Seminar 06
Composition, Friends, Static, Design Pattern Ideas and more

1. Const methods.

- Every method that does **NOT** change the object should be declared as **const**.
- Example in the header file:
```
void print() const;
```
- Example in the source file:
```
void ClassName::print() const
{
    std::cout << <member>;
}
```

2. Initializer list.

- Used for initializing const members or any member that needs to be initialized before the body of the constructor.
- Separated by commas, used like a constructor for the members.
- Other constructors can be called using the initializer list.
- *Note: The order in which members are initialized is the order in which they are declared in the class and **NOT** the order they are stated in the initializer list!*
- Example:
```
Person::Person() : age(0), name(nullptr)
{ … }
```
- Example2:
```
Person::Person(int age, const char* name) : Person()
{ … }
```

3. Parameterized constructor with default parameters.

- Sometimes instead of defining a default constructor and a parameterized constructor, we can just define a parameterized constructor and use default values for the parameters.
- Example:
```
Person::Person(int age = 0, const char* name = "")
{
    setAge(age);
    setName(name);
}
```

4. Resource Acquisition Is Initialization.

- From cppreference.com:

  *Resource Acquisition Is Initialization or **RAII**, is a C++ programming technique which binds the life cycle of a resource that must be acquired before use (allocated heap memory, thread of execution, open socket, open file, locked mutex, disk space, database connection - anything that exists in limited supply) to the lifetime of an object.*

  ***RAII** can be summarized as follows:*

  - ***encapsulate** each resource into a **class**, where:*
    - *the constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done,*
    - *the destructor releases the resource and never throws exceptions;*
  - ***always** use the resource via an instance (object) of a **RAII-class** that either:*
    - *has automatic storage duration or temporary lifetime itself, or*
    - *has lifetime that is bounded by the lifetime of an automatic or temporary object;*

5. Composition.

- Composition is the design technique in object-oriented programming to implement **"has-a"** relation between objects.
- Example: A car **has an** engine, **has a** make.

```
class Car {
    public:
        // ...
    private:
        Engine engine;
        String make;
        int yearManufactured;
};
```

6. Friend classes and friend functions.

- Classes and functions can be declared as friends to a given class.
- Friends of the class can access all private data members and methods.
- `friend class <class-name>;`
- `friend <function declaration/definition>;`
- `friend <other class's method declaration>;`

7. Static members and methods.

- Members and methods marked as static are linked to the **class** and not to an object of that class. And thus, are accessed by specifying the class first, followed by :: and then the name of the member/method. *(ClassName::member)*
- We can think of static members as global variables for the class.
  Example:

Person.h

```cpp
class Person
{
public:
    static int publicMember;
    static const int MAX_SOMETHING;
    Person();
    static int getNumOfPeople() { return Person::numOfPeople; }
private:
    static int numOfPeople;    // Used as people counter
};
```

Person.cpp

```cpp
#include “Person.h”

int Person::publicMember = 42; // Static data members’ declaration
int Person::numOfPeople  = 0;  // goes in the source file (or directly
const int Person::MAX_SOMETHING = 1337; // in the header file)

Person::Person()
{
    numOfPeople++;
}
```

Source.cpp

```cpp
#include “Person.h”
int main()
{
    Person p1;
    std::cout << Person::getNumOfPeople();  // 1
    Person p2;
    std::cout << Person::getNumOfPeople();  // 2
    Person p3;
    Person p4;
    Person p5;
    std::cout << Person::getNumOfPeople();  // 5
    // This static member cannot be accessed since it's private
    ~~std::cout << Person::numOfPeople;~~

    // But this static member is public, so it can be accessed
    std::cout << Person::publicMember;

    // As well as changed
    Person::publicMember = 5;

    return 0;
}
```

8. = default and = delete.

- = default marks a constructor, destructor or operator= to be defined by the compiler using the **default implementation** (shallow copy or default (zero) values).
  Example:
  ```
  ClassName() = default;
  ```

- = delete marks a constructor or operator= as **non-existent** for this class.
  i.e. this class will not have the marked constructor/operator=.
  Example:
  ```
  // This prohibits the class from being copied
  ClassName(const ClassName& other) = delete;
  ClassName& operator=(const ClassName& other) = delete;
  ```

9. Design patterns.

- From geeksforgeeks.org:

  Software design patterns are general, reusable solutions to common problems that arise during the design and development of software. They represent best practices for solving certain types of problems and provide a way for developers to communicate about effective design solutions.

- Singleton
  - Creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code.

- Creates a “global” state that is capsulated unlike global variables.

- Not proclaimed as “best practice”, as it’s still a global state.

- There are multiple ways to achieve this design pattern, but the essence is:
  1. Create a hidden static instance of the Singleton class.
  2. Provide a public static method that gives access to the static instance.
  3. Forbid copying of that class.