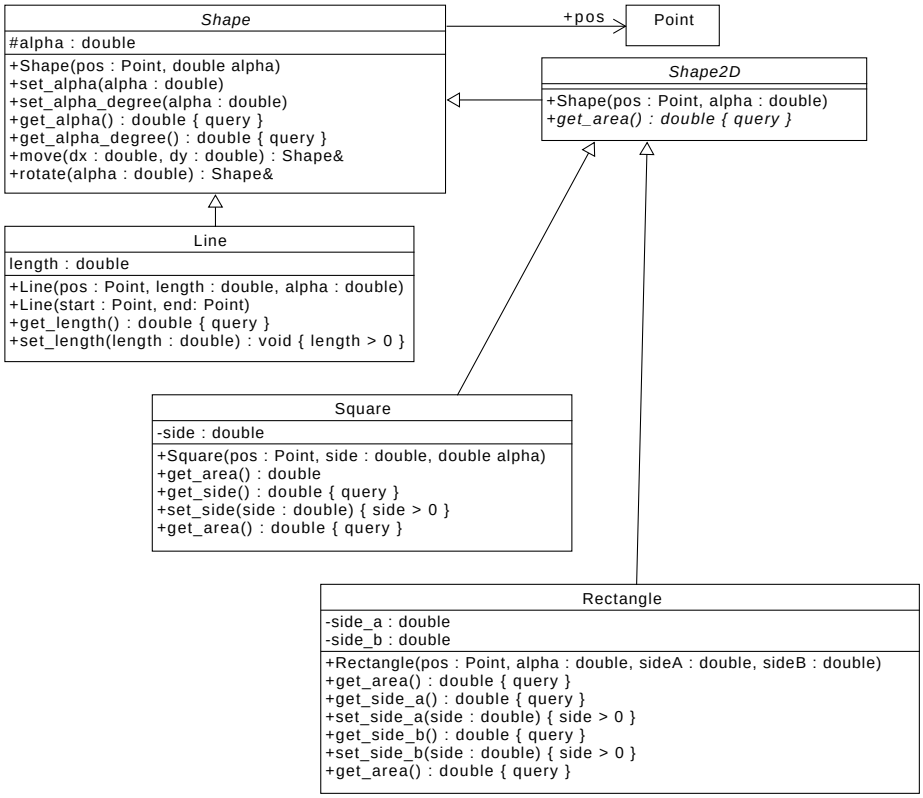
   There are several kinds of shapes: 0-dimensional (a point), 1-dimensional (a line or a polyline), 2-dimensional (e.g. a square). For our purposes we have, up to now, no need to differentiate between

0-dimensional and 1-dimensional shapes. But we could take advantage of a class `Shape2D` which could be a super class to `Square` and `Rectangle`.

We define a 2D shape as something which has a left bottom point, an angle α and consists of several sides (must be greater than zero as usual) and build up an area in the euclidean space. Then, our existing classes `Square` and `Rectangle` behave exactly like a 2D shape.

So, which functionality could `Shape2D` be consists of: attribute `pos`, `set_alpha`, `set_alpha_degree`, `get_alpha`, `move`, `rotate`. Hmm, we could also consider to derive `Line` from `Shape2D`. But `Line` is no 2-dimensional shape. Ok, but does it harm to derive `Line` from `Shape2D`? Yes, imagine if we will add a method `abstract double get_area()` to `Shape2D` (and we will!) then it is not feasible any more.

So, we will introduce the next abstraction (`Shape`) and this leads to the following class diagram:



Implement this class hierarchy but leave the abstract method `get_area()` for the moment. Nevertheless, be aware of the visibility specifications in particular `protected`.

Also, add the method `get_alpha_degree()`. Afterwards, it will be handy.

6. Next, implement the method `get_area()` along the inheritance tree. By the way, you will learn to specify an abstract method and, hence, construct an abstract class. Do not forget about the specifier `override`!

   You will remark that there you get compiler warnings and, subsequently, errors since... Read it yourself!

   Ok, that's clear but why is it so? We know the compiler will generate a destructor on his own. Yes, but this desctructor will not be `virtual`. So, you have to specify that a default destructor has to be generated *that* should be `virtual`...

7. I suppose that you did't forget the `const` of `get_area()`, did you?

8. We already implemented an inheritance tree. That's fine. Yet another way is to add an additional method to `Rectancle` to check if the particular instance is a square:

   ```
   bool is_square() const;
   ```

We have to check if both sides are equal. Maybe you think that's an easy task because you have just to use the == operator for checking the equalness of both sides.

But: Don't do it this way! Why? Because we can not be sure that both sides have *not* been calculated before passing it to the constructor or the respective setter method. And if floating point numbers are being calculated we must not test for equality simply by using the operator ==!

We assume that the side will be calculated before a rectangle will be created. Therefore, we perform the check the following way:

```
bool is_square() {
    return abs(side_a - side_b) < EPS;
}
```

where EPS is a constant, e.g.with a value of `1e-5`. Ok, it's not totally correct but it works at least partially.

Hence, implement this method inside `Rectangle`.

9. Furthermore, it makes sense to write a method `Square to_square()` that returns an instance of `Square` iff (if and only if) the actual instance of `Rectangle` is really a square, otherwise throw an `domain_error`.

10. Next, we tackle polymorphism...

    We now have points, lines, squares, and rectangles. But what could we do about them? For this particular use case extend our function `main()` so that it contains a new `vector` of `unique_ptrs` of `Shape`-instances and fill it with a `Line`, a `Square`, and a `Rectangle`.

    Next, iterate over this container and call the `rotate` and `move` (see the next item, but the output will be handled later on). Nothing particularly complicated... This is just for training and entertainment.

11. Next, we want to output a representation of each referred object of the container. The output should look like in the following printing:

```
Line({1, 1}, 1.41421, 45)
    move it by (1, 1): Line({2, 2}, 1.41421, 45)
    rotate it by 45ř: Line({2, 2}, 1.41421, 90)
Square({2, 2}, 2, 0), Area=4
    move it by (1, 1): Square({3, 3}, 2, 0)
    rotate it by 45ř: Square({3, 3}, 2, 45)
Rectangle({3, 3}, 3, 3, 45), Area=9
    move it by (1, 1): Rectangle({4, 4}, 3, 3, 45)
    rotate it by 45ř: Rectangle({4, 4}, 3, 3, 90)
```

    How to achieve this? Yes, it's a bit problematic because the function `operator<<` is not polymorphic and, up to now, there is no `operator«` at all for the class `Shape`.

    a) First things first. We need function `operator<<` for the class `Shape`. Implement such a thing that outputs "Quaxi". No need to worry! We replace it in just within a minute by something more useful.

    b) Next, we need a *polymorphic* member function because a ordinary function can not be polymorphic at all. I suggest the following inside the class `Shape`:

    ```
    ostream& print(ostream&) const;
    ```

    Of course, it should be abstract.

    c) Then, don't forget about a virtual destructor of `Shape`!

    d) Next, we have to adapt `operator<<` of `Shape` accordingly to use this new member function to use print the representation to the given output stream.

e) It's time for writing the actual member function definitions for the relevant classes.

f) Furthermore, the operator << functions of the subclasses of `Shape` are not needed anymore.

g) Lastly, you can (and should) remove the virtual destructor of `Shape2D`. It's also not needed.

You will see that though the vector contains pointers to objects of the static type `Shape` the right methods will be executed. I.e, independent of the static type the method of the concrete type will be selected. That's the sense of polymorphism. In combination with inheritance we get a highly efficient tool.

12. If the current instance represents a `Shape2D` call the `get_area` method.

    `C++` currently does not have a complete RTTI (runtime type information) system. Therefore, to get the actual type of an object is a bit tricky and depends an polymorphic classes (since it's based on vtable, btw).

    You have to test for a specific type by casting it the following way:

    ```
    if (Shape2D* s{dynamic_cast<Shape2D*>(o.get())}) {
    ```

    Of course, `o` denotes the concrete `unique_ptr` instance to a particular shape.

    Try to analyze this statement and implement it!

4 Übungszweck dieses Beispiels

- Recognize abstractions and build new ones
- select suitable methods and choose visibility carefully
- mutable vs. immutable objects
- `override`, `virtual`, abstract method using `= 0`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `pow`, `sqrt`, `fmod` of `<cmath>` resp. `<cstdlib>`
- convert degree and radian
- DRY
- modulo operation for floating point numbers
- `public` inheritance
- `protected` visiblity
- circle-ellipse problem
- abstract methods and abstract classes
- polymorphism
- testing equality to zero of floating point numbers
- implementing `operator<`, `operator==`, `operator()`
- `std::set` and `std::unordered_set` and defining appropriate classes for storing in such containers, in particular implementing `std::hash`.
- "instance of" in `C++`
- implementing method chaining
- defining default `virtual` destructors

- getters and setters in `C++`
- attribute `[[nodiscard]]`
- UML class diagrams
- implementing polymorphic output functions along the inheritance tree