OOP - Object Oriented Programming

1. OOP is a programming paradigm based on the concept of objects. The main point of OOP is creating objects based on class. A common feature of objects is that methods are attached to them and can access and modify the object's data fields.

2. Methods are functions of class that allow us to communicate with class fields.

3. Fields are attributes that describe a class.

4. Say principles of OOP

Inheritance
When you describe a new class, you can describe it based on the existing class. Subclass will inheritance base class properties and we can reimplement properties of base class

Encapsulation
All properties or fields of class need to be private, and access needs to be via public methods. An object need to control their properties.

Polymorphism
There are two types of polymorphism:
Dynamic - when the base class contains virtual functions that can be re-implemented in its subclasses. When we call such a function in a subclass, its behavior may differ from calling it in the base class. Virtual. Compiler choose which method call at the run-time

Static - function reimplementation by their signature and templates. For each usage will generate a new function. Overloading. Compiler choose which method call at the compile-time

Classes

1. What is class and object in C++?

Class is data type. Object is an instance of class.

2. What are the classes?

Class is like a blueprint of an object. And you can create an object based on this blueprint.

3. What is the difference between struct and class?

In structure all fields default are public. In class are private

4. Explain constructor in C++?

Constructor in C++ are special methods that are used to allocate memory for objects. A constructor is automatically called when an object is created.

5. Explain destructors in C++?

Destructors in C++ are special methods that are used to deallocate memory of objects. A destructor is automatically called when an object is deleted.

6. What is a copy constructor in C++?

In C++, a copy constructor is a special member function that is used to create a new object as a copy of an existing object.
MyClass(const MyClass &source);

7. What is an abstract class and when do you use it?

Abstract class is class that will never have an object, all methods are pure virtual

8. Types of class inheritance?

There are 3 types: public, private, protected.
Public: all public class members in base class are public in subclass
Protected: all public and protected class members become protected in subclass
Private: all members of base class become private in subclass

9. Solve this problem.

```
class B{};
class C : protected B{};
void some(){
        B *b = new C; // ERROR
}
```

1. We can use friend function:

```
class B{};
class C : protected B{friend void some();};
void some(){
        B *b = new C; // OK
}
```

2. We can place this function into class C

```
class B{};
class C : protected B{
        void some(){
                B *b = new C;
        }
};
```

10. **Friend** class and friend function?

Friend function is function that is not a class member but has access to private fields of class
Friend class is the same thing. Friend class has an access to private fields of another class

```
class A{friend void friend_function(MyClass &obj);};
```

11. How to **prohibit class inheritance**?

Use keyword "final": class A final {};

12. Virtual inheritance

Virtual inheritance allows us to solve the "diamond problem". Keyword "virtual" means that there should be only one instance of the base class A in the hierarchy.

13. Diamond problem. How to solve it? <u>And define</u>:

```
class A { int h; };
class B : A {};
class C : A {};
class D : B, C {};
int main() {
        D d;
        d.h; - error
}
```

Two ways to solve this problem:
1. Use virtual inheritance

```
class B : virtual A {};
class C : virtual A {};
class D : B, C {};
```

```
int main() {
	D d;
	d.h;
}
```
In this case class A will be initialized one time in subclass D
2.
```
int main() {
	D d;
	d.B::h;
}
```

14. Tell me about **virtual function**

Virtual function allows us to override this function in subclass.

15. What is **override**?

Overriding is when we have one method realization in one(base class) class and we can reimplement this method in another class(subclass).

16. What is a virtual destructor?

The primary purpose of having a virtual destructor is to ensure proper destruction of derived class objects when they are deleted through a pointer to the base class.

```
class A{virtual ~A(){}};
class B : A {~B() override{}}
int main(){
	A *a = new B;
	delete a;
}
```

17. What is **overloading**?

Overloading when multiple functions have the same name but different implementations.

18. What is operator overloading?

This operator overloading allows to rewrite behavior of operator like a equal, divide, plus, minus

19. Is destructor overloading possible? If yes then explain and if no then why?

No, you can't overload the destructor.

20. Exception from constructor

If we throw an exception from the constructor, a memory leak will happen. Because after exception memory isn't freed. To solve this problem we can use smart pointers

```
std::unique_ptr<MyClass> ptr(new MyClass());
```

If the constructor throw an exception, the smart pointer automatically frees memory.

21. Keyword **static**

Keyword "static" allows us to set field visibility only for that .cpp file where this is implemented.
Analog of "static" is "anonymous namespace"

22. What is **static function?**

Static member functions can access only static members (variables or functions) of the class. It can be called on the class itself, without creating an instance of the class.

23. How to protect class from being copied?

We can declare a private copy constructor and copy assignment operator.

```
class A{
private:
	A(const A&);
	A& operator=(const A&);
};
int main(){
	A a1;
	A a2 = a1; // ERROR
	A a3;
	a3 = a1; //ERROR
}
```

24. What is a **pure virtual method**?

Pure virtual method is a method is a virtual function declared in a base class that has no realization in the base class and must be realized in any subclass.

```
class A{
	virtual void some() = 0;
};
class B{
	void some() override{
		std::cout << "B" << std::endl;
	}
};
```

but there is one problem. When we try to call a pure virtual method from the constructor of the base class we will receive an exception.

```
class A{
	A(){
		some();
	}
	virtual void some() = 0;
};
class B{
	void some() override{
		std::cout << "B" << std::endl;
	}
};
int main{
	A *a = new a; // COMPILATION ERROR
}
```

Because the compiler thinks that function some() does not exist. To solve this problem we can call this function form non pure virtual function:

```
class A{
	A(){
		foo();
	}
	void foo(){
		some();
	}
	virtual void some() = 0;
};
```

```
class B{
	void some() override{
		std::cout << "B" << std::endl;
	}
};
int main{
	A *a = new a; // OK
}
```

25. Size of the empty class is 1 byte. Size of the empty subclass of empty base class also is 1 byte.(optimization)

26. Keyword **explicit**

Functions

1. What is **Template?**

Template allows you to define a function/class with a placeholder for the type of its parameters.

```
template <typename T>
class A{
	T field;
};

template <typename T>
void some(T a, T b){}

int main(){
	A<int> a;
	some(12, 11); // int
}
```

2. What is the **constant methode**?

A constant method is a function that cannot modify the class members. This is because, for this function, the object is treated as constant. However, a constant method can modify certain class members using the "mutable" keyword.

```
void constMethode() const{}
```

3. What is the keyword "**mutable**"?

This keyword allows you to mark class members as modifiable within constant methods.

```
mutable int a;
```

4. **Constant arguments** in function

We use constant arguments in function when we don't want to change arguments.

STL(Standard Templates Library)

1. **enum** is a user data type that allows to create named constants of ints.

```
enum Color {
  RED,    // 0
  GREEN,  // 1
  BLUE    // 2
};

Color myColor = GREEN;
if (myColor == RED) {
	std::cout << "Цвет: Красный" << std::endl;
```

} else if (myColor == GREEN)

We can use enum when we have certain range of arguments

2. **Vector**

Vector is a standard container in STL.
Vector is a dynamic array.
std::vector<TYPE> NAME = {1, 2, 3, 4};

Methodes:
NAME.push_back();
NAME.pop_back();
NAME.insert(NAME.begin(), ELEMENT);
NAME.erase(NAME.begin());
NAME.erase(NAME.begin() + 2, NAME.begin() + 3);
NAME.size();
NAME.resize(SIZE); // if SIZE < NAME.size() elements out of range will be deleted
NAME.reserve(SIZE); // just reserve memory for future elements

If we will add 100 elements by push_back; each loop will be memory reallocation to 2 times of size of vector

3. What is .**end()**;

The .end() function or method that point at the one past element of the container.

4. **Sort**

std::sort(NAME.begin(), NAME.end()); from smallest to biggest
std::sort(NAME.rbegin(), NAME.rend()); from biggest to smallest

5. **Find**

std::find(startIterator, endIterator, value);

6. What are associative containers in C++?

```
std::map<int, std::string> myMap;
   myMap[1] = "One";
std::unordered_map<int, std::string> myUnorderedMap;
   myUnorderedMap[1] = "One"
std::set<int> mySet;
std::unordered_set<int> myUnorderedSet;
   myUnorderedSet.insert(3);
   mySet.insert(3);
```

7. What is the difference between std::map and std::unordered_map?

std::map is realized by a binary tree and this tree is sorted.
std::map is realized by hash-table and it's not sorted

8. If we want to use class in std::map we need to implement less operator

9. Other containers?

vector, list, forward_list, queue, stack

10. What are the differences between queue and stack?

Queue if FIFO, stack is LIFO
FIFO - First In First Out

LIFO - Last In First Out

11. How to use stack and queue

```
std::stack<DataType> myStack;
myStack.push();
myStack.pop();
myStack.top();
myStack.empty();

std::queue<DataType> myQueue;
myQueue.front();
myQueue.back();
```

12. Why are STL algorithms better than hand-written algorithms?

Because STL algorithms are tested by thousands of developers.
They are optimized.
They are ready to work with different arguments in different conditions.

13. std::endl; is a predefined object of ostream class to insert new line characters.

14. Include all STL libraries <bits/stdc++.h>

15. What is the difference between an array and a list in C++?

Array has a fixed size, list has a variable size.
An array supports random access using indexes, list has access via iterators.

Stages of compilation process

1. Preprocessing: the preprocessor handles preprocessor directives such as #include and #define. It essentially prepares the source code for compilation by resolving macros and including necessary header files.

Directives: #include #define #ifndef #endif #if

2. Compilation: This stage involves syntax checking and the generation of object files(binary code).

3. Linking: The linking stage combines object code with library code to create an executable program.

4. Linker links object files into executable file.

5. How work #include?

Algorithms

1. Bubble sort - Big O of N SQUARED:

```
void bubbleSort(std::vector<int> &A){
  for(int i = 0; i < A.size(); i++){
     for(int j = 0; j < A.size() - i - 1; j++){
        if(A[j] > A[j + 1]){
           std::swap(A[j], A[j + 1]);
        }
     }
  }
}
```

2. Selection sort - Big O of N SQUARED:

```
void selectionSort(std::vector<int> &A){
   for(int i = 0; i < A.size(); i++){
      auto *minimalElement = &A[i];
      for(int j = i; j < A.size(); j++){
         if(A[j] < *minimalElement){
            minimalElement = &A[j];
         }
      }
      std::swap(A[i], *minimalElement);
   }
}
```

3. Random number

```
      int max=100, min=54,i;
  int range = max - min + 1;
  for (i=min; i<max;i++)
   {
      int num = rand() % range + min;
      cout<<num;
   }
```

Algorithm complexity

For counting of algorithm complexity we use Big O notation.
1. It's machine independent
2. We can count time & space efficiency
Types of measurement:
1. Worst case
2. Best case
3. Average case
O(1) - Big O of ONE - const
O(N) - Big O of N - linear
O(N^2) - Big O of N SQUARED - quadratic

Multithreading

1. What is multithreading?

Multithreading is the process of running two or more tasks concurrently to improve the performance of a program.

```
std::thread myThread(myFunction, ARG);
```

2. What the difference between .join() and .detach()

.join() use to wait for a thread to finish its execution
.detach() use to allows few threads work independently

3. What is mutex, how to use it?

Mutex is a synchronization primitive used to protect shared resources between a few threads from race condition.
std::lock_guard<std::mutex> lock(myMutex);
sharedData++;

4. What is race condition?

Race condition is when we have few threads and they try to use shared resources at one time. Resulting in unpredictable behavior(nepředvídatelné chování)

Other

1. The size of pointer depends on the system word size. On a 32-bit PC sizeof pointer is 32 bits (4 bytes), while on 64 bit PC it's 8 byte.

2. What are **pointers**?

Pointer is data type that store the memory address of another variable.

3. What is the namespace in C++?

A namespace allows group under one name classes, objections, and functions

4. getline(cin, c); input string with spaces

5. **Rule of Five**

Before standard C++11 was Rule of Three :
If we implement one of the following three functions we need to implement all three:
1. Destructor: ~ClassName()
2. Copy Constructor: ClassName(const ClassName&)
3. Copy Assignment Operator: ClassName& operator=(const ClassName&)
After after appearance of "move semantic" in C++11 Rule of Three become Rule of Five, were added two more definitions:
1. Move Constructor: ClassName(ClassName&&)
2. Move Assignment Operator: ClassName& operator=(const ClassName&)

6. Tell about "**move semantic**"

Move semantic was introduced in C++11. When you assigned or passed objects by value, a deeply copy of object's contents is created. This process could be expensive, especially for large objects. Move semantic provides a way to transfer resources more efficiently, often without the need for copying.
When we use move semantics, a new object uses a memory site that has previously used another object. We use move when we understand that we will not use old object in the future.

7. What is deep copy

Deep copy is a process of copying objects byte by byte.

8. What are the **base types** in C++

int, float, char, long, double, bool, unsigned types

9. Dynamic memory allocation

We can dynamically allocate memory if we need to manually control the object's life time.
We can dynamically allocate memory using command new. We can delete object using command delete;

10. What is **rvalue**

Rvalue is a temporary value that doesn't necessarily have a named memory location

11. What is **lvalue**

An lvalue refers to an object that occupies some identifiable location in memory

12. How to cast variable

std::static_cast<int>(x);
(int)x;
std::dynamic_cast<ClassB*>instanceOfClassA;

13. How to cast int to std::string

```
int m = 123123;
std:string str = std::to_string(m);
```

14. What is the difference between dynamic and static memory allocation?

When we allocate memory staticy, it happens at the compilation.
When we allocate dynamically, it happens while the program is working.
In dynamic allocation we can change the size of allocated memory. And we need to delete this dynamically allocated object.
Also exists operator std::placement_new. This operator allows the creation of objects into concrete addresses in memory.

15. Dynamically allocated array

```
int* a =  new int[row * col];
```

16. What do you mean by call by value and call by reference?

Call by value means to send a value and make a copy of it and work with the copy. Call by reference means to send a address of value and work directly with this value, not with the copy

17. Smart pointers.

Smart pointers in C++ are objects that act as pointers but provide additional features to manage the memory they point to.
In C++ there are few smart pointers:
1. std::unique_ptr<int> uniquePtr = std::make_unique<int>(42); - represents exclusive ownership of a dynamically allocated object

```
int* rawPtr = uniquePtr.release();
```

2. std::shared_ptr<int> sharedPtr = std::make_shared<int>(42) - allows multiple smart pointers to share ownership of the same dynamically allocated object.

```
sharedPtr.use_count();
sharedPtr.unique(); //bool
```

3. std::weak_ptr;

QT

1. What is Frameworks?

Frameworks are platforms that allow developers to create a variety of software applications.

2. What is QT?

Qt is a cross-platform framework that allows its users to develop interfaces.

3. What is QML?

QML is a declarative programming language for Qt. QML is an abbreviation for Qt modeling language. QML enables developers and designers to create dynamic and visually appealing applications.

4. What is declarative?

Declarative when we describe the result, not the algorithm.

5. What is QtWidget?

Widgets are the basic elements that allow users to create user interfaces within the Qt framework. They are able to display information and receive user input.

6. How signals and slots work in Qt?

In Qt, signals and slots are a mechanism for communication between objects. Signals are emitted when a particular event occurs, and slots are functions that can be connected to those signals to respond to the event.

How do you parse JSON?

I use JsonCPP library for parse Json response.

```
nlohmann::json jsonData = nlohmann::json::parse(response);
float c = jsonData["c"];
```

How do you send CURL requests?

```
CURL *curl;
std::string readBuffer;

curl = curl_easy_init();
if(curl) {
   curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
   curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, writeCallback);
   curl_easy_setopt(curl, CURLOPT_WRITEDATA, &readBuffer);
   curl_easy_perform(curl);

   curl_easy_cleanup(curl);

   return readBuffer;
}
```

Tasks

1. Merge two sorted arrays

```
int main(){
   std::vector<int> A1 = {1, 3, 4, 6, 8, 20};
   std::vector<int> A2 = {1, 2, 5, 7, 8, 9, 21};

   std::vector<int> output;

   size_t counter1 = 0;
   size_t counter2 = 0; // or size_t counter2{};

   while(counter1 < A1.size() && counter2 < A2.size()){
      if(A1[counter1] <= A2[counter2]){
         output.push_back(A1[counter1]); // or  output.push_back(A1[counter1++]);
         counter1++;
      }else{
         output.push_back(A2[counter2]);
         counter2++;
      }
   }

   while(counter1 < A1.size()){
      output.push_back(A1[counter1++]);
   }

   while(counter2 < A2.size()){
```

```
        output.push_back(A2[counter2++]);
    }}
```

or

```
if(A1[i] <= A2[i]){
            output.push_back(A1[i]);
            output.push_back(A2[i]);
        }else{
            output.push_back(A2[i]);
            output.push_back(A1[i]);
        }
```

2. Delete element in vector

```
std::vector<int> A = {1, 2, 3, 7, -1, 1, 10, -222};

    int target = 1;

    int k = 0;
    for(int i = 0; i < A.size(); i++){
        if(A[i] != target){
            A[k] = A[i];
            k++;
        }
    }
```

or

```
    for(int i = 0; i < A.size(); i++){
        if(A[i] == target){
            A.erase(A.begin() + i);
        }
    }
```