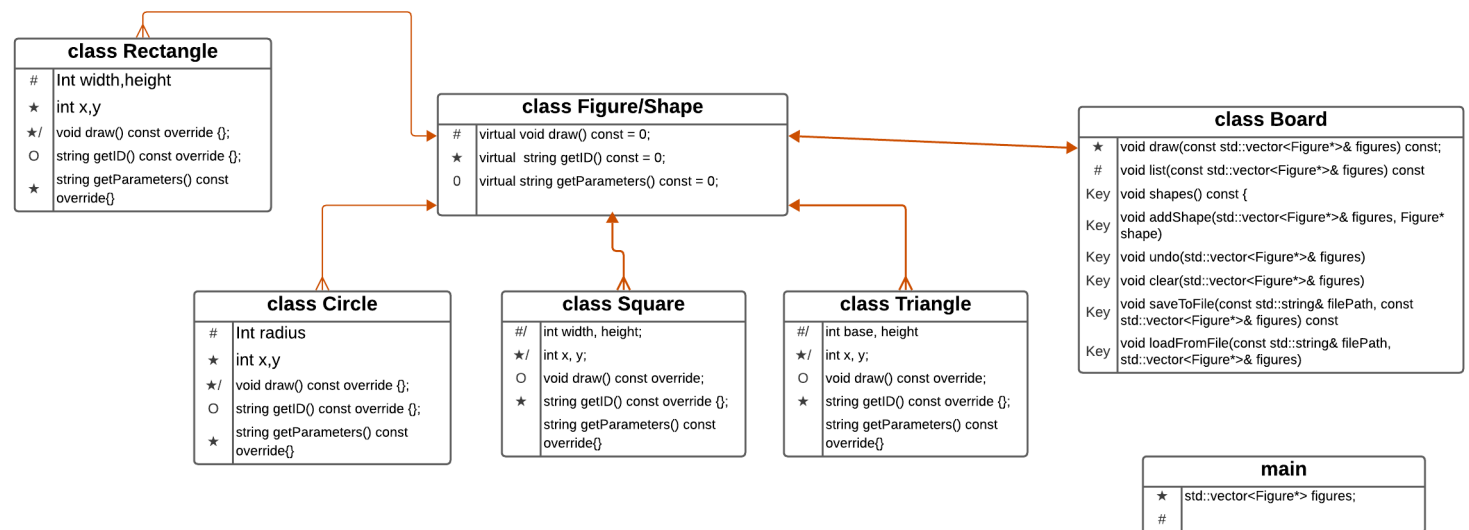


# Report Template: Object-Oriented Programming and Design

## Anna Yevtushenko OOP2 TASK:

<https://github.com/Anna-Yevtushenko/OOPD-assignment2.git>

Develop a console-based application which will represent a shapes board with the ability to draw different types of shapes on it. The main idea is that the user has an empty blackboard and can interact with it in a few ways: show the blackboard, place or remove some shape within the blackboard or clear it completely. Additionally, the user should be able to save and load the state of the blackboard from the file.



## Commands

Command	Parameters	Description	Result	Example
draw	-	Draw blackboard to the console	ASCII-drawn blackboard with all shapes on it, or empty	<pre>&gt; draw</pre>
list	-	Print all added shapes with their IDs and parameters	List of all added shapes. One shape per line with shape ID and all information related to the shape	<pre>&gt; list &gt; id circle radius coordinates &gt; id square width height coordinates</pre>
shapes	-	Print a list of all available shapes and parameters for the add command	List of all available shapes. One shape per line with shape info as defined by the student.	<pre>&gt; shapes &gt; circle radius coordinates &gt; square width height coordinates</pre>
add	shape parameters	Add shape to the blackboard	Shape is added to the blackboard	<pre>&gt; add circle 10 10 5</pre>

undo	-	Remove the last added shape from the blackboard	The last added shape is removed	> undo
clear	-	Remove all shapes from the blackboard	All shapes are removed from the blackboard	> clear
save	file-path	Save the blackboard to the file	Blackboard is saved to the file. The state of the blackboard remains unchanged.	> save D:\test.bb
load	file-path	Load a blackboard from the file	Blackboard is loaded from the file. Previous state of the blackboard is cleared.	> load D:\test.bb

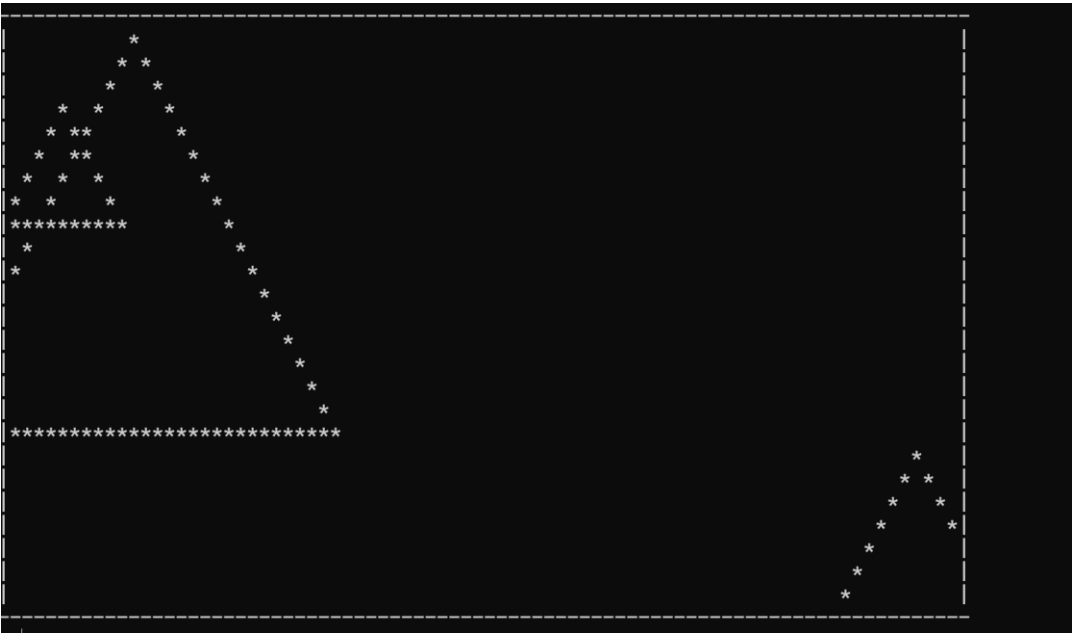
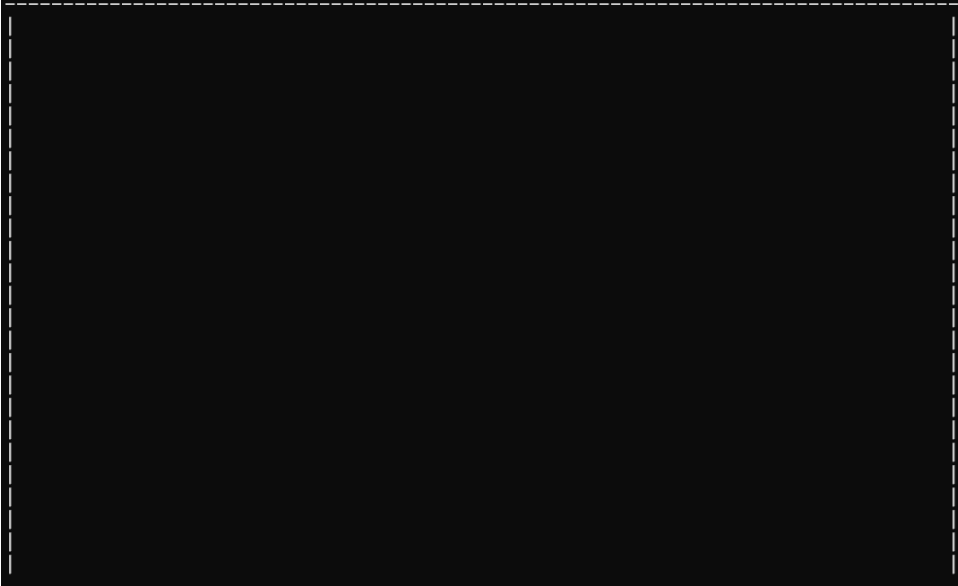
## Constrains:

**There are some constraints and preferences related to that task.**

1. Shapes are drawn as frames.
2. Shapes can overlap. In that case, shapes should be correctly drawn.
3. Shapes cannot be placed outside of the board.
4. Shapes bigger than the board cannot be placed on the board.
5. Shapes that are partially outside of the board can be placed, but they are cropped.
6. Figures with the same type and parameters cannot be placed on the same spot.
7. There is no limit to the shape count on the board.
8. Empty boards should also be correctly saved and loaded.
9. A file with an invalid board should not be loadable.
10. There must be shapes with different parameter count (radius, width\height, height\angle).

# TESTING:

```
7. save <file>
8. load <file>
9. exit
> draw
```



Triangle: 76 18 10,4 3 6 , 10 0 18

Rectangle: 75 0 10 15

Circle: 10 10 10

Square: 10 10 20

[illegible]

```
> add
Enter shape type (circle, rectangle, square, triangle): square
Enter top-left x, y, and size for the square: 10 10 26
The side of the square exceeds the board's height.
square is too big for the board
>
```

## 1.What is inheritance?

```
class Animal {
public:
    void eat() {
        cout << "The animal is eating" << endl;
    }
};

class Cat : public Animal {
public:
    void meow() {
        cout << "The cat says meow'" << endl;
    }
};

Cat myCat;
myCat.eat();
myCat.meow(); |
```

In functional programming, for example, polymorphism can be achieved through higher-order functions or parametric polymorphism. Can be achieved through interfaces or abstract classes

### 3. What is operator overloading?

**Перевантаження операторів** — це коли ти можеш використовувати звичні оператори, такі як +, -, або ==, для своїх власних об'єктів.

### 4. Which types of constructors do you know?

Move, Copy, Default, Constructor with parameters

### 5. What does the virtual keyword mean?

Used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class.

використовується для того, щоб дозволити дочірньому класу **перевизначити** метод, який визначений у батьківському класі. Воно дає можливість викликати правильну версію методу (того класу, об'єкт якого ми використовуємо) навіть якщо ми працюємо через вказівник на батьківський клас.

### 6. What does the override keyword mean?

використовується для того, щоб показати, що метод у дочірньому класі **перевизначає** метод із батьківського класу

```
class Animal {
public:
    virtual void sound() { // Віртуальний метод
        cout << "Some sounds" << endl;
    }
};

class Cat : public Animal {
public:
    void sound() override {
        cout << "The cat says meaw'" << endl;
    }
};
```

## 7. What does the final keyword mean? In which contexts it can be used?

використовується для того, щоб заборонити подальше успадкування класу або перевизначення методів у дочірніх класах. Його можна застосовувати у двох контекстах: до класів та до методів.

```
class Animal final { // Від цього класу не можна успадковуватись
public:
    void sound() {
        cout << "some sounds" << endl;
    }
};

// Наступний код викличе помилку
class Cat : public Animal {
    // Помилка: не можна успадковуватись від 'Animal', оскільки final
};
```

Якщо позначити метод як **final**, це означає, що в жодному дочірньому класі цей метод не можна перевизначити.

```
class Animal {
public:
    virtual void sound() final { // Цей метод не можна перевизначати
        cout << "Some sounds" << endl;
    }
};

class Cat : public Animal {
public:
    // Наступний код викличе помилку
    void sound() override {
        cout << "The cat says meaw!" << endl;
    }
};
```

## 8. What does the pure method mean?

**Відсутність побічних ефектів:** Метод не змінює жодного стану або змінних поза межами свого обсягу. Він працює лише з вхідними даними, які отримує, і не змінює глобальних змінних, статичних полів чи полів екземпляра класу.

Метод завжди **повертає однаковий результат** для однакових вхідних даних

## 9. When do we need virtual functions?

коли ми хочемо реалізувати **поліморфізм** у програмуванні. Це дозволяє використовувати один і той самий інтерфейс для виклику функцій, які реалізовані по-різному в базовому та похідних класах.

## 10. What is dynamic\_cast and how is it implemented under the hood?

Оператор в C++, який використовується для приведення вказівників/посилань на об'єкти одного типу до іншого типу в ієрархії класів. На відміну від інших типів приведення (наприклад, static\_cast), він перевіряє тип об'єкта **під час виконання** (run-time) і використовується для **перетворення типів** в ієрархії спадкування, особливо для приведення від базового класу до похідного або між класами, пов'язаними через загальний базовий клас.

Для роботи з dynamic\_cast у базовому класі має бути одна віртуальна функція.

**Статичне приведення:** це найпростіший тип приведення, який можна використовувати. **Це приведення** під час компіляції. Він виконує такі речі, як неявні перетворення між типами (наприклад, int у float або вказівник на void\*), а також може викликати функції явного перетворення (або неявні).

RTTI- це механізм у C++, який дозволяє дізнатися **тип об'єкта під час виконання програми**. RTTI використовується для роботи з поліморфізмом, коли реальний тип об'єкта може бути визначений тільки під час виконання, а не під час компіляції.

```
#include <iostream>
using namespace std;

// Базовий клас
class Animal {
public:
    virtual void sound() {
        cout << "Some generic animal sound\n";
    }
};

// Похідний клас
class Dog : public Animal {
public:
    void sound() override {
        cout << "Woof woof!\n";
    }
    void wagTail() {
        cout << "The dog is wagging its tail!\n";
    }
};

int main() {
    Animal* animalPtr = new Dog(); // Вказівник на базовий клас, але він вказує на об'єкт
    // Використання dynamic_cast для перевірки, чи animalPtr насправді є об'єктом класу Dog
    Dog* dogPtr = dynamic_cast<Dog*>(animalPtr);

    if (dogPtr) { // Якщо приведення вдалось
        dogPtr->wagTail(); // Викликаємо функцію, яка доступна лише для класу Dog
    } else {
        cout << "This animal is not a dog.\n";
    }

    delete animalPtr;
    return 0;
}
```



### 11. Explain the difference between `unique_ptr` and `shared_ptr`.

Критерій	<code>unique_ptr</code>	<code>shared_ptr</code>
Виключне володіння	Володіє об'єктом ексклюзивно. Лише один вказівник може володіти об'єктом у будь-який момент часу.	Підтримує розділене володіння. Кілька вказівників можуть одночасно посилатися на один об'єкт.
Передача володіння	Передача здійснюється через <code>std::move()</code> . Оригінальний вказівник після передачі стає порожнім.	Копіюється без використання <code>std::move()</code> . Об'єкт може мати кілька копійованих вказівників.
Лічильник посилань	Немає лічильника, оскільки тільки один вказівник володіє об'єктом.	Є лічильник посилань, який збільшується при кожному копіюванні. Об'єкт видаляється, коли лічильник стає нульовим.
Ресурсоемність	Менший за ресурси, оскільки не підтримує лічильник посилань.	Деяка додаткова витрата через підтримку лічильника посилань.
Використання	Використовується для ексклюзивного володіння об'єктом, де не потрібно розділяти об'єкт між частинами коду.	Використовується, коли потрібно забезпечити розділене володіння об'єктом між кількома частинами програми.

### 12. Explain the difference between `shared_ptr` and `weak_ptr`.

Критерій	<code>shared_ptr</code>	<code>weak_ptr</code>
Циклічні посилання	Викликає проблему циклічних посилань, якщо два об'єкти посилаються один на одного за допомогою <code>shared_ptr</code> . Це призводить до витоків пам'яті.	Використовується для уникнення циклічних посилань, не перешкоджаючи видаленню об'єкта.
Лічильник посилань	Підтримує лічильник посилань. Об'єкт видаляється, коли лічильник досягає нуля.	Не збільшує лічильник посилань і не впливає на тривалість життя об'єкта.
Володіння об'єктом	Володіє об'єктом і гарантує його існування до тих пір, поки є хоча б один <code>shared_ptr</code> .	Не володіє об'єктом. Лише перевіряє наявність об'єкта і може тимчасово отримати до нього доступ.
Конвертація	Може перетворюватися на <code>weak_ptr</code> .	Може тимчасово перетворитися на <code>shared_ptr</code> за допомогою методу <code>lock()</code> , якщо об'єкт ще існує.
Використання	Використовується для спільного володіння об'єктом між кількома частинами програми.	Використовується для безпечного доступу до об'єкта без збільшення лічильника посилань, особливо для запобігання циклічним посиланням.

### 13. RAII idiom.

це техніка програмування C++, яка пов'язує життєвий цикл ресурсу, який необхідно отримати перед використанням (виділена купа пам'яті, потік виконання, відкритий сокет, відкритий файл, заблокований м'ютекс, дисковий простір, підключення до бази даних...

