Задача 1 (Вариант А) (20 точки)

Отговор: Принципът **S** е "Single-responsibility" - това означава, че даден клас/функция трябва да имат точно една отговорност. Функцията отговаря за две неща - изчистването на вектора и принтирането му.

Ако е казано кой е принципът **S** от SOLID: 10 точки
Ако е казано, че метода прави две неща: 10 точки

Задача 2 (Вариант А) (20 точки)

Отговор:

```
Hello !
0.14
7
```

Защо ? - Първия print се извиква с аргумент string, затова той се изписва директно, без промени. Вторият print има аргумент Bar, затова се изивиква функцията, която добавя 3.14 към стойността. **Понеже 3.14 е дробно число, то се cast-ва до double, и затова резултата е 0.14**. Третото извикване е с аргумент int, затова към стойността се добавя 10.

Ако е посочен само изхода, без обяснение - 10 точки. Ако е посочен изхода и обяснението (без частта на cast-ването) - 15 точки. Ако е посочено всичко - 20 точки.

Задача 1 (Вариант Б) (20 точки)

O от SOLID означава Open-close principle. т.е. всеки клас трябва да е отворен за разширение, но затворен за промяна. Този принцип не е спазен защото ако добавим нов клас, който наследява Person, ще се наложи да променим класа Serializer.

Задача 2 (Вариант Б) (20 точки)

Програмата ще изведе 123 на конзолата, след което ще запише във файл с името `f.out` стринга This is a very important sentece. След това ще изведе на конзолата you've got mail. Накрая НЯМА да запише This is another sentence във файла, защото файлът вече е затворен.

Задача 3 (80 точки)

```
#include <iostream>
#include <vector>
#include <string>

template <class T>
class ProtectedValue{
public:
    ProtectedValue();
    ProtectedValue(const T& initial_value, const unsigned& initial_code);
    T get_value(const unsigned& code) const
```

```
private:
    T value;
    unsigned security_code;
};

template <class T>
ProtectedValue<T>::ProtectedValue(): value(T()), security_code(-1) {}

template <class T>
ProtectedValue<T>::ProtectedValue(const T& initial_value, const unsigned&
initial_code): value(initial_value), security_code(initial_code) {}

template <class T>
T ProtectedValue<T>::get_value(const unsigned& code) const
{
    if(code == security_code)
    {
        return value;
    }
    else
    {
        throw std::invalid_argument("Security code is wrong");
    }
}

template <class T>
class ProtectedArray{
public:
    ProtectedArray();
    ProtectedArray(const ProtectedArray& from);
    ProtectedArray& operator=(const ProtectedArray& from);
    ~ProtectedArray();

    void add_value(const T& value, const unsigned& code);
    T get_value(const int& index, const unsigned& code) const;
private:

    static void copy_memory(const ProtectedValue<T>* source, ProtectedValue<T>*
destination, const int& size, const int& capacity);
    static void clear_memory(ProtectedValue<T>* source);
    static void resize(ProtectedValue<T>* source, const int& size, const int&
capacity);

    ProtectedValue<T>* array;
    int size;
    int capacity;
};

template<class T>
void ProtectedArray<T>::copy_memory(const ProtectedValue<T>* source,
ProtectedValue<T>* destination, const int& size, const int& capacity)
{
    destination = new ProtectedValue<T>[size];

    for(int i = 0; i < capacity; i++)
    {
        destination[i] = source[i];
    }
```

```
}

template<class T>
void ProtectedArray<T>::clear_memory(ProtectedValue<T>* source)
{
    delete[] source;
}

template<class T>
void ProtectedArray<T>::resize(ProtectedValue<T>* source, const int& size, const
int& capacity)
{
    ProtectedValue<T>* new_array = new ProtectedValue<T>[size * 2];

    for(int i = 0; i < capacity; i++)
    {
        new_array[i] = source[i];
    }

    clear_memory(source);
    source = new_array;
}

template<class T>
ProtectedArray<T>::ProtectedArray(): array(new ProtectedValue<T>[1]), size(1),
capacity(0) {}

template<class T>
ProtectedArray<T>::ProtectedArray(const ProtectedArray<T>& from):
size(from.size), capacity(from.capacity)
{
    copy_memory(from.array, this->array, size, capacity);
}

template<class T>
ProtectedArray<T>& ProtectedArray<T>::operator=(const ProtectedArray<T>& from)
{
    if(this != &from)
    {
        clear_memory(this->array);
        this->capacity = from.capacity;
        this->size = from.size;

        copy_memory(from.array, this->array, size, capacity);
    }
    return *this;
}

template<class T>
ProtectedArray<T>::~ProtectedArray()
{
    clear_memory(this->array);
}

template<class T>
void ProtectedArray<T>::add_value(const T& value, const unsigned& code)
{
    if(this->capacity == this->size)
```

```
    {
        ProtectedArray<T>::resize(this->array, this->size, this->capacity);
    }

    this->array[this->capacity] = ProtectedValue<T>(value, code);
    this->capacity++;
}

template<class T>
T ProtectedArray<T>::get_value(const int& index, const unsigned& code) const
{
    try{
        return this->array[index].get_value(code);
    }
    catch(const std::invalid_argument& ex)
    {
        std::cout << "Code is not valid...";
        return T();
    }
}

int main()
{
    ProtectedArray<int> b;
    b.add_value(123, 1234);
    return 0;
}
```

ProtectedValue

- Използване на темплейти - 15 точки
- Липса на сетъри - 7,5 точки
- Правилен метод `get_value` (логика) - 3,5 точки
- Стил - const &, const методи, използване на инициализиращ списък, използване на `unsigned` - 10 точки

ProtectedArray

- Използване на темплейти - 7 точки
- Голяма четворка - 10 точки (всяко по 2,5т.)
- Динамичен масив - 10 точки
- Правилен логически метод - 7 точки
- Стил - const &, const методи - 5 точки
- Изнесена логика за копиране на динамични масиви, изтриване и resize в отделни функции , използване на `unsigned` - 5 точки

Ако някой от класовете не се компилира, 50% надолу за съответния клас

Задача 4 (80 точки)

```
#include <iostream>
#include <vector>
#include <string>

class VehicleInShop{
```

```
public:
    enum Severity{
        Low,
        Medium,
        High
    };

    VehicleInShop() = default;
    VehicleInShop(const std::string init_maker, const std::string& init_model,
     const unsigned& init_year, const std::string& init_problem, const Severity&
init_severity);

    void set_maker(const std::string& new_maker);
    void set_model(const std::string& new_model);
    void set_year(const unsigned& new_year);
    void set_problem(const std::string& new_problem);
    void set_severity(const Severity& new_severity);

    unsigned get_year() const;
    Severity get_severity() const;
    std::string get_maker() const;
    std::string get_model() const;
    virtual std::string get_problem() const = 0;

    virtual ~VehicleInShop() = default;
protected:
    unsigned year;
    Severity severity;

    std::string maker;
    std::string model;
    std::string problem;
};

VehicleInShop::VehicleInShop(const std::string init_maker, const std::string&
init_model,
     const unsigned& init_year, const std::string& init_problem, const Severity&
init_severity): maker(init_maker), model(init_model),
     year(init_year), problem(init_problem), severity(init_severity) {}

void VehicleInShop::set_maker(const std::string& new_maker)
{
    this->maker = new_maker;
}
void VehicleInShop::set_model(const std::string& new_model)
{
    this->model = new_model;
}
void VehicleInShop::set_year(const unsigned& new_year)
{
    this->year = new_year;
}
void VehicleInShop::set_problem(const std::string& new_problem)
{
    this->problem = new_problem;
}
void VehicleInShop::set_severity(const VehicleInShop::Severity& new_severity)
{
```

```
    this->severity = new_severity;
}

unsigned VehicleInShop::get_year() const
{
    return this->year;
}
VehicleInShop::Severity VehicleInShop::get_severity() const
{
    return this->severity;
}
std::string VehicleInShop::get_maker() const
{
    return this->maker;
}
std::string VehicleInShop::get_model() const
{
    return this->model;
}


class Car: public VehicleInShop{
public:
    Car() = default;
    Car(const std::string init_maker, const std::string& init_model,
     const unsigned& init_year, const std::string& init_problem, const
VehicleInShop::Severity& init_severity, const bool& init_is_car_private);

    void set_is_car_private(const bool& new_is_car_private);
    bool get_is_car_private() const;
    std::string get_problem() const override;
private:
    bool is_car_private;
};

Car::Car(const std::string init_maker, const std::string& init_model,
     const unsigned& init_year, const std::string& init_problem, const
VehicleInShop::Severity& init_severity, const bool& init_is_car_private):
     VehicleInShop(init_maker, init_model, init_year, init_problem,
init_severity), is_car_private(init_is_car_private) {}

void Car::set_is_car_private(const bool& new_is_car_private)
{
    this->is_car_private = new_is_car_private;
}
bool Car::get_is_car_private() const
{
    return this->is_car_private;
}

std::string Car::get_problem() const
{
    return this->problem;
}

class Microbus: public VehicleInShop{
public:
    Microbus() = default;
```

```
    Microbus(const std::string init_maker, const std::string& init_model,
     const unsigned& init_year, const std::string& init_problem, const
VehicleInShop::Severity& init_severity, const bool& init_is_bus_for_passengers);

    void set_is_bus_for_passengers(const bool& new_is_bus_for_passenger);
    bool get_is_bus_for_passengers() const;
    std::string get_problem() const override;
private:
    bool is_bus_for_passengers;
};

Microbus::Microbus(const std::string init_maker, const std::string& init_model,
     const unsigned& init_year, const std::string& init_problem, const
VehicleInShop::Severity& init_severity, const bool& init_is_bus_for_passengers):
     VehicleInShop(init_maker, init_model, init_year, init_problem,
init_severity), is_bus_for_passengers(is_bus_for_passengers) {}

void Microbus::set_is_bus_for_passengers(const bool& new_is_bus_for_passenger)
{
    this->is_bus_for_passengers = new_is_bus_for_passenger;
}
bool Microbus::get_is_bus_for_passengers() const
{
    return this->is_bus_for_passengers;
}

std::string Microbus::get_problem() const
{
    return this->problem;
}
class Autoshop
{
public:
    static Autoshop* get_instance();

    void add_to_shop(const Car& new_car);
    void add_to_shop(const Microbus& new_microbus);
    bool is_shop_full() const;
private:
    static const unsigned MAX_CAPACITY = 10;
    static Autoshop* instance;

    std::vector<VehicleInShop*> vehicles_in_shop;

    Autoshop() = default;
    ~Autoshop();
};

Autoshop* Autoshop::get_instance()
{
    return Autoshop::instance;
}

void Autoshop::add_to_shop(const Car& new_car)
{
    this->vehicles_in_shop.push_back(new Car(new_car));
}
void Autoshop::add_to_shop(const Microbus& new_microbus)
```

```
{
    this->vehicles_in_shop.push_back(new Microbus(new_microbus));
}
bool Autoshop::is_shop_full() const
{
    unsigned current_capacity = 0;

    for(auto* vehicle: this->vehicles_in_shop)
    {
        current_capacity += vehicle->get_severity();
    }

    return current_capacity >= Autoshop::MAX_CAPACITY;
}
Autoshop::~Autoshop()
{
    for(auto* vehicle: this->vehicles_in_shop)
    {
        delete vehicle;
    }
    delete instance;
}

Autoshop* Autoshop::instance = new Autoshop();

int main()
{
    return 0;
}
```

Клас `VehicleInShop` - 10 точки (ако го няма, 0 точки)

Клас `Car` и `Microbus` - 5 точки (2 по 2,5т)

Клас `Autoshop` - логика - 5 точки

Хетерогнен контейнер в `Autoshop` (вектор от указатели към базов клас - 5точки, копиране в динамичната памет - 5 точки, виртуален деструктор - 5 точки) - 15 точки

Стил - const&, const методи, unsigned при годината, enum за сериозност на проблема - 5 точки

Singleton - 5 точки

Ако някой от класовете не се компилира, 50% надолу за съответния клас