MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

NATIONAL TECHNICAL UNIVERSITY
"KHARKIV POLYTECHNIC INSTITUTE"

GUIDELINES

to the laboratory work for the courses “Fundamentals of Programming” and “Algorithmization and Programming”

for students of specialty 121 “Software engineering” and 122 “Computer Science”

Approved
by the Editorial and Publishing Council of the University,
Protocol №1, February 13, 2025.

Kharkiv
NTU “KhPI”
2025

Guidelines to the laboratory work for the courses “Fundamentals of Programming” and “Algorithmization and Programming” for students of specialties 121 “Software engineering” and 122 “Computer Science” // Author: L. V. Ivanov. – Kharkiv: NTU “KhPI”, 2025. – 69 p.

Author: L. V. Ivanov

Reviewer: docent A. A. Pashnev

Department of Software Engineering and Management Intelligent Technologies

CONTENTS

LABORATORY TRAINING 1

Pointers to functions and header files

1 Training tasks

1.1 Output of the table of values of function

Create a separate translation unit in which you should implement the function of outputting a table of values of a certain function with a certain step. Function parameters are: the beginning, end of the interval, step and pointer to the function, the values of which will be displayed in the table. In the header file, you must declare the type of pointer to function and the prototype of the value table output function. In the implementation file, define the value table output function.

In another translation unit, you should place the functions for which you want to output values in a table, as well as the main () function, which outputs tables of values to at least two different functions.

1.2 Individual task

Write a program that implements exhaustive search of some value according to an individual assignment. Necessary value can be found by testing intermediate values of a function. Use typedefs and pointers to functions.

The source code should be split into two translation units. The first translation unit will be represented by both header file and implementation file. The `typedef` definition, as well as declaration of function that searches necessary value, should be placed into header file. The definition of this function will take place in implementation file corresponding to header file. The testing function, as well as `main()` function, should be placed into another translation unit.

You should check the functionality of the program on at least two arbitrary functions. One of the functions can be standard.

Note: To calculate first derivative of $y(x)$, you can use the following formula:

$$y'(x) = (y(x + \Delta x) - y(x)) / \Delta x$$

where Δx is some tiny value, such as 0.0000001.

Table 1.1 – Individual tasks

Index of variant	Rule of searching:
1, 15	Maximum value of the second derivative
2, 16	Minimum value of the first derivative
3, 17	The least root
4, 18	The greatest root
5, 19	The sum of minimum and maximum values
6, 20	The product of minimum and maximum values
7, 21	Roots count
8, 22	The least root of the second derivative
9, 23	Minimum value of the second derivative
10, 24	Maximum value of the first derivative
11, 25	The least root of the first derivative
12, 26	The greatest root of the first derivative
13, 27	The greatest root of the second derivative
14, 28	The sum of minimum and maximum values

1.3 Working with an array of pointers to functions

In a separate namespace, describe an alias for a type of pointer to a function that takes two arguments of type `double` and returns a result of type `double`. In the same namespace, implement a function with result of type `void`, which receives as a parameter the previously described pointer to a function and the values of two arguments and displays the values of the arguments and the value of the function on the screen.

Outside the created namespace, define a function that takes two arguments of `double` type and returns the sum of the second powers of the arguments.

In the `main()` function, create an array of pointers to functions of the previously described type. In the array, place pointers to the standard functions of the `cmath` header file, which receive two arguments, as well as a pointer to the previously created function for calculating the sum of the second powers. Read the values of two arguments from the keyboard and output the arguments and function for all pointers to functions from the array in a loop.

Recommended functions of `cmath` header file that take two type arguments and return a result of type `double`:

– `pow()` an arbitrary power of a number;

– `hypot()` the value of the hypotenuse for the two specified legs of a right triangle;

– `fmax()` the maximum of two values;

– `fmin()` minimum of two values;

Demonstrate different ways to connect namespace elements.

2 Guidelines

The goal of this laboratory training is obtaining practical skills in creation and usage of pointers to functions and header files in C++.

The following questions should be explored:

– C++ language and its versions;

– integrated development environments for C++ development;

– typedefs;

– pointers to functions;

– header files;

– namespaces.

Necessary theoretical concepts and syntax constructs are discussed in lectures on the topic “Callback. Physical and logical structure of the program”.

Most tasks can be implemented based on sample programs discussed below.

3 Sample programs

3.1 Bisection (Dichotomy) method

The following program finds roots of an equation using dichotomy method. The algorithm of the method can be simply described as follows:

– The interval on which the equation $f(x) = 0$ has one root is determined.

– In a loop, the middle of the interval is calculated.

The signs of the function at the beginning and inside the interval are compared. If the signs match, there is no root on the first half of the interval, and

we move the origin to the middle of the interval. If the signs are different, the first half has a root, and we move the end of the interval to the middle.

The cycle is repeated until the length of the interval is greater than the specified precision.

The only restriction on use of dichotomy method is that the equation must have exactly one root on a given interval.

```
#include <iostream>
#include <cmath>

using std::cout;
using std::endl;
using std::sin;

typedef double (*FuncType)(double);

// The fourth argument has default value:
double root(FuncType f, double a, double b, double eps = 0.001)
{
    double x;
    do
    {
        x = (a + b) / 2;
        if (f(a) * f(x) > 0)
        {
            a = x;
        }
        else
        {
            b = x;
        }
    }
    while (fabs(b - a) > eps);
    return x;
}

double g(double x)
{
    return x * x - 2;
}

void main()
{
    cout << root(g, 0, 6) << endl;
    cout << root(g, 0, 6, 0.00001) << endl;
    cout << root(sin, 1, 4) << endl;
    cout << root(sin, 1, 4, 0.00001) << endl;
}
```

As can be seen from the above code, a callback mechanism is used to obtain intermediate values of the function.

The program can also be extended by adding an alternative algorithm for finding the root, for example, full search. We go through the interval with a step `eps` and exit the loop when the sign has changed on a small interval.

Now the user, for example, can choose the root search algorithm. The code will be as follows:

```
#include <iostream>
#include <cmath>

using std::cin;
using std::cout;
using std::endl;
using std::sin;

typedef double (*FuncType)(double);
typedef double (*AlgorithmType)(FuncType, double, double, double);

double dichotomy(FuncType f, double a, double b, double eps = 0.001)
{
    double x;
    do
    {
        x = (a + b) / 2;
        if (f(a) * f(x) > 0)
        {
            a = x;
        }
        else
        {
            b = x;
        }
    } while (b - a > eps);
    return x;
}

double fullSearch(FuncType f, double a, double b, double eps = 0.001)
{
    for (double x = a; x < b; x += eps)
    {
        if (f(x) * f(x + eps) <= 0)
        {
            return x + eps / 2;
        }
    }
    return INFINITY;
}

double g(double x)
{
    return x * x - 2;
}

void main()
{
    cout << "Enter the solution method (1 - dichotomy method, "
         << "2 - the method of full search):";
    int answer;
```

```
    cin >> answer;
    AlgorithmType root = nullptr;
    switch (answer)
    {
        case 1: root = dichotomy;
            break;
        case 2: root = fullSearch;
            break;
    }
    if (root != nullptr)
    {
        cout << root(g, 0, 6, 0.0000001) << endl;
        cout << root(sin, 1, 4, 0.0000001) << endl;
    }
    else
    {
        cout << "error" << endl;
    }
}
```

As you can see from the code, `typedef` can use previous `typedef` definitions.

Running this program can, in particular, show very low efficiency of a full search.

3.2 Using header files

Consider we want to create unit built from header file `SomeFile.h` and implementation file `SomeFile.cpp`.

We should add include guards first:

```
#ifndef SomeFile_h
#define SomeFile_h

#endif
```

Header file contains function prototype before `#endif`:

```
#ifndef SomeFile_h
#define SomeFile_h

int sum(int a, int b);

#endif
```

Implementation file contains function definition placed after `#include "SomeFile.h"` directive.

```
#include"SomeFile.h"

int sum(int a, int b)
{
    return a + b;
```

```
}
```

We should include `SomeFile.h` in main unit:

```
#include <iostream>
#include "SomeFile.h"

using namespace std;

void main()
{
    int x, y;
    cout << "Enter two integer values:" << endl;
    cin >> x >> y;
    int z = sum(x, y);
    cout << "Sum is " << z << endl;
}
```

3.3 Working with an array of pointers to functions

Suppose we want to create a function that, for a given argument and two defined functions, returns the sum of the values of these functions. This problem can be solved through the use of pointers to functions.

In a separate namespace, we describe an alias for a type of pointer to a function that takes argument of type `double` and returns a result of type `double`. In the same namespace, we implement a function with result of type `double`, which receives as parameters the previously described pointers to functions and the value of argument and returns the sum of the values of these functions.

Outside the created namespace, define a `sqr()` function that takes one argument of `double` type and returns the second power of the argument.

In the `main()` function, we create two arrays of pointers to functions of the previously described type. In the arrays, we place pointers to the standard functions of the `cmath` header file, which receive one argument, as well as a pointer to the previously created function for calculating the second power. We read the values of an argument from the keyboard and output the function for all pointers to functions from arrays in a loop:

```
#include <iostream>
#include <cmath>

using std::cin;
using std::cout;
using std::endl;
```

```
using std::sin;
using std::cos;
using std::exp;
using std::sqrt;

namespace Func
{
    typedef double (*OneArgFunc)(double);

    double sum(OneArgFunc first, OneArgFunc second, double x)
    {
        return first(x) + second(x);
    }
}

using Func::OneArgFunc;

double sqr(double x)
{
    return x * x;
}

int main()
{
    const int n = 3;
    OneArgFunc firstArr[n] = { sin, exp, sqr };
    OneArgFunc secondArr[n] = { cos, sqr, sqrt };
    double x;
    cin >> x;
    for (int i = 0; i < n; i++)
    {
        cout << Func::sum(firstArr[i], secondArr[i], x) << endl;
    }
    return 0;
}
```

As can be seen from the example, we refer to typedef using the `using` directive, and refer to the function from the `Func` namespace with the use of a prefix.

4 Quiz

1) What programming paradigms does C++ support?

2) What are the versions of the C++ standard?

3) What is an integrated development environment?

4) How to create synonym for existing type?

5) What is pointer to function?

6) What is usage of painters to functions?

7) How to define pointer to function?

8) What is translation unit?

9) What is the usage of #define directive?

10) What are rules of distribution of source code between header file and implementation file?

11) What is the difference between inclusion of standard header files and user header files?

12) What are include guards?

13) What is namespace?

14) How to join several namespaces into one?

15) How to define alias for existing namespace?

LABORATORY TRAINING 2

Working with enumerations and structures

1 Training tasks

1.1 An enumeration for presenting the months of the year

Create an enumeration to represent the months of the year. Implement and demonstrate overloaded operators ++ so −− that after December there is January and before January there is December.

1.2 3D-points

Create a structure to describe a 3D-point. Write a program that calculates distance between two 3D-points. To calculate the distance, create a function with two parameters of structure type to represent the points.

1.3 Representation and processing students data

Create a structure for presenting data concerned with student. The structure should contain the following data members:

– student identity number (`unsigned int`);

– surname (array of characters);

– marks of the last session in the form of an array of integers from 0 to 100 (marks on subjects).

Implement a function that outputs student data to the console window. The first parameter of the function should be a structure that describes the student.

Implement functions that receive an array of pointers to student and the length of the array and execute

– array sorting according to the criterion specified in the individual task;

– searching for data about students that meet the condition given in the individual task;

– displaying all items of the array in the console window.

When sorting an array of structures, swap the items of the array of pointers instead of swapping two structures.

Create an array of students. Create an array of pointers to students, filling it with addresses of structures from the array of students. Demonstrate sorting and searching for students.

Individual tasks are listed in the table 2.1:

Table 2.1 – Individual tasks

Index in the List of Students	Sorting Conditions	Data Selection Conditions
1	Alphabetically	With an average score in the interval "64" – "74"
2	Increasing the average score	With a surname length of more than 7 characters
3	Decreasing the length of surname	With an average score in the interval "90" – "100"
4	Descending numbers of student identity numbers	Surname starts with the letter "A"
5	Decreasing the average score	Surname ends with the letter "a"
6	Increasing the sum of scores	With the odd length of surname
7	Increasing the length of surname	With odd numbers of student identity numbers
8	Alphabetically	With even numbers of student identity numbers
9	Alphabetically reversed	More "A" grades than "D" grades
10	Increasing numbers of student identity numbers	Surname contains the letter "e"
11	Decreasing the product of scores	With the odd length of surname
12	Increasing the product of scores	With a surname length of less than 8 characters
13	Alphabetically	With an even sum of scores
14	Alphabetically reversed	With odd numbers of student identity numbers
15	Alphabetically	With an average score in the interval "75" – "81"
16	Increasing the average score	With a surname length of less than 7 characters

Continuation of table 2.1

17	Increasing the length of surname	With an average score in the interval "82" – "89"
18	Increasing numbers of student identity numbers	Surname starts with the letter "A"
19	Increasing the average score	Surname ends with the letter "a"
20	Increasing the sum of scores	With the odd length of surname
21	Increasing the length of surname	With even numbers of student identity numbers
22	Alphabetically reversed	With even numbers of student identity numbers
23	Alphabetically	More "A" grades than "E" grades
24	Increasing numbers of student identity numbers	Surname contains the letter "o"
25	Decreasing the product of scores	With the even length of surname
26	Decreasing the product of scores	With a surname length of less than 9 characters
27	Alphabetically reversed	With an odd sum of scores
28	Alphabetically	With odd numbers of student identity numbers

1.4 Working with a linked list

You should write a program that provides file input and output and implements an assignment of Laboratory training #5 of the course "Programming Basics (Part 1)". You should implement following steps:

– definition of a constant (`n`) which determines column count of two-dimensional array;

– opening file for reading (file should be prepared using some text editor);

– reading integer values until the end of file and storing them in the linked list;

– creation of two-dimensional array in free store; row count should be calculated based on amount of integer values read from file and count of columns;

– filling of two-dimensional array row by row; missing items of the last row should be set to zeroes;

– removing elements of linked list from free store;

– implementation of the task of Laboratory training # 5 of the course "Programming Basics (Part 1)";

– storing results in a new file;

– removing both arrays using `delete` operators.

2 Guidelines

The goal of this laboratory training is obtaining practical skills in working with enumerations and structures in C++.

The following questions should be explored:

– user defined types;

– enumerations;

– structures;

– unions;

– sorting arrays of structures using pointers;

– use of linked lists.

Necessary theoretical concepts and syntax constructs are discussed in lectures on the topic “Creating and using custom types”.

Most tasks can be implemented based on sample programs discussed below.

3 Sample programs

3.1 Enumeration for presenting the days of week

Suppose there is an enumeration type:

```
enum DayOfWeek {
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

To overload the prefix `++` operator, in the same namespace where the type `DayOfWeek` is defined, the following function must be implemented:

```
// Overloading the prefix ++ operator
DayOfWeek operator++(DayOfWeek& day)
{
    if (day == Saturday) // special case
    {
        day = Sunday;
    }
    else
    {
        day = (DayOfWeek) (day + 1); // all other days
    }
    return day;
}
```

To overload the postfix ++ operator, we need to implement a function with two parameters:

```
// Overloading the postfix ++ operator
DayOfWeek operator++(DayOfWeek& day, int)
{
    DayOfWeek oldDay = day;// save the previous day
    operator++(day);       // call the previous function
    return oldDay;         // return the previous day
}
```

In the list of formal parameters `int` is a type of parameter that is not used, but its presence allows the compiler to distinguish a postfix operation from a prefix operation.

The entire code of the program will be as follows:

```
#include <iostream>

enum DayOfWeek {
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};

// Overloading the prefix ++ operator
DayOfWeek operator++(DayOfWeek& day)
{
    if (day == Saturday) // special case
    {
        day = Sunday;
    }
    else
    {
        day = (DayOfWeek) (day + 1); // all other days
    }
    return day;
}
```

```
// Overloading the postfix ++ operator
DayOfWeek operator++(DayOfWeek& day, int)
{
    DayOfWeek oldDay = day;// save the previous day
    operator++(day);       // call the previous function
    return oldDay;         // return the previous day
}

// Getting the names of the days of the week
const char* getName(DayOfWeek day)
{
    switch (day)
    {
        case Sunday:
            return "Sunday";
        case Monday:
            return "Monday";
        case Tuesday:
            return "Tuesday";
        case Wednesday:
            return "Wednesday";
        case Thursday:
            return "Thursday";
        case Friday:
            return "Friday";
        default:
            return "Saturday";
    }
}

int main()
{
    // Sequentially output the days from Wednesday to the following Monday
    for (DayOfWeek d = Wednesday; d != Tuesday; d++)
    {
        std::cout << getName(d) << std::endl;
    }
    return 0;
}
```

3.2 Distance between points

In the following example, we create structure which represents point on the screen. The program calculates distance between two points:

```
#include <iostream>
#include <cmath>

struct Point
{
    int x, y;
};

double sqr(double x)
{
    return x * x;
}

double distance(Point p1, Point p2)
{
```

```
    return std::sqrt(sqr(p1.x - p2.x) + sqr(p1.y - p2.y));
}

int main()
{
    Point p1 = { 1, 2 };
    Point p2 = { 4, 6 };
    std::cout << distance(p1, p2);
    return 0;
}
```

3.3 Working with an array of structures

The program we will create describes the structure for representing the city. The city is described by its name, the name of the country in which it is located, the name of the region where the city is located and the number of inhabitants.

The program should create an array of several cities and implement the following functions:

– sorting of cities by population;

– search for cities that belong to a certain region (region);

– display of information about cities.

For a better understanding of the code and to ensure the possibility of its multiple use, individual actions (sorting, searching, output) should be implemented as separate functions. To increase the efficiency of working with data, sorting is performed not for an array of cities, but for an array of pointers to cities.

The program code will be as follows:

```
#include <iostream>
#include  <cstring>

// Structure for describing the city
struct City
{
    char name[30];
    char country[30];
    char region[30];
    int population;
};

// Sort by population
// Instead of modifying the array of cities, we modify the array of pointers
void sortByPopulation(City** arr, int size)
{
    bool mustSort; // repeat sorting
                   // if mustSort is true
    do
    {
        mustSort = false;
```

```
        for (int i = 0; i < size - 1; i++)
        {
            if (arr[i]->population > arr[i + 1]->population)
                // Swap items
            {
                City* temp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = temp;
                mustSort = true;
            }
        }
    }
    while (mustSort);
}

// Display data about the city.
// To improve efficiency, we use reference
// to a constant object
void printCity(const City& city)
{
    std::printf("City: %s. Country: %s. Region: %s. Population: %d\n",
        city.name, city.country, city.region, city.population);
}

// Display data about all cities.
// We access cities through an array of pointers
void pintCities(City** arr, int size)
{
    for (int i = 0; i < size; i++)
    {
        printCity(*arr[i]);
    }
    std::cout << "\n";
}

// Display data about cities,
// which are in a certain region
void printIf(City cities[], int size, const char* region)
{
    for (int i = 0; i < size; i++)
    {
        if (std::strcmp(cities[i].region, region) == 0) {
            printCity(cities[i]);
        }
    }
}

int main()
{
    const int n = 4;

    // Create and fill the array of cities:
    City cities[n] = {
        { "Kharkiv", "Ukraine", "Kharkiv region", 1421125 },
        { "Poltava", "Ukraine", "Poltava region", 284942 },
        { "Lozova", "Ukraine", "Kharkiv region", 54618 },
        { "Sumy", "Ukraine", "Sumy region", 264753 }
    };

    // Create and fill an array of pointers:
    City* pointers[n];
```

```
    for (int i = 0; i < n; i++)
    {
        pointers[i] = &cities[i];
    }

    pintCities(pointers, n);
    sortByPopulation(pointers, n);
    pintCities(pointers, n);
    printIf(cities, n, "Kharkiv region");
    return 0;
}
```

As can be seen from the code, the elements of the array of pointers are swapped during sorting. Nothing changes in the structure array.

3.4 Reverse order

The following program reads floating point numbers from a text file and puts these numbers into a new file in reverse order. Data are read until the end of file. During data loading, a temporary chain is used. A structure called Link is used for representation of particular links of a chain. Loaded data are copied into a new array, and then temporary chain should be destroyed.

```
#include<iostream>
#include <fstream>

using namespace std;

// A link:
struct Link
{
    double data;
    Link *next;
};

// The function reads numbers from the specified file
// and returns a pointer to the initial item of the array.
// The array is created inside the function, its items
// are located in the free store
// The parameter count after the function completes contains the length of //
the array
double *readFromFile(const char *fileName, int &count)
{
    // Prepare the linked list to work and create a file stream:
    Link *first = 0;
    Link *last = 0;
    Link *link;
    ifstream in(fileName);
    double d;
    count = 0;      // counter of numbers read from the file
    while (in >> d) // read to the end of the file
    {
        count++;
        // Create a new list item:
        link = new Link;
        link->data = d;
```

```
        link->next = 0;
        if (last == 0)
        {
            first = link;
        }
        else
        {
            last->next = link;
        }
        last = link;
    }
    // Create and fill an array of numbers:
    double *arr = new double[count];
    link = first;
    for (int i = 0; i < count; i++)
    {
        arr[i] = link->data;
        link = link->next;
    }
    // Deleting elements of the linked list from the free store:
    while (first)
    {
        link = first;
        first = first->next;
        delete link;
    }
    return arr;
}

// Writes the elements of the array arr of length count
// to the specified text file
void outToFile(const char *filename, double *arr, int count)
{
    ofstream out(filename);
    for (int i = count - 1; i >= 0; i--)
    {
        out << arr[i] << " ";
    }
}

int main()
{
    int count = 0;
    double *arr = readFromFile("data.txt", count);
    outToFile("results.txt", arr, count);
    delete [] arr;
    return 0;
}
```

4 Quiz

1) What syntactic constructs C++ provides to create user defined types?

2) What is use of anonymous enumerations?

3) What is a scoped enum?

4) What are structures?

5) What is the difference between structures and arrays?

6) Why do you need to place semicolons after brace that closes structure's body?

7) How to transfer structures to functions?

8) What are bit fields?

9) In what cases bit fields enhance the effectiveness of the program?

10) What are the features of unions and for what they are used?

11) What are the potential dangers of using unions?

12) Why is it appropriate to use arrays of pointers when sorting arrays of structures?

13) What are dynamic data structures?

14) What are the advantages of lists compared to arrays?

15) What are the disadvantages of lists compared to arrays?

LABORATORY TRAINING 3

Creation and use of c++ classes

1 Training tasks

1.1 Class for representing a point in three-dimensional space

Create a class for representing a point in three-dimensional space (3D points). The class must meet the following requirements:

– have a constructor without parameters;

– have a constructor with three parameters;

– contain data members of `double` type to represent the coordinates of a point;

– implement public data access functions (setters and getters).

To calculate the distance, create an operator function `operator-()` ("minus") with two parameters (3D points) and a result of type `double`. Declare this function as a friend of the class.

In the `main()` function, you should create two objects of 3D point type by applying different constructors. Use access functions to set and read values. Calculate the distance between two points by using the minus operation, then change the coordinates of one of the points and calculate the distance between the two points by explicitly calling the `operator-()` function.

1.2 Class for representing a simple fraction

Create a class that represents a simple fraction. Implement constructors, a function for reduction of fractions, and overload operations `+`, `-`, `*`, `/`, `<`, `<=`, `>`, `>=`, as well as stream input and output. In the output operator function, implement the most correct output: get and output the whole part for improper fractions, do not output the denominator if it is equal to 1 or the numerator is equal to 0, etc.

Demonstrate class features in the `main()` function.

Read two fractions and demonstrate the reduction of fractions and all overloaded operations in the `main()` function.

Note: you should not store the integer part of the fraction separately.

1.3 Classes for representing students and groups

Create classes for presenting data about students and groups of students. The "Student" class should contain the following private data members:

– student ID number (`unsigned int`);

– last name (pointer to character); the corresponding string will be created in a free store as needed;

– grades for the last session in the form of an array of integers from 0 to 100 (grades by subject); an array should be created in a free store;

– a pointer to an object of class "Group".

Define the following members in the "Student" class:

– constructor without parameters and constructor with parameters;

– copy constructor;

– destructor;

– data access functions (setters and getters);

– overloaded assignment operator.

The operator function of output data about the student to the stream should be implemented as an external function (class friend). To ensure sorting, overload the necessary comparison (relation) operator functions.

A constant should be defined for the maximum possible number of students (for example, 50)

The following data elements should be defined in the "Group" class:

– group index;

– an array of pointers to students of the maximum possible length;

– the actual number of pointers in the array.

Define the following elements in the "Group" class:

– constructor without parameters and constructor with parameters;

– group index data access functions (setter and getter);

– overloaded assignment operator;

– the function of sorting according to the specified criterion;

– finding students by certain attribute.

Instead a getter for an array of pointers, overload the subscript operator of getting an item by index. An operator function of output data is required.

In the `main()` function, create a "Group" object, add an array of students and demonstrate all implemented functions, in particular, sorting and searching according to task 3.1 of the previous laboratory training.

1.4 Class for representing a two-dimensional array

Develop a class to represent two-dimensional array of integers (matrix) of arbitrary size. Create constructors and destructors, overload operations of addition, subtraction and multiplication (according to the rules of working with matrices), getting items by index, as well as stream input and output. Create your own exception classes and generate relevant objects.

Create a separate function that gets reference to an array and performs actions listed in the table 3.1. The function should not be friend or member function.

Table 3.1 – Individual tasks

Index of variant		Rule of a source array transformation
1	15	All elements with odd values should be doubled
2	16	All elements with even values should be replaced with their squares
3	17	All elements with null value should be replaced with ones
4	18	All elements with even values should be doubled
5	19	All elements should be replaced with their absolute magnitudes
6	20	All elements with even values should be tripled
7	21	All positive elements should be replaced with integer parts of their Briggs (base ten) logarithms
8	22	All negative elements should be replaced with their squares
9	23	All positive elements should be replaced with integer parts of their Napierian (natural) logarithms
10	24	All positive elements should be replaced with integer parts of their square roots
11	25	All positive elements with even values should be doubled
12	26	All negative elements with odd values should be tripled
13	27	All negative elements with odd values should be doubled
14	28	All positive elements with even values should be tripled

Demonstrate all class features in the `main()` function. Solve individual task with catching of possible exceptions.

1.5 Calculation of the sum of entered values

Create a class with a private data member, getter, and constructor with one argument. This class also should contain a static private data member that stores the sum of all data members of previously created objects. Each call of the constructor includes adding of a new value to the static field. Public static function of the same class should return this sum.

The `main()` function should contain creation of several objects and output of the calculated sum.

2 Guidelines

The goal of this laboratory training is obtaining practical skills in creation and use of C++ classes.

The following questions should be explored:

Prerequisites for the emergence of an object-oriented approach;

– definition of classes;

– object initialization; constructors and destructors;

– class scope; Nested classes;

– static class members;

– class friends;

– operator overloading;

– exception handling;

– class composition.

Necessary theoretical concepts and syntax constructs are discussed in lectures on the topic “Creating classes. Class members”.

Most tasks can be implemented based on sample programs discussed below.

3 Sample programs

3.1 Class for representing a point on the screen

Our goal is to create a class to describe a point on the screen. The class must meet the following requirements:

– have a constructor without parameters;

– have a constructor with two parameters;

– contain data elements of the type int to represent the coordinates of a point on the screen;

– implement public data access functions (setters and getters).

To calculate the distance, we'll create a separate function that we will define as the second class. We can create a static class function to calculate the second power.

The source code will be as follows:

```
#include <iostream>
#include <cmath>

// Class for representing a point on the screen
class Point2D
{
    friend double distance(Point2D, Point2D);
private:
    int x, y; // coordinates of the point
public:
    // A constructor without parameters calls another constructor:
    Point2D() : Point2D(0, 0) { }
    // The constructor initializes the data members:
    Point2D(int x, int y) : x(x), y(y) { }
    int getX() { return x; }
    void setX(int x) { this->x = x; }
    int getY() { return y; }
    void setY(int y) { this->y = y; }
    // Auxiliary static function for calculating the second power:
    static double sqr(double x) {  return x * x; }
};
// Calculate the distance between two points
// The friend function has access to the data members of the class
double distance(Point2D p1, Point2D p2)
{
    return std::sqrt(Point2D::sqr(p1.x - p2.x) +
           Point2D::sqr(p1.y - p2.y));
}

int main()
{
    // We create the first point using the constructor with parameters:
    Point2D p1(1, 2);
    std::cout << p1.getX() << " " << p1.getY() << "\n";
```

```
    // For the second point, we indicate the coordinates with setters:
    Point2D p2;
    p2.setX(4);
    p2.setY(6);
    std::cout << p2.getX() << " " << p2.getY() << "\n";
    // Calculate the distance:
    std::cout << distance(p1, p2);
}
```

3.2 Class for representation of mathematical vector

Suppose we need to create a class for representation of mathematical vector in two-dimensional space. We can describe the vector using two coordinates: *x* and *y*. Typical operations on vector are finding sum, multiplication by scalars, getting scalar product. For reasons of convenience of working with the objects of our class we'll overload appropriate operators. Operator functions can be implemented as friend functions of our class. The C++ syntax allows us to place the implementation of these functions in the class body. We also need to overload input and output operations.

The checking class capabilities is performed the `main()` function. The program will be is follows:

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
using std::istream;
using std::ostream;

class Vector {
    friend istream& operator>>(istream& in, Vector& v)
        { return in >> v.x >> v.y; }
    friend ostream& operator<<(ostream& out, const Vector& v)
        { return out << "x=" << v.x << " y=" << v.y; }
    friend Vector operator+(Vector v1, Vector v2)
        { return Vector(v1.x + v2.x, v1.y + v2.y); }
    friend Vector operator*(double k, Vector v)
        { return Vector(v.x * k, v.y * k); }
    friend Vector operator*(Vector v, double k)
        { return operator*(k, v); }
    friend double operator*(Vector v1, Vector v2)
        { return v1.x * v2.x + v1.y * v2.y; }
private:
    double x, y;
public:
    Vector() { x = y = 0; }
    Vector(double x, double y);
    double getX()       { return x; }
    void setX(double x) { this->x = x; }
    double getY()       { return y; }
    void setY(double y) { this->y = y; }
```

```
};

Vector::Vector(double x, double y) {
    this->x = x;
    this->y = y;
}

void main()
{
    Vector v1, v2;
    cout << "Input first vector: ";
    cin >> v1; // For example, 1 2
    cout << "Input second vector: ";
    cin >> v2; // For example, 3 4
    cout << v1 + v2 << endl; // x=4 y=6
    cout << v1 * v2 << endl; // 11
    cout << v1 * 2 << endl;  // x=2 y=4
    cout << 3 * v2 << endl;  // x=9 y=12
}
```

This example shows that the scalar multiplication operator should be individually implemented for different locations of operands. Constructor with parameters implemented outside of class body, because its implementation requires more than one line.

3.3 Classes for representing the city and country

In the previous laboratory training, an example of creating a structure to represent the city, as well as working with an array of structures, was given. Now, using the previous solutions, we can create classes to represent the city and country, and the "Country" object should store an array of cities located in this country. For simplicity, let's assume that we will store no more than 100 cities in our array.

Note: this assumption will be removed in later stages of development of this solution.

The `City` class will have the following data members:

– name of the city;

– pointer to the country in which the city is located;

– name of the region;

– population of the city.

It is advisable to place the names of the city and region in the free store. A class must provide a constructor with no parameters and a constructor with four

parameters. Memory for data will be freed from the free store in the destructor. In addition to setters and getters, an overloaded assignment operator should be implemented to avoid errors with the free store. We implement the operator function of writing data about the city to the stream as an external friend function. To implement output, it is advisable to use the `sprintf()` function, which formats a string (similarly to `printf()`, which outputs a string to the console window). In order to use the `sprintf()` function you should add the definition `#define _CRT_SECURE_NO_WARNINGS`.

To ensure that cities are sorted by population, comparison operator for two cities should be overloaded. Such a function can be implemented as an external one. For our algorithm, it is sufficient to redefine the `operator>()` function. Other comparison operations may be useful for other algorithms.

The following data members will be created in the `Country` class:

– name of country;

– an array of city directories;

– the actual number of pointers in the array.

We will create an array of pointers to cities of the maximum length, and we will use the logical (actual) length, which determines how many pointers were actually written into the array. This approach is not universal, but it will ensure the best efficiency of the program.

We need to create a getter and setter for the country name. Instead of a getter for an array of pointers, we can overload the operator of getting an element by index (subscript operator). The `setCities()` function will set data about all cities. In addition, a city sorting function and an external output operator function are required.

In the `main()` function, we will create a country object and an array of cities (via a helper function), and then demonstrate sorting and data output. The source code will be as follows:

```
#define _CRT_SECURE_NO_WARNINGS
#include <cstring>
#include <iostream>
```

```
using std::strlen;
using std::strcpy;
using std::cout;
using std::endl;

const int MAX_COUNT = 100; // Maximum number of cities

// The class must be declared so that the pointer can be created:
class Country;

// Class to represent the city
class City
{
    // Overloaded operator to output to a stream
    friend std::ostream& operator<<(std::ostream& out, const City& city);
private:
    char *name = nullptr;      // city name
    Country *country = nullptr;// pointer to the location country
    char *region = nullptr;    // the name of the region
    int population = 0;        // population
public:
    // Constructors:
    City() { }
    City(const char* name, Country* country, const char* region, int
population);
    City(const City& city);

    ~City(); // destructor

    // Getters:
    const char* getName() const { return name; }
    Country* getCountry() const { return country; }
    const char* getRegion() const { return region; }
    int getPopulation() const { return population; }

    // Setters:
    void setName(const char* name);
    void setRegion(const char* region);
    void setCountry(Country* country) { this->country = country; }
    void setPopulation(int population) { this->population = population; }

    // Overloaded assignment operator
    const City& operator=(const City& city);
};

// Constructor with parameters is implemented by calling setters
City::City(const char* name, Country* country, const char* region, int
population)
{
    setName(name);
    setCountry(country);
    setRegion(region);
    setPopulation(population);
}

// Copy constructor
City::City(const City& city)
{
    name = new char[strlen(city.name) + 1];
    strcpy(name, city.name);
```

```
    region = new char[strlen(city.region) + 1];
    strcpy(region, city.region);
    country = city.country;
    population = city.population;
}

// Remove city and region names from memory (if arrays were created)
City::~City()
{
    if (name != nullptr)
    {
        delete[] name;
    }
    if (region != nullptr)
    {
        delete[] region;
    }
}

// Remove the previous name of the city, create a new array and put the new
name
void City::setName(const char* name)
{
    if (this->name != nullptr)
    {
        delete[] this->name;
    }
    this->name = new char[strlen(name) + 1];
    strcpy(this->name, name);
}

// Remove the previous name of the region, create a new array and put the new
name
void City::setRegion(const char* region)
{
    if (this->region != nullptr)
    {
        delete[] this->region;
    }
    this->region = new char[strlen(region) + 1];
    strcpy(this->region, region);
}

// We implement an overloaded assignment operator by calling setters
const City& City::operator=(const City& city)
{
    if (&city != this)
    {
        setName(city.name);
        setCountry(city.country);
        setRegion(city.region);
        setPopulation(city.population);
    }
    return *this;
}

// Overloaded operation of comparing two cities
bool operator>(const City& c1, const City& c2)
{
    return c1.getPopulation() > c2.getPopulation();
}
```

```
// Class to represent the country
class Country
{
    // Overloaded operator to output to a stream
    friend std::ostream& operator<<(std::ostream& out, const Country& country)
    {
        out << country.name << endl;
        for (int i = 0; i < country.count; i++)
        {
            out << *(country.cities[i]) << endl;
        }
        out << endl;
        return out;
    }
private:
    char name[40];                  // name of country
    City *cities[MAX_COUNT] = { }; // array of pointers to cities
    int count = 0;                  // number of pointers in the array
public:
    // Constructors:
    Country() { }
    Country(const char* name) { setName(name); }

    const char* getName() const { return name; } // getter

    // Overloaded operator for getting array items
    City* operator[](int index) const { return cities[index]; }

    // Setters:
    void setName(const char* name) { strcpy(this->name, name); }
    void setCities(City* cities[], int  count);

    void sortByPopulation(); // Sort by population
};

// Fill the array of cities
void Country::setCities(City* cities[], int  count)
{
    this->count = count;
    for (int i = 0; i < count; i++)
    {
        this->cities[i] = cities[i];
        this->cities[i]->setCountry(this);
    }
}

// Sort by population
void Country::sortByPopulation()
{
    bool mustSort = true; // repeat sorting
                          // if mustSort is true
    do
    {
        mustSort = false;
        for (int i = 0; i < count - 1; i++)
        {
            // Dereferencing
            // because we can compare objects, not pointers:
            if (*(cities[i]) > *(cities[i + 1]))
                // Exchange items
```

```
            {
                City* temp = cities[i];
                cities[i] = cities[i + 1];
                cities[i + 1] = temp;
                mustSort = true;
            }
        }
    } while (mustSort);
}

// Overloaded operator for output to a stream
std::ostream& operator<<(std::ostream& out, const City& city)
{
    char buffer[300];
    sprintf(buffer, "City: %s.\tCountry: %s.\tRegion: %s.\tPopulation: %d",
        city.name, city.country->getName(), city.region, city.population);
    out << buffer;
    return out;
}

// Helper function for filling an array of pointers to cities
void createCities(City *cities[])
{
    cities[0] = new City("Kharkiv", nullptr, "Kharkiv region", 1421125);
    cities[1] = new City("Poltava", nullptr, "Poltava region", 284942);
    cities[2] = new City("Lozova", nullptr, "Kharkiv region", 54618);
    cities[3] = new City("Sumy", nullptr, "Sumy region", 264753);
}
int main()
{
    const int realCount = 4;     // we work with four cities
    City *cities[realCount];     // create an array of pointers to cities
    createCities(cities);        // fill the array
    Country country = "Ukraine"; // create the Country object,
                                 // call the constructor with one parameter
    country.setCities(cities, realCount); // copy the cities to the Country
object
    cout << country << endl;     // output all data
    cout << *country[0] << endl; // display information about the city by index
    country.sortByPopulation();  // sort
    cout << country << endl;     // output all data
    // Remove cities stored in array of pointers to cities
    for (int i = 0; i < realCount; i++)
    {
        delete cities[i];
    }
    return 0;
}
```

The disadvantage of the program is the lack of testing of some functions. You can correct this.

3.4 Class for presentation of one-dimensional array

The following example shows the creation and usage of a class that represents one-dimensional array with overloading necessary operators. Array items are allocated in a free store.

```
#include <iostream>

using std::cin;
using std::cout;
using std::endl;
using std::istream;
using std::ostream;
// A class for representing a one-dimensional array
class IntArray
{
    // Friend operator functions for output and input:
    friend ostream& operator <<(ostream& out, const IntArray& a);
    friend istream& operator >>(istream& in, IntArray& a);
private:
    int* pa = nullptr; // pointer to future array
    int  size = 0;     // current array size
public:
    // Nested class for creating an exception object
    class OutOfBounds
    {
        int index; // index out of range
    public:
        OutOfBounds(int i) : index(i) { } // constructor
        int getIndex() const { return index; } // getter for index
    };

    // Constructors:
    IntArray() { }
    IntArray(int n) { pa = new int[size = n]; }
    IntArray(IntArray& arr);

    ~IntArray(); // destructor
    void addElem(int elem); // function to add an element
    int& operator [](int index); // read and write access to elements

   // Overloaded operators:
    const IntArray& operator =(const IntArray& a);
    bool operator ==(const IntArray& a) const;

    int getSize() const { return size; } // returns the number of array
elements
};
// Overloaded operator of writing to stream
ostream& operator <<(ostream& out, const IntArray& a)
{
    for (int i = 0; i < a.size; i++)
    {
        out << a.pa[i] << ' ';
    }
    return out;
}
// Overloaded operator of reading from stream
istream& operator >>(istream& in, IntArray& a)
{
    for (int i = 0; i < a.size; i++)
    {
        in >> a.pa[i];
    }
    return in;
}
// Copy constructor
```

```
IntArray::IntArray(IntArray& arr)
{
    size = arr.size;
    pa = new int[size];
    for (int i = 0; i < size; i++)
    {
        pa[i] = arr.pa[i];
    }
}
// Destructor
IntArray::~IntArray()
{
    if (pa != nullptr)
    {
        delete[] pa;
    }
}
// Adds an item. Allocates an array in a new place
void IntArray::addElem(int elem)
{
    int* temp = new int[size + 1];
    if (pa != nullptr)
    {
        for (int i = 0; i < size; i++)
        {
            temp[i] = pa[i];
        }
        delete[] pa;
    }
    pa = temp;
    pa[size] = elem;
    size++;
}
// Provides read and write access to elements
// Throws an OutOfBounds exception in case of a wrong index
int& IntArray::operator [](int index)
{
    if (index < 0 || index >= size)
    {
        throw OutOfBounds(index);
    }
    return pa[index];
}
// Overloaded assignment operator
const IntArray& IntArray:: operator =(const IntArray& a)
{
    if (&a != this)
    {
        if (pa != nullptr)
        {
            delete[] pa;
        }
        size = a.size;
        pa = new int[size];
        for (int i = 0; i < size; i++)
        {
            pa[i] = a.pa[i];
        }
    }
    return *this;
}
```

```
// Overloaded array comparison operator
bool IntArray:: operator ==(const IntArray& a) const
{
    if (&a == this)
    {
        return true;
    }
    if (size != a.size)
    {
        return false;
    }
    for (int i = 0; i < size; i++)
    {
        if (pa[i] != a.pa[i])
        {

            return false;
        }
    }
    return true;
}
// Global function for finding the minimum array item
// The function does not have direct access to the data and uses
// overloaded operators and member functions
int getMin(IntArray a) // call the copy constructor
{
    int min = a[0];
    for (int i = 1; i < a.getSize(); i++)
    {
        if (min > a[i])
        {
            min = a[i];
        }
    }
    return min;
}
void main()
{
    setlocale(LC_ALL, "UKRAINIAN");
    IntArray a(2); // An array of two items
    cout << "Enter two array elements: ";
    cin >> a;
    cout << "Array items: " << a << endl;
    a.addElem(12);
    cout << "Adding element" << endl;
    cout << "Array items: " << a << endl;
    try
    {
        a[1] = 2;   // changed
        a[10] = 35; // wrong index
    }
    catch (IntArray::OutOfBounds& e)
    {
        cout << "Incorrect index: " << e.getIndex() << endl;
    }
    cout << "New items: " << a << endl;
    IntArray b; // created a new array
    b = a; // copying items
    if (a == b)
    {
        cout << "Arrays a and b are equal" << endl;
```

```
    }
    else
    {
        cout << "Arrays a and b are different";
    }
    cout << "Minimum element: " << getMin(a) << endl;
}
```

As you can see from the example, work with an object outside of class is very similar to working with a conventional array.

3.5 Counting created objects

Suppose we need to perform counting objects of a certain class. We can create a counter in the form of a static data member that holds the number of objects present in memory. The invocation of a constructor provides increasing the counter, and the invocation of a destructor provides decreasing. The program can be as follows:

```
#include <iostream>
using std::cout;
using std::endl;

class ObjectCount
{
private:
    static int count;
public:
    static int getCount()
    {
        return count;
    }
    ObjectCount()
    {
        count++;
    }
    ~ObjectCount()
    {
        count--;
    }
};

// Static data member must be defined and initialized outside of class:
int ObjectCount::count = 0;
void main()
{
    ObjectCount c1;
    cout << c1.getCount() << endl;  // 1
    ObjectCount *p1 = &c1; // copy the address, constructor is not called
    cout << p1->getCount() << endl; // 1
    ObjectCount *p2 = new ObjectCount();
    cout << p2->getCount() << endl; // 2
    delete p2;
    cout << p2->getCount() << endl; // 1
}
```

As we can see from the results of the `main()` function, the `getCount()` function can be called for the object even after it was removed from the heap. This is due to the fact that the only object type or pointer type (but not object itself) is important for compiler if the static function calls. A more correct is function call via class name:

```
void main()
{
    ObjectCount c1;
    cout << ObjectCount::getCount() << endl; // 1
    ObjectCount *p1 = &c1;
    cout << ObjectCount::getCount() << endl; // 1
    ObjectCount *p2 = new ObjectCount();
    cout << ObjectCount::getCount() << endl; // 2
    delete p2;
    cout << ObjectCount::getCount() << endl; // 1
}
```

4 Quiz

1) What is the class and what is its content?

2) What is the concept of encapsulation?

3) What access levels to the class members supports C++?

4) What are access functions?

5) Is it possible to implement member functions outside the class body?

6) How to create a constant member function?

7) What is a `this` pointer and what its usage?

8) What is constructor and what mechanism its call?

9) How many constructors without parameters can be created in the same class?

10) How to create a class with no constructors?

11) What is the copy constructor and when it should be defined?

12) What is destructor and what mechanism its call?

13) How many destructors can be defined in a class?

14) What determines the class scope?

15) Is it possible to create classes within other classes?

16) What is a static class member?

17) What are the limitations of static functions implementation?

18) What class friends and what they are?

19) Are class friends members of a class scope?

20) What is the use of operator overloading?

21) What operators cannot be overloaded?

22) When the operator should only overload using the member function?

23) When the operator should only overload using the global function?

24) How to overload type casting operator?

25) What is the purpose of exception handling mechanism?

26) How to create an exception object?

27) What types of exception objects can be used?

28) Is it possible to use the main function result if an exception was thrown?

29) How to catch and handle an exception?

30) How to create a block that handles all exceptions?

LABORATORY TRAINING 4
Using polymorphism and templates

1 Training tasks

1.1 The hierarchy of classes

Implement classes "Human," "Citizen", "Student", and "Employee". Each class must implement virtual function that shows related data on the screen. Create an array of pointers to different objects of the class hierarchy. Create a cycle and display data that represents objects of different types.

1.2 Using polymorphism for callback

Create a class for solving problem set in task 1.2 of the first laboratory training. The class should contain at least two member functions: a function that returns some value according to individual task, and pure virtual function that is called from the previous one and defines the left side of the equation or researched function (according to the task).

Class should be allocated in a separate header file. The relevant implementation file should contain the definition of non-abstract member functions.

Another translation unit should contain a class derived from the previous one. This class should contain the definition of the virtual function, which is a subject of investigation. Create an object of the derived class and implement the individual task in `main()` function.

Note: You should add some base class member functions that will calculate the first (second) derivative.

1.3 Using templates for callback

In a separate header file, create a template function for solving task 1.2 of the first laboratory training. The first parameter of the function must be an object of the template type to which the parentheses operation can be applied.

Check the operation of the template on two functions that are subject to research. One of the functions must be implemented as a functional object.

Note: To calculate the first (second) derivative, separate template functions should be added.

1.4 Class template for representation of two-dimensional array

Convert class created in the task 1.4 of the previous laboratory training into class template. Implement global template function that returns minimum array item. Create arrays of integers, real numbers, and simple fractions (previously created class) in `main()` function. For these three arrays you should test the function of finding the minimum value, as well as other class features with catching possible exceptions. You should also solve the problem from the individual task.

Note: in order to be able to find the minimum value in the array of fractions you should overload comparison operation for objects of the class "Simple fraction".

1.5 Library of template functions for working with an array (additional task)

Create a header file with functions that work with an array of an arbitrary generic type. The following functions should be implemented:

– exchange of places of elements with the specified indexes;

– search for an element with a certain value;

– exchange of places of all pairs of adjacent elements (with even and odd index).

Demonstrate the operation of all functions using at least three different types of data.

2 Guidelines

The goal of this laboratory training is obtaining practical skills in using polymorphism and templates in C++.

The following questions should be explored:

– using UML to represent classes;

– inheritance;

– multiple inheritance;

– hierarchies of exception classes;

– polymorphism;

– abstract classes;

– using templates.

Necessary theoretical concepts and syntax constructs are discussed in lectures on the topic “Inheritance and polymorphism. Templates”.

Most tasks can be implemented based on sample programs discussed below.

3 Sample programs

3.1 Hierarchy of real world objects

Suppose you want to develop the following class hierarchy: "Region" - "Populated region" - "Country". Certain classes of this hierarchy can be used as base classes for other classes (for instance, "Uninhabited island", "National park", "Borough", "Autonomous Republic", etc.). Class hierarchy can be expanded by classes "City" and "Island." It is advisable to add constructors that initialize all fields. We can also create an array of pointers to different objects of the hierarchy and display data that represents objects of different types.

We can offer such a class hierarchy, in which for greater clarity all functions are implemented within the classes::

```
#pragma warning(disable:4996)


#include <cstring>
#include <iostream>

using std::strcpy;
using std::cout;
using std::endl;

class Region
{
private:
    char name[30];
    double area;
public:
    Region(const char *name, double area)
    {
        strcpy(this->name, name);
```

```
        this->area = area;
    }
    char* getName()
    {
        return name;
    }
    double getArea() const
    {
        return area;
    }
    virtual void show()
    {
        cout << endl << "Name: " << name << ".\tArea: " << area << " sq.km.";
    }
    virtual ~Region() { }
};

class PopulatedRegion : public Region
{
private:
    int population;
public:
    PopulatedRegion(const char* name, double area, int population)
        : Region(name, area) { this->population = population; }
    int getPopulation() const
    {
        return population;
    }
    double density() const
    {
        return population / getArea();
    }
    void show() override
    {
        Region::show();
        cout << "\tPopulation:" << population << " inh.\tDensity: "
             << density() << " inh./sq.km.";
    }
};

class Country : public PopulatedRegion
{
private:
    char capital[20];
public:
    Country(const char* name, double area, int population, const char* capital)
        : PopulatedRegion(name, area, population)
                                            { strcpy(this->capital, capital); }
public:
    char* getCapital()
    {
        return capital;
    }
    void show() override
    {
        PopulatedRegion::show();
        cout << "\tCapital" << capital;
    }
};

class City : public PopulatedRegion
```

```
{
private:
    int boroughs;
public:
    City(const char* name, double area, int population, int boroughs)
        : PopulatedRegion(name, area, population) { this->boroughs = boroughs;
}
public:
    int getBoroughs() const
    {
        return boroughs;
    }
    void show() override
    {
        PopulatedRegion::show();
        cout << "\tBoroughs: " << boroughs;
    }
};

class Island : public PopulatedRegion
{
private:
    char sea[30];
public:
    Island(const char* name, double area, int population, const char* sea)
        : PopulatedRegion(name, area, population) { strcpy(this->sea, sea); }
    char* getSea()
    {
        return sea;
    }
    virtual void show() override
    {
        PopulatedRegion::show();
        cout << "\tSea: " << sea;
    }
};

void main()
{
    const int N = 4;
    Region *regions[N] = {
        new City("Kyiv", 839, 2679000, 10),
        new Country("Ukraine", 603700, 46294000, "Kyiv"),
        new City("Kharkiv", 310, 1461000, 9),
        new Island("Zmiyinyy", 0.2, 30, "Black Sea")
    };
    for (int i = 0; i < N; i++)
    {
        regions[i]->show();
    }
    for (int i = 0; i < N; i++)
    {
        delete regions[i];
    }
}
```

Note: the inclusion of nonstandard directive `#pragma warning(disable:4996)` in Visual Studio allows to avoid error messages concerned with the use of "dangerous" `strcpy()` function.

We use an array of pointers instead of array objects, because only in this case the mechanism of polymorphism can work. Otherwise the `show()` function would be invoked only from the base class.

3.2 Class for solving equation using Dichotomy method

An example of solving the problem of finding the root of the equation by dividing in half the segment that has been shown in previous laboratory work opportunities can be modified using polymorphism.

After a new empty project was created, we can add a new class, `AbstractDichotomy`. This can be done by using the main menu function `Project | Add Class...`. Enter the class name `AbstractDichotomy` and click `OK`. The pair of files is automatically generated: header file and implementation file. The `AbstractDichotomy.h` file contains the following code:

```
#pragma once
class AbstractDichotomy
{
};
```

The `AbstractDichotomy.cpp` file contains header file inclusion:

```
#include "AbstractDichotomy.h"
```

Generated text contains a non-standard preprocessor directive `#pragma once`. The use of this directive instead of the standard inclusion guards makes it impossible to compile this code in other environments than Visual Studio. We'll replace this directive with inclusion guards. Such replacement should be performed for each generated class. We'll declare the function of solving equation `root()` and pure virtual function `f()`:

```
#ifndef AbstractDichotomy_h
#define AbstractDichotomy_h

class AbstractDichotomy
{
public:
    double root(double a, double b, double eps = 0.001);
    virtual double f(double x) = 0;
};

#endif
```

The implementation file contains the definition of the `root()` function:

```
#include <cmath>
#include "AbstractDichotomy.h"

double AbstractDichotomy::root(double a, double b, double eps)
{
    double x;
    do
    {
        x = (a + b) / 2;
        if (f(a) * f(x) > 0)
        {
            a = x;
        }
        else
        {
            b = x;
        }
    }
    while (std::fabs(b - a) > eps);
    return x;
}
```

Now the program that needs solving equations should contain the inclusion of our header file and the definition of a new derived class. This class will define a function that represents a first member of equation:

```
#include <iostream>
#include "AbstractDichotomy.h"

using std::cout;
using std::endl;

class MyDichotomy : public AbstractDichotomy
{
    virtual double f(double x) override
    {
        return x * x - 2;
    }
};

void main()
{
    MyDichotomy d;
    cout << d.root(0, 6) << endl;           // 1.41431
    cout << d.root(0, 6, 0.00001) << endl;  // 1.41421
}
```

New classes should be created for each function.

3.3 Using templates to create a universal function

An alternative version of a universal solution for creating a universal root finding function is provided by templates.

We create a header file:

```
#ifndef DICHOTOMY_H
```

```
#define DICHOTOMY_H

template <typename F> double root(F f, double a, double b, double eps = 0.001)
{
    double x;
    do
    {
        x = (a + b) / 2;
        if (f(a) * f(x) > 0)
        {
            a = x;
        }
        else
        {
            b = x;
        }
    }
    while (b - a > eps);
    return x;
}

#endif
```

A separate implementation file is not required.

In the file with the `main()` function, we allocate the class for creating a functional object. In addition, we can create a function for which the root will also be found. We find roots of both functions in the `main()` function..

```
#include <iostream>
#include <cmath>
#include "Dichotomy.h"

// A class for creating a functional object: a polynomial of the Nth degree.
// The template parameter is an integer constant that can be applied
// to create an array.
template <int N> class Polynomial
{
private:
    double coefs[N + 1] = { 0 };
public:
    Polynomial(std::initializer_list<double> coefs)
    {
        int i = 0;
        for (const double& k : coefs)
        {
            if (i <= N)
            {
                this->coefs[i++] = k;
            }
        }
    }

    // Thanks to the parentheses operation overload
    // we can work with the object as with the function
    double operator()(double x)
    {
        double sum = 0;
        double p = 1;
```

```
        for (int i = N; i >= 0; i--)
        {
            sum += p * coefs[i];
            p *= x;
        }
        return sum;
    }
};

double my_sin(double x)
{
    return std::sin(x);
}

int main()
{
    std::cout << root(my_sin, 1, 4, 0.000001) << std::endl;
    Polynomial<2> poly = { 1, -1, -2 };
    std::cout << root(poly, 0, 3, 0.000001) << std::endl;
    for (double x = 0; x < 3; x += 0.25)
    {
        std::cout << x << "\t" << poly(x) << std::endl;
    }
    return 0;
}
```

Unfortunately, unlike using pointers to functions, a template-based solution does not allow you to apply standard functions. As in the above example, they must be "wrapped" with our functions.

3.4 Class template for presentation of one-dimensional array

The class that was created in the previous laboratory work can be converted into a class template so that we can create an array of different types. The class can be placed in a separate header file called `Array.h`. A template function called `getSum()`, which is located in the same file, allows us to find the sum of array items. The code of `Array.h` file will be as follows:

```
#ifndef Array_h
#define Array_h

#include <iostream>

using std::istream;
using std::ostream;

template <typename T> class Array
{
    friend ostream& operator<<(ostream& out, const Array& a)
    {
        for (int i = 0; i < a.size; i++)
        {
            out << a.pa[i] << ' ';
        }
```

```
        return out;
    }
    friend istream& operator >> (istream& in, Array& a)
    {
        for (int i = 0; i < a.size; i++)
        {
            in >> a.pa[i];
        }
        return in;
    }
private:
    T *pa = nullptr;
    int  size = 0;
public:
    class OutOfBounds
    {
        int index;
    public:
        OutOfBounds(int i) : index(i) { }
        int getIndex() const { return index; }
    };
    Array() { }
    Array(int n);
    Array(Array& arr);
    ~Array()
    {
        if (pa)
        {
            delete[] pa;
        }
    }
    void addElem(T elem);
    T& operator[](int index);
    const Array& operator=(const Array& a);
    bool operator==(const Array& a) const;
    int getSize() const { return size; }
};

template <typename T> Array<T>::Array(int n)
{
    pa = new T[size = n];
}

template <typename T> Array<T>::Array(Array& arr)
{
    size = arr.size;
    pa = new T[size];
    for (int i = 0; i < size; i++)
    {
        pa[i] = arr.pa[i];
    }
}

template <typename T> void Array<T>::addElem(T elem)
{
    T *temp = new T[size + 1];
    if (pa)
    {
        for (int i = 0; i < size; i++)
        {
            temp[i] = pa[i];
```

```
        }
        delete[] pa;
    }
    pa = temp;
    pa[size] = elem;
    size++;
}

template <typename T> T& Array<T>::operator[](int index)
{
    if (index < 0 || index >= size)
    {
        throw OutOfBounds(index);
    }
    return pa[index];
}

template <typename T> const Array<T>& Array<T>::operator=(const Array<T>& a)
{
    if (&a != this)
    {
        if (pa)
        {
            delete[] pa;
        }
        size = a.size;
        pa = new T[size];
        for (int i = 0; i < size; i++)
        {
            pa[i] = a.pa[i];
        }
    }
    return *this;
}

template <typename T> bool Array<T>::operator==(const Array<T>& a) const
{
    if (&a == this)
    {
        return true;
    }
    if (size != a.size)
    {
        return false;
    }
    for (int i = 0; i < size; i++)
    {
        if (pa[i] != a.pa[i])
        {
            return false;
        }
    }
    return true;
}


template <typename T> T getSum(Array<T>& a)
{
    T sum = T();
    for (int i = 0; i < a.getSize(); i++)
    {
```

```
        sum = sum + a[i];
    }
    return sum;
}

#endif
```

To perform testing of our class for different types, we can create `Vector.h` header file and copy source code of the class `Vector` previous laboratory into this header file:

```
#ifndef Vector_h
#define Vector_h

#include <iostream>

using std::istream;
using std::ostream;

class Vector {
    friend istream& operator >> (istream& in, Vector& v)
                                                { return in >> v.x >> v.y; }
    friend ostream& operator<<(ostream& out, const Vector& v)
                                { return out << "x=" << v.x << " y=" << v.y; }
    friend Vector operator+(Vector v1, Vector v2)
                                  { return Vector(v1.x + v2.x, v1.y + v2.y); }
    friend Vector operator*(double k, Vector v)
                                           { return Vector(v.x * k, v.y * k); }
    friend Vector operator*(Vector v, double k) { return operator*(k, v); }
    friend double operator*(Vector v1, Vector v2)
                                         { return v1.x * v2.x + v1.y * v2.y; }
private:
    double x, y;
public:
    Vector() { x = y = 0; }
    Vector(double x, double y) { this->x = x; this->y = y; }
    double getX() { return x; }
    void setX(double x) { this->x = x; }
    double getY() { return y; }
    void setY(double y) { this->y = y; }
};

#endif
```

Now you can use the template array class to store as integers and vectors:

```
#include <iostream>
#include "Array.h"
#include "Vector.h"

using std::cout;
using std::endl;

void main()
{
    Array<int> intArray;
    intArray.addElem(11);
    intArray.addElem(12);
    cout << intArray << endl;
```

```
    try
    {
        intArray[1] = 4;
        intArray[10] = 35;
    }
    catch (Array<int>::OutOfBounds e)
    {
        cout << "Bad index: " << e.getIndex() << endl;
    }
    cout << getSum(intArray) << endl;
    Array<Vector> vectorArray;
    vectorArray.addElem(Vector(1, 2));
    vectorArray.addElem(Vector(3, 4));
    vectorArray.addElem(Vector(5, 6));
    cout << vectorArray << endl;
    cout << getSum(vectorArray) << endl; // x=9 y=12
}
```

You can also store items of other data types, such as pointers to integer:

```
int m = 1, n = 2;
// Create an array of pointers to int
Array<int*> pointerArray;
pointerArray.addElem(&m);
pointerArray.addElem(&n);
// Output dereferenced values:
cout << *pointerArray[0] << " " << *pointerArray[1] << endl; // 1 2
```

But trying to find the sum of items will result in compilation error because you cannot find the sum of pointers:

```
// You cannot find the sum of pointers:
cout << getSum(pointerArray) << endl; // Compile error (cannot add two
pointers)
```

4 Quiz

1) How to use UML to represent a class?

2) What connections between classes does UML offer and how are they implemented in C++?

3) What is the concept of inheritance?

4) What are differences between public, protected, and private inheritance?

5) What are the advantages and disadvantages of multiple inheritance?

6) What is the usage of virtual base classes?

7) What is the usage of a hierarchy of exceptions?

8) What is the concept of polymorphism?

9) What is the difference between compile time polymorphism and runtime polymorphism?

10) What is a virtual function?

11) What classes are considered to be polymorphic?

12) Why polymorphic classes require definition of virtual destructors?

13) What is a virtual method table?

14) What is pure virtual function?

15) What is an abstract class?

16) What is the use of template functions?

17) Is it possible to implement template function separately for certain type?

18) When you need to specify explicitly a template parameter of template functions?

19) How to create a class template?

20) What are the template parameters?

21) What is the use of integer template parameters?

22) What is template instantiation?

23) What are rules of type conversion for objects of instantiated types?

LABORATORY TRAINING 5

Use of the Standard template library

1 Training tasks

1.1 Presentation and processing of data about students using the Standard template library

Complete task 1.3 of the third laboratory training (Classes for Representing Students and Groups).

In the "Student" class, use a `std::string` of type to store the last name. Store grades for the last session in a vector of integers `std::vector<int>`). In addition to the functions listed in task 1.3 of the second laboratory training, you should implement functions that perform:

– calculating the value that is used for sorting according to individual task;

– test conditions used for searching data according to individual task.

In the "Group of students" class, place a vector of objects that represents a student. Implement the functions defined in task 1.3 of the second laboratory training for this class. To sort the array according to the criteria specified in the individual task, use the `sort()` algorithm. To find data about students that meet the condition given in the individual task, apply the `for_each()` algorithm.

Additionally, place objects of" "Student" class into priority queue and obtain objects in descending order of the average score.

1.2 The vector of vectors for representation of two-dimensional array

Complete task 1.4 of the previous laboratory training (Class for Representing a Two-Dimensional Array) by creating a class whose field is a vector of vectors of the Standard template library. The created class does not require a copy constructor, destructor and overloaded assignment operator, because data is not placed into free store (vectors are responsible for this).

1.3 Counting the number of repetitions of values (additional task)

Develop a program in which integer values are read and the number of repetitions of each value is counted, except for the numbers given in some list.

Counting the repetitions of a number should be implemented using map, the list of numbers for exclusion should be implemented using a set. Each time a number from th list is encountered, output message.

2 Guidelines

The goal of this laboratory training is obtaining practical skills in use of the Standard template library (STL).

– the following questions should be explored:

– the overall structure of the Standard C++ library;

– standard sequential containers;

– associative arrays and sets;

– algorithms of the standard library;

– function objects; predicate functions; function adaptors.

Necessary theoretical concepts and syntax constructs are discussed in lectures on the topic “Using the C++ Standard library, lambda expressions and modules”.

Most tasks can be implemented based on sample programs discussed below.

3 Sample Programs

3.1 The use of vectors

The following program creates vector of vectors of integers whose values are displayed:

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

using std::cout;
using std::vector;

void main()
{
    vector<vector<int> > a = { vector<int>({ 1, 2 }), vector<int>({ 3, 4 }) };
    for (auto &row : a)
    {
        for (auto &x : row)
        {
            cout << x << "\t";
        }
```

```
        cout << "\n";
    }
}
```

3.2 Work with priority queue

Suppose there is a class for representation of the country:

```
#include <iostream>
#include <queue>
#include <string>

using std::string;

class Country
{
private:
    string name;
    double area;
    long   population;
public:
    Country() : name(""), area(1), population(0) { }
    Country(string n, double s, long p) : name(n), area(s), population(p) { }
    string getName()       const { return name; }
    double getArea()       const { return area; }
    long   getPopulation() const { return population; }
    void   setName(string name)  { this->name = name; }
    void   setArea(double area)  { this->area = area; }
    void   setPopulation(long population) { this->population = population; }
    double density() const { return population / area; }
};
```

Now we can create priority queue, which will contain objects of `Country` type. Countries with greater population density will be obtained first. In order to allocate items of `Country` type in the priority queue, it is necessary for overload < operation. This operator can be overloaded by external function. This function may not even be a friend of the `Country` class:

```
bool operator<(const Country& c1, const Country& c2)
{
    return c1.density() < c2.density();
}
```

Now objects can be placed into a priority queue:

```
using std::priority_queue;
using std::cout;
using std::endl;

int main()
{
    Country c0("Ukraine", 603700, 42539000);
    Country c1("France", 544000, 57804000);
    Country c2("Sweden", 450000, 8745000);
    Country c3("Germany", 357000, 81338000);
    priority_queue<Country> cq;
```

```
    cq.push(c0);
    cq.push(c1);
    cq.push(c2);
    cq.push(c3);
    while (cq.size())
    {
        cout << cq.top().getName() << endl;
        cq.pop();
    }
    return 0;
}
```

An alternative to the use of overloaded operator is the use of functional object: the object of class has overloads operator `()`. The name of this class should be specified as the third parameter of the `priority_queue` constructor:

```
class Less
{
public:
    bool operator()(const Country& c1, const Country& c2) const
    {
        return c1.density() < c2.density();
    }
};

int main()
{
    . . .
    priority_queue<Country, std::vector<Country>, Less> cq;
    . . .
}
```

3.3 Working with map

Suppose we want to read integer values from the file "`values.txt`", count the number of each of the values and display the values and counts of repetitions.

```
#include <iostream>
#include <fstream>
#include <map>

using std::map;
using std::ifstream;
using std::cout;
using std::endl;

int main()
{
    map<int, int> m;
    int i;
    {
        ifstream in("values.txt");
        while (in >> i)
        {
            m[i]++;
        }
    }
```

```
    map<int, int>::iterator mi;
    for (mi = m.begin(); mi != m.end(); mi++)
    {
        cout << mi->first << " " << mi->second << endl;
    }
    return 0;
}
```

As the development of this example is to offer the use of a functional object that specifies the reverse sort logic.

3.4 Using the Standard template library to represent cities and countries

In the example of the Laboratory training #3, we gave an example of creating classes to represent a city and a country. Now, using the Standard Template Library, we can improve the solution. We create a new version of the class `City` by first declaring the class `Country`:

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

using std::string;

// Class declaration&
class Country;

// Class to represent the city
class City
{
    // Overloaded operator to output to a stream:
    friend std::ostream& operator<<(std::ostream& out, const City& city);
private:
    string name = nullptr;     // city name
    Country *country = nullptr;// pointer to the location country
    string region = "";        // the name of the region
    int population = 0;        // population
public:
    // Constructors:
    City() { }
    City(const char* name, const char* region, int population) :
        name(name), country(country), region(region), population(population) {
}

    // Getters:
    string getName() const { return name; }
    Country* getCountry() const { return country; }
    string getRegion() const { return region; }
    int getPopulation() const { return population; }

    // Setters:
    void setName(string name) { this->name = name; };
    void setName(const char* name) { this->name = name; };
    void setRegion(string region) { this->region = region; };
    void setRegion(const char* region) { this->region = region; };
    void setCountry(Country* country) { this->country = country; }
```

```
    void setPopulation(int population) { this->population = population; }
};
```

We create a new version of the `Country` class:

```
class Country
{
    // Overloaded operator to output to a stream:
    friend std::ostream& operator<<(std::ostream& out, const Country& country);
private:
    string name;                    // name of country
    std::vector<City> cities = { }; // vector of cities
public:
    // Constructors:
    Country() { }
    Country(string name) : name(name) { }
    Country(const char* name) : name(name) { }

    string getName() const { return name; } // getter

    // Overloaded operator for getting array items
    City operator[](int index) const { return cities[index]; }

    // Setters:
    void setName(string name) { this->name = name; }
    void setName(const char* name) { this->name = name; }

    void addCity(City city); // Adding a city
    void sortByPopulation(); // Sort by population
};
```

We implement the previously declared functions:

```
// Overloaded operator to output to a stream:
std::ostream& operator<<(std::ostream& out, const City& city)
{
    out << "City: " << city.name << "\t Country: " << city.country->getName()
        << "\t Region: " << city.region << "\t Population: " <<
city.population;
    return out;
}

// Overloaded operator to output to a stream:
std::ostream& operator<<(std::ostream& out, const Country& country)
{
    out << country.name << std::endl;
    for (const City& city : country.cities)
    {
        out << city << std::endl;
    }
    out << std::endl;
    return out;
}

void Country::addCity(City city) // Adding a city
{
    city.setCountry(this);
    cities.push_back(std::move(city));
}

bool byPopulation(City c1, City c2) // Define the sorting criterion
```

```
{
    return c1.getPopulation() < c2.getPopulation();
};

void Country::sortByPopulation() // Sort by population
{
    std::sort(cities.begin(), cities.end(), byPopulation);
}
```

In the `main()` function, we create a country object, add cities, and demonstrate the implemented capabilities of the classes:

```
int main()
{
    std::system("chcp 1251 > nul");
    Country country = "Ukraine"; // create the "Country" object",
                                 // call the constructor with the single
parameter
    // Create and add cities:
    country.addCity(City("Kharkiv", "Kharkiv region", 1421125));
    country.addCity(City("Poltava", "Poltava region", 284942));
    country.addCity(City("Lozova", "Kharkiv region", 54618));
    country.addCity(City("Sumy", "Sumy region", 264753));

    std::cout << country << std::endl;   // output all data
    std::cout << country[0] << std::endl;// display information about the city
by index
    std::cout << std::endl;
    country.sortByPopulation();          // sort
    std::cout << country << std::endl;   // output all data

    return 0;
}
```

As can be seen from the implementation, we use a global function `byPopulation()` to determine the sorting criterion.

3.5 Generic class for representing a one-dimensional array

The class created in the previous lab can be implemented using a Standard Library vector. The practical meaning of such an implementation may arise when a certain class has been previously implemented, and it is not desirable to change the object type. Without changing the public part of the previously created `Array` class, an alternative implementation can be proposed.

The class can be placed in a separate header file `Array.h`:

```
#ifndef Array_h
#define Array_h

#include <iostream>
#include <vector>

using std::istream;
```

```
using std::ostream;

template <typename T> class Array
{
    friend ostream& operator<<(ostream& out, const Array& a)
    {
        for (int i = 0; i < a.pa.size(); i++)
        {
            out << a.pa[i] << ' ';
        }
        return out;
    }
    friend istream& operator >> (istream& in, Array& a)
    {
        for (int i = 0; i < a.pa.size(); i++)
        {
            in >> a.pa[i];
        }
        return in;
    }
private:
    std::vector<T> pa;
public:
    class OutOfBounds
    {
        int index;
    public:
        OutOfBounds(int i) : index(i) { }
        int getIndex() const { return index; }
    };
    Array() { }
    Array(int n);
    void addElem(T elem) { pa.push_back(elem); }
    T& operator[](int index);
    int getSize() const { return pa.size(); }
};

template <typename T> Array<T>::Array(int n)
{
    pa.resize(n);
}

template <typename T> T& Array<T>::operator[](int index)
{
    try
    {
        return pa.at(index);
    }
    catch (...)
    {
        throw OutOfBounds(index);
    }
}

template <typename T> T getSum(Array<T>& a)
{
    T sum = T();
    for (int i = 0; i < a.getSize(); i++)
    {
        sum = sum + a[i];
    }
```

```
    return sum;
}

#endif
```

The function `main()` can demonstrate the capabilities of the class. The usage `Array` has practically not changed:

```
#include <iostream>
#include "Array.h"

using std::cin;
using std::cout;
using std::endl;

int main()
{
    Array<int> intArray(2);
    cin >> intArray;
    intArray.addElem(11);
    intArray.addElem(12);
    cout << intArray << endl;
    try
    {
        intArray[0] = 4;
        intArray[1] = 5;
        intArray[10] = 35;
    }
    catch (Array<int>::OutOfBounds e)
    {
        cout << "Bad index: " << e.getIndex() << endl;
    }
    cout << intArray << endl; // 4 5 11 12
    cout << getSum(intArray) << endl; // 32
    return 0;
}
```

It is also possible to demonstrate the class's work on other data types.

4 Quiz

1) What are the main elements of the Standard C++ Library?

2) What is a Standard Library container?

3) What is the difference between sequential and associative containers?

4) What is iterator?

5) What are the requirements for the iterators of the Standard Library?

6) What are the advantages vectors compared with arrays?

7) How to create a `vector` object?

8) How to work with individual items of vectors?

9) What is the difference between the list and the array?

10) How the character strings are represented in the Standard Library?

11) What is a sequence adapter?

12) What is the difference between a queue and a stack?

13) What is associative array?

14) What is the difference between map and multimap?

15) Where are associative arrays used?

16) What sets are different from other containers?

17) What are the standard classes for representing a set?

18) What is the Standard Library algorithm?

19) What are groups of algorithms?

20) How does the algorithm relate to the type of container to which it is applied?

21) What is function object?

22) What is predicate function? What are the standard predicate functions?

23) What are function adaptors?

LIST OF SOURCES

Recommended reading:

1 Stroustrup B. The C++ Programming Language: 4th Edition, Addison-Wesley, 2013, 1368 p.

2 Lippman S. B., Lajoie J., Moo B. E. C++ Primer: 6th Edition, Addison-Wesley Professional, 2011. 992 p.

3 Schildt H. C++: The Complete Reference: 4th Edition, McGraw-Hill Education, 2002, 1056 p.

4 Schildt H. Java: A Beginner's Guide: 8th Edition, McGraw-Hill Education, 2018, 684 p.

Internet resources

1 International Standard ISO/IEC 14882:2014(E) – Programming Language C++ // https://isocpp.org/std/the-standard

2 C/C++ language and standard libraries reference // https://msdn.microsoft.com/en-us/library/hh875057.aspx

3 The C++ Programming Language (Bjarne Stroustrup's homepage) // http://www2.research.att.com/~bs/C++.html

4 ISO/IEC 14882:2003 Programming languages - C++ (International Standard) // http://cs.nyu.edu/courses/summer12/CSCI-GA.2110-001/downloads /C++%20Standard%202003.pdf

5 The C++ Resources Network // http://www.cplusplus.com/

6 The C++ Tutorial // http://www.learncpp.com/

7 C++ Tutorial - W3Schools // https://www.w3schools.com/cpp/

8 C++ Tutorial - Tutorialspoint // https://www.tutorialspoint.com /cplusplus /index.htm

9 C++ - Wikipedia, the free encyclopedia // http://en.wikipedia.org /wiki/C++

10 C++ - Wikiversity// https://en.wikiversity.org/wiki/C++

Educational edition

GUIDELINES

to the laboratory work for the courses “Fundamentals of Programming” and “Algorithmization and Programming” for students of specialties 121 “Software engineering” and 122 “Computer Science”

Author:
Ivanov Lev Vadymovych

Responsible for the issue Assoc. Prof. Kopp A.M.
The paper has been recommended for publication by Prof. Hamaiun I.P.

In the author's edition

Plan of 2025 year, position 160

Signed for publication on 13.02.2025. Headset Times New Roman.

Publishing Centre of NTU "KhPI
Certificate of state registration DK № 5478 of 21.08.2017

Electronic publication