Laboratory lesson №3
(The spring session)

Classes and objects. Abstraction and encapsulation. The basics of UML for presenting classes. Class in C ++

Purpose of work: to study the basic concepts of object-oriented programming and the basics of working with classes in C ++.

Work task: description of a given concept in the form of a class with the definition of the necessary attributes and methods, its presentation in UML notation, software implementation of the class.

Theoretical information

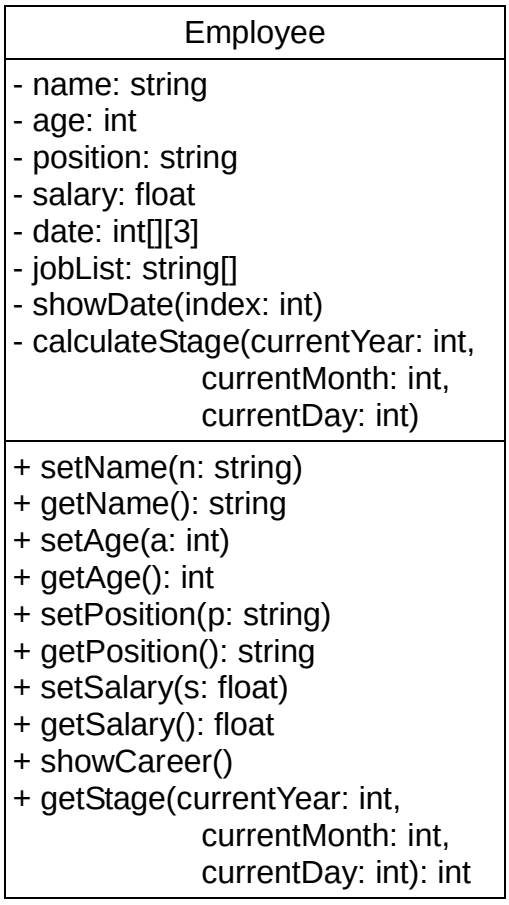
To create a program, using an object-oriented approach, you must:
1. To make up a dictionary of the subject area, which defines the minimum necessary concepts (attributes) for subsequent analysis. This stage is called abstraction; all future work of the developed system depends on its result.
2. To determine what methods are needed for this class to work correctly.
3. To Determine which of the attributes and methods will be internal (**private**) for a given class, and which will determine the interaction of the class with the external environment (other classes, the program in general, etc.) - **public**. As a rule, all class attributes are made internal, and to access them and their changes are created special methods: **getters** and **setters**, which can be called from other places in the program. Class methods, on the contrary, are made available to other program components, with the exception of specific methods necessary for calculating internal or auxiliary variables.
4. To define data types for methods parameters and value types, returned by methods.

Lets look at these steps with the example of the class "Worker company":
1. Abstraction. The main attributes of the class "Worker" from the point of view of the employer, for example, may be the following:
• name of the worker;
• age of the worker;
• position;
• salary of the worker;
• career history of a worker in this company with job titles and job start dates.
2. The following methods may be necessary:
• getters and setters for the attributes specified in p.1 (abstraction).
• a method for displaying career history;
• calculation method and getter for employee's work experience in a company;
• date display method.
3. All attributes are set as internal without external access.
All getters, setters, methods for salary calculating and displaying career history are set **public**.
Methods for showing dates and work experience calculating seniority are **private**

4. Define data types for attributes and methods:

• worker name: **string**;
• age of the worker: **integer**;
• position: **line**;
• worker wages: **actual number**;
• experience in the company: **integer;**
• career history**: array of strings;**
• start dates of working**: two-dimensional array of integers**;
• getters to return a variable of the corresponding type and not accept any parameters, except for the getter name and position, which will accept references to the corresponding variable;
• the position setter additionally accepts the year, month and day of admission to the position. The method of he work experience calculating is automatically called up;
• a career history method displays all elements of an array of work positions with corresponding dates;
• the method of calculating the experience checks if the current date is the date of the last position, if they are different, then the value of the experience is recalculated, which is then returned from the method;
• all setters return nothing, accept a variable of the appropriate type.

One way to describe and represent of object-oriented programs in general, and classes in particular, is to use UML notation (see the application below). Here are the representations of the class being developed in UML:



Classes in C ++. Classes are declared using the class keyword, which determines the appearance of a new type that will combine data and methods with each other. A class represents a logical abstraction and defines the structure of a certain subject area, while an object, a concrete instance, a variable of this class, which implements such class.
The class declaration in C ++ is like structure declaration:

```
class ім'я_класу {
        приватні атрибути та методи;
специфікатор доступу: //public, private або protected
        атрибути та методи;
специфікатор доступу:
        атрибути та методи;
...
    } список_об'єктів_класу;
```

Access specifiers can stand in random order, the scope of one specifier extends to the next specifier or to the end of the class declaration.

For convenience of further support and perception of the object-oriented code of the program, class declaration and its implementation should be separated; The class declaration should be provided in header files (.h, .hpp), and implementation in source files (.cpp).
Here is an example of a code for the declaration and implementation of the "Work" class:

• Header file "Employee.h"

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include <cstring>

class Employee
{
   char name[25];
   int age;
   char position[25];
   float salary;
   int stage;
   static const int maxlen = 255;
   int date[maxlen][3];
   std::string jobList[maxlen];
   int job_index = 0;
   void showDate(int index);
   void calculateStage(int currentYear, int currentMonth, int currentDay);

   public:
      void setName(char *n);
      void getName(char *n);
```

```
        void setAge(int s);
        int getAge();
        void setPosition(char *p, int year, int month, int day);
        void getPosition(char *p);
        void setSalary(float s);
        float getSalary();
        void showCareer();
        int getStage(int currentYear, int currentMonth, int currentDay);
};

#endif // EMPLOYEE_H
```

- File with class implementation «Employee.cpp»

```
#include "Employee.h"

void Employee::setName(char* n)
{
    strcpy(name, n);
}

void Employee::getName(char* n)
{
    strcpy(n, name);
}


void Employee::setAge(int s)
{
    age = s;
}

int Employee::getAge()
{
    return age;
}

void Employee::setPosition(char* p, int year, int month, int day)
{
    strcpy(position, p);
    date[job_index][0] = year;
    date[job_index][1] = month;
    date[job_index][2] = day;
    jobList[job_index] = position;
    calculateStage(year, month, day);
    job_index++;
}

void Employee::getPosition(char* p)
{
    strcpy(p, position);
}
```

```
float Employee::getSalary()
{
    return salary;
}

void Employee::setSalary(float s)
{
    salary = s;
}

void Employee::showCareer()
{
    for(int i=0; i<job_index; i++)
    {
        std::cout << "Job " << i+1 << ": " << jobList[i] << ", started from: ";
        showDate(i);
    }
}

void Employee::showDate(int index)
{
    std::cout << date[index][0] << '.' << date[index][1] << '.' << date[index][2] << std::endl;
}

void Employee::calculateStage(int currentYear, int currentMonth, int currentDay)
{
    if(job_index == 0)
        stage = 0;
    else
    {
        int daysDelta = currentDay - date[0][2];
        if(daysDelta < 0)
            currentMonth -= 1;
        int monthsDelta = currentMonth - date[0][1];
        if(monthsDelta < 0)
            currentYear -= 1;
        stage = currentYear - date[0][0];
    }
}

int Employee::getStage(int currentYear, int currentMonth, int currentDay)
{
    if((currentYear != date[job_index][0]) || (currentMonth != date[job_index][1]) ||
(currentDay != date[job_index][2]))
        calculateStage(currentYear, currentMonth, currentDay);
    return stage;
}
```

- та файл «main.cpp» з прикладом створення і використання класу

```
#include <iostream>
```

```
#include <Employee.h>

using namespace std;

int main()
{
    Employee ivan;
    char name[25] = {"Ivan Ivanov"};
    ivan.setName(name);
    ivan.setSalary(10000);
    ivan.setPosition((char *)"Manager", 2004, 3, 1);
    char new_name[25];
    ivan.getName(new_name);
    cout << new_name << endl;
    cout << ivan.getSalary() << endl;
    ivan.setPosition((char *)"Director", 2010, 8, 21);
    ivan.showCareer();
    cout << ivan.getStage(2020, 2, 18) << endl;
    return 0;
}
```

Laboratory work №3.

Task progress:

1. Create the class for a given concept.
2. Describe the class in UML notation.
3. Implement the class as program code.
4. Write a program in which the user will be able to manipulate the created class.

The Variants.

1. The prism
2. The cylinder.
3. The scope.

Application 1. UML Basics

UML (unified modeling language) is a generalized language for the description of subject areas, in particular, software systems, object-oriented analysis and design. It is used for visualization, specification, design and documentation of software systems [1]. In the context of object-oriented programming and analysis, UML contains the following basic components:

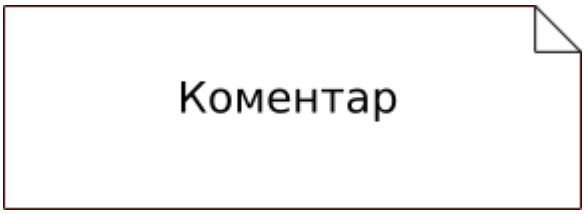
1. Entities that represent the abstractions that make up the model.
2. Connections that connect entities with each other.
3. Diagrams for a static representation of a structure, group sets of entities into a concept. They represent a connected graph of vertices (entities) and their relations (connections). UML includes 13 basic diagrams; one of the most important is the class diagram, which reflects a set of classes, interfaces, and their connections.

Types of Entities:

Structural - conceptual or physical elements of the system, the main view in structural entities is the ***class***.

Behavioral - dynamic elements of diagrams, the main one is the interaction, which means the exchange of messages between diagram objects, indicated by straight line with an arrow, above which the name of the operation is indicated:

Anotational - elements explain the parts of UML models, the main view is a note, indicated by rectangle with a curved upper right corner, inside which a text or graphic comment is placed.



The structural essence of a class is a description of a set of objects with the same properties (attributes), operations (methods), relations, and behavior. It is graphically depicted as a rectangle, divided into 3 blocks by horizontal lines, the order of blocks (from top to bottom):

1. The name of the class.
2. Attributes (properties) of the class.
3. Operations (methods) of the class.

Additionally, one of three types of visibility can be specified for attributes and operations:

• **public** is indicated by a “+” sign before the name of the attribute or method;

• **protected** is indicated by a "#" before the name of the attribute or method;

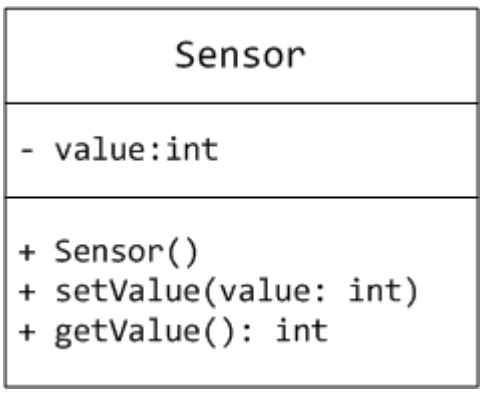
• **private** is indicated by a “-”before the name of the attribute or method.

Each class must have a unique name, distinguishes it from other classes. The name of the class is a text string, which can consist of any number of letters, numbers and other signs, except for a colon and a point, and can be composed of several lines. Best practice is short (if possible) class names consisting of the names of the concepts being modeled.

One approach to class naming is the so-called CamelCase (camel letter), according to which every word in the class name is form he capital letter, for example: Sensor, TemperatureSensor or VelocityVectorField and etc.

The names of *abstract* classes are written in *italics*.

For example:



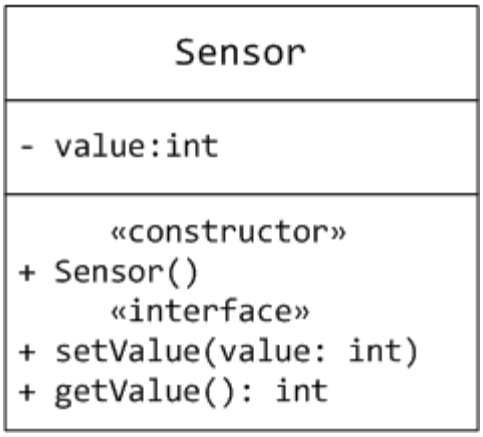
Class attribute - a named property of a class, sets the range of possible values of attributes. This property has all instances of a given class. Each class can contain any number of attributes or not contain any. In the latter case, the attribute block remains empty. Attribute names also

consist of domain names, but are written in a lowerCamelCase (lower camel letter) variation, according to which all words in the attribute name except the first letter begin with a capital letter, for example: **value, rangeMaximum or pointPositionX.**

For attributes, you can specify the type, number of elements (if the attribute is an array), the initial value.

Method is a named implementation of some class function. A class can have any number of methods, or not have any, in which case the block of operations remains empty. LowerCamelCase can also be used to name methods, but the verb is most often used as the first word, for example getValue, isEqual, saveFileAndExit. For a method, you can specify its signature, including the names, types, and default values of all parameters and the type of the return value.

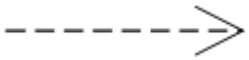
Class declaration requires display all its attributes and methods. For clarity, long lists of attributes or methods can be grouped using stereotypes - the names of groups of attributes or methods that follow them. Stereotypes are indicated in angle brackets:



Relations between classes in UML.

There are 4 types of connections:

1. Dependence - a relationship between two elements of a model in which a change in one element (independent) leads to a change in the behavior of the second element (dependent). Represents as a dotted line, sometimes with an arrow pointing to the independent entity:

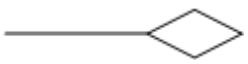


2. Association - reflects how objects of one entity (class) are associated with objects of another entity. Depicted as a solid line with an arrow indicating the direction of association. Usually, associations contain a name that is placed above the arrow.

Associations can be multiple, in which case both ends of the arrow are affected by ranges of integers indicating the possible number of related objects, for example: exact number 3, zero or one 0..1, any number 0 .. * or *, one or more 1 .. * or a range of integers 2..5.

Aggregation (by reference) is a special kind of association that reflects the structural connection of the whole with its components. Used when one class is a container for other classes. The aggregation relationship cannot include more than two classes (container and its contents).

Depicted as:



Composition, or aggregation by value, is a more strict variant of aggregation, in which the destruction of an instance of a container class leads to the destruction of instances of classes that are contained in it. Graphic designation:

3. Generalization - used to display the inheritance in which the descendant element is built according to the specifications of the parent element. The descendant will inherit the structure and behavior of the parent. It is graphically indicated as a line with an empty arrow that points to the parent element:

4. Implementation is a connection between two classes in which one of them (the provider) defines the agreement that the second class (the client) follows. This type of connection can be between interfaces and classes that implement these interfaces. It is indicated by a dotted line from the customer with an empty arrow that indicates the supplier: