# Programming Technique

## Lab 8 – Class & Object oriented programming

### 1 Expected outcomes

- Understand basic object-oriented concepts
- Can create a class and use it to declare an instance of that class.
- Know how to call a method using a class instance.
- Know how to design simple classes to solve a problem.
- Understand inheritance.
- Understand basic class diagram.
- Know polymorphism.

### 2 Mandatory exercises

**Exercise 1.** Given the following program:

```
#include <iostream>
#include <iomanip>
using namespace std;
class A{
private:
	int m_value;
public:
	A() : m_value(123){
		cout << "In A(), value = " << m_value << endl;
	}
	A(const A& ob) : m_value(ob.m_value){
		cout << "In A(const A& ob), value = " << m_value << endl;
	}
	A(int value) : m_value(value){
		cout << "In A(int value), value = " << m_value << endl;
	}
	~A(){
		cout << "In ~A(), value = " << m_value << endl;
	}
};

int main(){
	A ob1;
	A *ptr1 = new A(246);
	{
		A ob2(456);
```

```
            A ob3(ob1);
        }
        delete ptr1;
        A ob4(789);
        A ob5 = 357;
        return 0;
};
```

a) Run the program. Print and explain the results.
b) What is function ~A()? When is it called?
c) If we remove `public:`, will there be an error? Why?
d) If `private:` is replaced with `protected:`, will there be an error? Why?
e) If we remove the constructor A(), will there be an error? Why?
f) Does removing `A(const A& ob)` lead to any error? Why? If the compilation is successful then what will be printed?
g) If we remove the destructor, will there be an error? Why? What will it print?
h) Instead of initialize m_value as follow `A() : m_value(123)`, there is another way. Rewrite A() using the other way to initialize m_value. Compile the new codes and run it.

Guidelines:

- Run the codes to see the results.
- Change the codes accordingly to the steps above, explain the results.

**Exercise 2.**

Given the following program:

```
class X{
public:
        int m_value;
};

class Y : public X{
public:
        int m_value;
};
class Z : public Y{
public:
        Z(int vx, int vy, int vz){
                //(1)
        }
public:
        int m_value;
};
int main(){
        Z obj(1, 2, 3);
        //(2)
        return 0;
};
```

a) Fill out `//(1)` to initialize m_value from X, Y and Z using vx, vy and vz respectively.
b) Fill out `//(1)` to print all three different values for m_value (each for X, Y and Z respectively).

Guidelines:

- To use overriden members of parent classes, you can use the scope operator (see slides).

**Exercise 3.**

Given the following program:

```
#include <iostream>
using namespace std;
class X{
public:
	void display(){
		cout << "class X" << endl;
	}
};
class Y : public X{
public:
	void display(){
		cout << "class Y" << endl;
	}
};
class Z : public Y{
public:
	void display(){
		cout << "class Z" << endl;
	}
};
int main(){
	Z obj;
	obj.X::display();
	obj.Y::display();
	obj.Z::display();
	obj.display();
	return 0;
};
```

a) Is there any compilation error? If no, what does the program print?
b) If we change `public` to `protected` in `class Z : public Y`, will there be any error? Why?
c) If we change `public` to `protected` in `class Y : public X`, will there be any error? Why?
d) If we change `public` to `protected` in `class X`, will there be any error? Why?

Guidelines:

- Run the codes to see the results.
- Change the codes accordingly to the steps above, explain the results.

**Exercise 4.**

Given the following program:

```
#include <iostream>
using namespace std;
class X{
public:
	/*(1)*/ void display(){
		cout << "Type of \"this\" object is: Class X" << endl;
	}
};
class Y : public X{
public:
	void display(){
		cout << "Type of \"this\" object is: Class Y" << endl;
	}
};
class Z : public Y{
public:

	void display(){
		cout << "Type of \"this\" object is: Class Z" << endl;
	}
};
int main(){
	cout << "Call method via NON-POINTER variable - CASE I:" << endl;
	X x;
	Y y;
	Z z;
	x.display();
	y.display();
	z.display();
	//////////////////
	cout << " Call method via NON-POINTER variable - CASE II:" << endl;
	X x1, x2, x3;
	Y y2;
	Z z3;
	x2 = y2; x3 = z3; //ATTENTION
	x1.display();
	x2.display();
	x3.display();
	//////////////////
	cout << " Call method via POINTER variable - CASE I:" << endl;
	X *px = new X();
	Y *py = new Y();
	Z *pz = new Z();
	px->display();
	py->display();
	pz->display();
	delete px;
	delete py;
	delete pz;
	//////////////////
	cout << " Call method via POINTER variable - CASE II:" << endl;
```

```
        X *px1 = new X();
        X *px2 = new Y(); //ATTENTION
        X *px3 = new Z(); //ATTENTION
        px1->display();
        px2->display();
        px3->display();
        delete px1;
        delete px2;
        delete px3;
        return 0;
};
```

a) Compile, run and explain the results.
b) Insert `virtual` keyword to (1) (display method in class X). Compile, run and explain the results.

**Guidelines:**

- Calling a method without pointer: assigning an instance of a derived class to an instance of a base class is allowed. This is because the instance of the derived class has all members of the base class (i.e. the instance of the derived class contains an instance of the base class). The reverse does not hold true; however, therefore you can't assign an instance of the base class to an instance of the derived class.
- Calling a method using pointer: a pointer pointing to the base class is always pointing to an instance of the base class. For example: `X *px2 = new Y();` and `X *px3 = new Z();`. This assignment is a hint of polymorphism in OOP. See slides.

**Exercise 5.**

Given the following program:

```
#include <iostream>
using namespace std;
class X{
public:
        /*(1)*/ void display(){
                cout << "Type of \"this\" object is: Class X" << endl;
        }
};
int main(){
        X x;
        Y y;
        Z z;
        x.display(); //OK
        y.display(); //OK
        z.display(); //Error during compilation
        return 0;
};
```

a) Define a new class called Y, which must inherit from base class X. Override the display method in Y so that it can be used outside class Y. With this, `y.display();` in the main function can be run without error.
b) Define a new class called Z, which must inherit from base class Y. Override the display method in Z so that it **can't** be used outside class Z. With this, `z.display();` in the main function yields an error. You must complete this task using two different ways.

**Guidelines:**

There are two methods to solve problem b):

- Using protected or private (default) inheritance. Example: `class Z: protected Y`.
- Use public inheritance, but when you override display in z, put the overriding method under the `private` keyword.

**Exercise 6.**

An instance of type Date stores information about date, month and year. Create a Date data type so that a programmer can run the following program:

```
1  #include <iostream>
2  #include <iomanip>
3  #include "Date.h"
4
5  using namespace std;
6
7  int main(){
8      Date d1, d2;
9      cout << "Enter date:";
10     cin >> d1;
11     cout << "Enter date:";
12     cin >> d2;
13
14     cout << "Date 1:" << d1 << endl;
15     cout << "Date 2:" << d2 << endl;
16     cout << (d1 < d2? "d1 < d2" : "d1 >= d2") << endl;
17
18     Date d3(25,2,2017);
19     cout << d3.toString() << endl;
20
21     return 0;
22 };
```

- Second line: insert header of class Date created by you.
- Line 8: declare two instances of type Date with some invalid default values. For example: year = - 1. Best practice: use system date to assign as default value. This makes it easier for other programmers to get the current Date and time (just by declaring new instances).

- Line 10 and 12: class Date supports inputting Date by using the >> operator. You can input a date by typing **5/2/2017** and hit ENTER.
- Line 14 and 15: class Date supports outputting Date by using the << operator. With the sample input above, it should print **05/02/2017** (pad 0 before dates or months with only one digit).
- Line 16: class Date supports comparison between two Date instances.
- Line 18: class Date allows instance initialization by passing values into the constructor.
- Line 19: class Date supports toString method. This method convert the Date instance into a string. With this method, line 19 will print **25/05/2017**.

**Guidelines:**

- Operator overloading: check array slides.

**Exercise 7.**

Build a data type called MyString. MyString provides easy string managements for other programmers: creating strings, string concatenation, printing string, inputting string, find substring, etc. You must manage your string using pointer and dynamic memory allocation.

**Note:** when building MyString, you must not use <string> provided by C++. You can only use <iostream> or <string.h> (or <memory>).

a) If you use <string.h>: you can only use strlen and strcpy.
b) If you use <memory>: you should re-implement strlen by yourself and use memcpy to replace strcpy.

The class MyString must have the necessary constructors, destructors, methods an operators for the following block of codes to run without any error:

```
36  int main(){
37      MyString str1("Programming fundamentals");
38      MyString str2 = "C++";
39      MyString str3 = str1 + " " + str2;
40      cout << str3 << endl;
41      MyString str4;
42      str4 = str3;
43      MyString str5;
44      cout << "Enter a String:";
45      cin >> str5;
46      str5 += " " + str4;
47      cout << str5;
48      return 0;
49  };
```

- Line 37: you must add a constructor so that a MyString instance can take in a **literal string**.
- Line 38: add a constructor so that you can initialize a MyString instance with a **literal string** on the right side of "=".
- Line 39: support string concatenation using "+".
- Line 40: print string object using "<<".
- Line 41: support empty string initialization.
- Line 42: support assignment between two instances of the same type MyString.
- Line 45: support MyString input from keyboard.
- Line 46: support MyString concatenation using "+=".

**Guidelines:**

1) With the requirement that you have to use dynamic memory allocation, your string data (a member of the class MyString) can be created as follow: `char* m_char;`
2) **Note:** we don't have to store the number of characters the string has because we always know where it ends (a string always ends with zero, that is, the char '\0').
3) Once a member in a class is dynamically allocated, the class must always follow this workflow:
    - Dynamically allocate the data in the constructor using the `new` keyword.
    - Free the allocated data in the destructor, using `delete`.
    - You must write a function for copy initialization and a copy assignment operator (=). These two members will ask for more memory and copy data into these new memory space. Without this, your program will get

an error sooner or later because all different instances share the same memory space. Each instance also has its own deletion (in the destructor) -> the same memory space is freed multiple times -> error.

4) Other operators: refer to the slides about array.

**Exercise 8.**

Assume you are developing a program involving drawing 2D and 3D objects. This program is clearly related to two most basic objects: point and vector in 2D space. (We will consider 3-dimensional objects later).

Requirement: build classes for "Point2D" and "Vector2D". You should design them so that they are convenient. The template source is attached to this document. Your task is to:

a) Implement all methods if they are not complete (has TODO comment).
b) Write the main program to create objects and do computation based on them to verify if your methods were correct. This is the same as creating "test-cases".

**Guidelines:**

Properties and methods of "Point2D" and "Vector2D" are designed using the following diagram:

| Point2D | | | |
|---|---|---|---|
| - | | x : double | |
| - | | y : double | |
| - | <<StaticAtrribute>> | radius : float = 1.5f | |
| + | <<Getter>> | getX () | : double |
| + | <<Setter>> | setX (double newX) | : void |
| + | <<Getter>> | getY () | : double |
| + | <<Setter>> | setY (double newY) | : void |
| + | <<Getter>> | getRadius () | : float |
| + | <<Setter>> | setRadius (float newRadius) | : void |
| + | <<StaticMethod>> | distanceAB (const Point2D& a, const Point2D& b) | : double |
| + | <<StaticMethod>> | vectorAB (const Point2D& a, const Point2D& b) | : Vector2D |
| + | <<Constructor>> | Point2D (double x, double y) | |
| + | | operator+ (const Vector2D& vector) | : Point2D |
| + | | operator- (const Point2D& point) | : Vector2D |

| Vector2D | | |
|---|---|---|
| - x : double | | |
| - y : double | | |
| + <<Getter>> | getX () | : double |
| + <<Setter>> | setX (double newX) | : void |
| + <<Getter>> | getY () | : double |
| + <<Setter>> | setY (double newY) | : void |
| + <<Constructor>> | Vector2D (const Point2D& head, const Point2D& tail) | |
| + <<Constructor>> | Vector2D (double x, double y) | |
| + | operator* (double factor) | : Vector2D |
| + | operator+ (const Vector2D& vector) | : Vector2D |
| + | operator- (const Vector2D& vector) | : Vector2D |
| + | dot (const Vector2D& vector) | : double |
| + | ortho (const Vector2D& vector) | : bool |

Not every operator belongs to class "Vector2D". For example, the operator supporting multiplication between a factor (real-valued number) and a vector:

```
factor * vector
```

I) Properties:
Both "Point2D" and "Vector2D" have two member variables x and y to store their coordinates. For Point2D, to actually draw it on screen, we need to know its "radius" (how big is the point we want to draw). Therefore, in addition to x and y, class Point2D also has a radius. This radius should be the same among all Point2D instances therefore we design it with `static` property. This exercise's aim is to help you understand what it means by using `static`. In the above diagram, radius is assigned with 1.5f by default.

II) Methods:

a) **Constructors**: these help initialization of "Point2D" and "Vector2D" instances easier. You should design them carefully.

Constructor for class Point2D:

- Allows instance initialization with coordinates x and y -> this constructor must have two parameters x and y, respectively.
- We don't have to implement copy initialization like we did in the previous exercise. This is because members of class Pointer2D are real values, not pointers. Therefore, C++, by default, copy their values when we use assignment operator. For the same reason, copy assignment is also not needed.

Constructor for class Vector2D:

- Allows instance initialization with coordinates x and y -> this constructor must have two parameters x and y, respectively.
- There should also be another constructor that takes in two Point2D: **head** and **tail**. The coordinates x and y of the Vector2D instance will be calculated from **head** and **tail** using the formula: **head** – **tail**. That is, x of vector equals to x of head minus x of tail, similar for y.

b) **Destructors**: There is no need for them in this exercise.

c) **Other methods**:

- For Point2D, you must add another method called "distanceAB" to calculate the distance between two points passed as arguments. When this method is called, it does not need to refer to any particular instance of class Point2D, therefore we can design it with the `static` property.
- Similarly, method "vectorAB" takes in two Point2D instances and return a newly-calculated vector from A to B. It's also `static`.
- Class Vector2D should have two methods: **dot** and **ortho**.
  - The **dot** method computes dot product between two vectors. For example, the dot product between two vectors [a b] and [c d] are a*c + b*d.
- The **ortho** method check if two vectors are orthogonal. Two vectors are orthogonal if their dot product is 0.
- Both **dot** and **ortho** should be `static`. They take in two vectors as arguments.

d) Operators for Point2D (red) and Vector2D (blue):

- [Point A] + [Vector V] → [Point B], moving A along V to get B. (Translation).
- [Point A] - [Point B] → [Vector V], vector V starts from B and ends at A.
- [Real number F] * [Vector V] → [Vector T], multiply F with each component of V, we get T (shrinking/expanding V to get T). This operator **cannot be a member of class Vector2D**. This is because the left term (F) is not a Vector2D.
- [Vector V] *[Real number F] → [Vector T], same as above except the position between F and V are reversed. This can be defined as a member of Vector2D because V is on the left of *.
- [Vector U] + [Vector V] → [Vector T], vector addition, x of T equals to x of U + x of V, the same goes for y.
- [Vector U] - [Vector V] → [Vector T], similar to the above.

**Exercise 9.**

Assume that you must develop a program that can draw 2D/3D geometric shapes. Apart from basic 2D objects such as Points and Vectors (we will consider 3D later), there should be more complicated objects such as polygon, triangle, quadrilateral, square, rectangle, etc. This exercise ignores generic quadrilateral for simplicity.

**Requirements:**

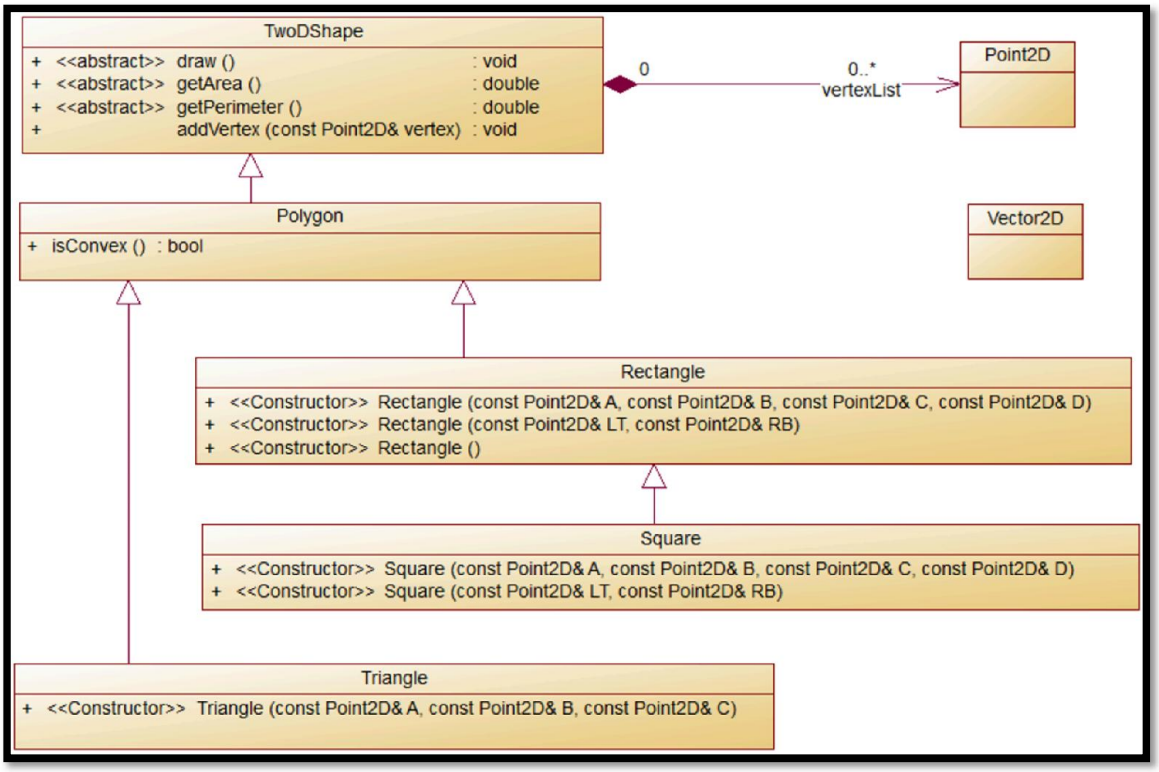


Diagram legend:

- +: These members are **public**.
- Arrow with big white head: inheritance. In this example, Square inherit from Rectangle. Read: "**Square is a Rectangle**".
- Arrow with thin head (between TwoDShape and Point2D): an instance of type TwoDShape always have a member called vertexList. This member is a list of Point2D instances. There are many ways to implement this list (array, vectors, etc.). In the codes attached to this document, vertexList is implemented using vector (which needs `#include <vector>` on top of the file).

a) Covert the above diagram to C++ codes. Classes such as Point2D, Vector2D and TwoDShape are attached with this document,

b) Implement constructors and destructors of the following classes:
   - Triangle, Rectangle, Square: inserting A, B, C and D into vertexList one by one (inherited from TwoDShape). For triangle, only A, B and C need to be inserted. Refer to the **legend** section below the diagram.
   - For Rectangle and Square: we need another constructor that takes only two points, left-top (LT) and right-bottom (BT). This constructor will calculate four vertices A, B, C and D of the rectangle/square based on LT and BT and add them to vertexList.

c) Overriding all methods inherited from TwoDShape in all of its derived classes, **if it is necessary**. These methods are:
   - **getArea**: calculate the area of the polygon and return it. The default getArea of TwoDShape return 0. **In polymorphism, this is an "abstract method"**. Note: only override this method if it is necessary.
   - **getPerimeter**: calculate the perimeter of the polygon and return it. **In polymorphism, this is an "abstract method"**. Note: only override this method if it is necessary.
   - **draw**: this method draw its respective instance on screen (*`this`). However, this course does not offer you any tool to draw. Therefore, you only need to print the list of Vertices on screen. You can also print more details about the shape if you want. **In polymorphism, this is an "abstract method"**.
   - **addVertex**: add a new vertex to vertexList.
   - **isConvex**: check if the vertices in vertexList forms a convex polygon. If the polygon is not convex, **getArea** will always return 0. You can check this link https://stackoverflow.com/questions/471962/how-do-determine-if-a-polygon-is-complexconvex-nonconvex for Jason S's algorithm.

d) After defining addVertex in Triangle, Rectangle and Square, move it from `public` to `private` – this is so that codes outside of them can't add more vertex to their vertexList.

**Guidelines:**

I) Properties:
   The property of classes TwoDShape is vertexList as explained before.

II) Methods:
   a) **Constructors**:

   Constructor for class TwoDShape: the default construct suffices.

   Constructor for class Triangle: it should take in 3 numbers.

   Constructor for class Rectangle and Square:

- There should be a constructor that takes in 4 points that makes a Square/Rectangle. This constructor adds them into vertexList one by one.
- Additionally, there must also be another constructor that takes in LT and RB. This constructor calculates 4 vertices of a Square/Rectangle based on LT and RB and add them into the vertexList.
- However, unlike a triangle, there can be cases when 4 inputted vertices do not form a rectangle/square. If they don't, announce an error. The best practice here is to use exception/try-catch but you haven't learned that yet. Therefore, we can always pretend that the 4 inputted vertices form a rectangle/square.

b) Destructors: they are not needed here. The reason is that the list of vertices in Vector stores instances instead of pointers.

c) Other methods:

- According to the diagram above, the following classes all inherited addVertex (`public`) from TwoDShape: Polygon, Triangle, Rectangle, Square. For Polygon, it is logical to keep addVertex as a public method because a polygon can have as many vertices as the programmer wants. However, a triangle always has three vertices, a square or a rectangle always has 4 vertices. Therefore, letting addVertex be a public method in them makes less sense. We should not allow other programmers to add more vertex to a triangle. We need to hide addVertex in Triangle, Rectangle and Square. But how?
    - We can override this method, move it under a `private` keyword.
    - We can still let it be `public`, but we can override this method so that it is empty (does not actually add a new vertex to vertexList). Similarly, we can just call addVertex of TwoDShape.