**Laboratory lesson №7**

(The spring session)

**Global variables in OOP: static class members.**

**Logger design template**

***Static and global variables.***

***Global variables*** are variables that were created outside of any block of code (global or file scope), they are accessible from anywhere in the program and stored in memory until the program terminates. Usually, global variables are declared after the ***#include*** block, but above any other code, for example:

#include <iostream>

int g\_x; //global variable g\_x

const int g\_y(3); // global variable g\_y

void doSomething()

{

// global variable can be used in any part of program

g\_x = 4;

std::cout << g\_y << "\n";

}

int main()

{

doSomething();

g\_x = 7;

std::cout << g\_y << "\n";

return 0;

}

**Local variables** are declared inside a block limited by curly braces and are visible only inside this block. A local variable in some block of code always overrides a global one that has the same name. In order to forcefully indicate that a global variable should be used at a given location in the block, the context solution *operator "::"* is used:

#include <iostream>

int value(4); // global variable

int main()

{

int value = 8; // local variable overrides a global one

// value=8

value++; // local variable increase

::value--; // global variable reduction using ::

std::cout << "Global value: " << ::value << "\n";

std::cout << "Local value: " << value << "\n";

return 0;

}

// local variable increase, the value of the global variable does not change, value=3

The use of global variables can significantly reduce the amount of program code, but at the same time, any functions can change the values of global variables, therefore, in complex projects, the use of such variables should be avoided, as this can lead to unpredictable changes in the values of such variables.

A compromise solution for providing global access to variables is to use ***static variables***, which are also globally accessible, but each such variable is only available in one instance in memory***. Static variables*** are similar to global ones in memory allocation, but ***static*** and ***public*** and ***private*** access specifiers act additionally.

***Static fields*** are most often used in the following situations:

• the need to control the total number of class objects;

• creating a single global variable to which all objects of the class have access.

***Static fields*** are defined by word ***static***, which can be used for both attributes and class methods. A feature of elements to which the static keyword is that they belong to a class and not an object of this class, therefore they can be used even without creating an object of the class, and with lot of created objects of this class. Only one copy of static element will be in memory.

Thus, only one copy of each static variable is stored in memory, and the number of copies of non-static fields will be equal to the total number of class objects.

The ***static*** keyword is used before the data type or methods.

Access to static variables or functions occurs using the class name and the context operator "::":

*Class\_name:: attribute\_name*

Static class attributes are provided in the class declaration (or in class header files), and are initialized outside the declaration block in the global scope. You cannot initialize static variables in the body of a class and in its methods. Thus, initialization is needed in order to avoid reinitializing the static variable.

The example of using ***static variable*** to calculate the total number of created class objects (file "main.cpp"):

#include <iostream>

using namespace std;

class X

{

static int n; //class variable counter

static char ClassName[30];

public:

static int getN() { return n; }

static char\* getClass() { return ClassName; }

X() { n++; } // constructor

};

int X::n = 0; // initialization of a private attribute outside the class body

char X::ClassName[] = "My Class";

int main()

{

X a, b, c; // declare three class objects X

cout << X::getN() << " objects of Class \"" << X::getClass() << "\"" << endl;

// class method call using ::

//cout << X::n << endl;

return 0;

}

***Template " Singleton "***. Design template “***Singleton***” is another example of managing of class objects creation using static variables and methods. It makes possible to create only one object of this class:

// Файл Singleton.h

class Singleton

{

private: // The only class instance and base constructors are declared in the //private section of the class.

static Singleton\* instance;

Singleton() {}

Singleton(const Singleton&);

Singleton& operator=(const Singleton&);

public:

static Singleton\* getInstance()

{

if(!instance)

instance = new Singleton();

return instance;

}

};

// Файл Singleton.cpp

#include “Singleton.h”

Singleton\* Singleton::instance = 0; // object initialization

// Файл main.cpp

#include “Singleton.h”

int main()

{

Singleton\* s = Singleton::getInstange(); // Get a pointer to a single //instance of the Singleton class

return 0;

}

**Laboratory work №7.**

**Task progress:**

1. For a class from **Laboratory work 3**, define a field that should be global. Also add a static field to count the number of instances of the specified class. The required field (non-global and non-static) must be the attribute "nameObject".
2. Create UML class diagram.
3. Create the “**Logger**” class based on the "**Singleton**" template:

|  |
| --- |
| Logger |
| - *instance: Logger\**  + log: string[] |
| - *necessary constructors and operators*  + getInstance(): Logger\*  + addRecord(obj: pointer on class object )  + saveLog() |

* The task of class **Logger** is to enter the correct value of your class public fields to the log field by use of addRecord() method. For example:

“object1Name: name

time: YY.MM.DD HH:MM:SS

object1Field1: field1Value

object1Field1: field2Value”

* saveLog() method will be called at the end of the program and save a text file with the name "log.txt" (date and time of creation). For example:

“ClassName: numberOfEntities

object1Name: name

time: YY.MM.DD HH:MM:SS

object1Field1: field1Value

object1Field2: field2Value

...

object2Name: name

time: YY.MM.DD HH:MM:SS

object2Field1: field1Value

object2Field2: field2Value

...“

4. Write a program in which the user will be able to manipulate a specific class and log them using the Logger class.

**The Variants.**

1. The prism

2. The cylinder.

3. The scope.