MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

NATIONAL TECHNICAL UNIVERSITY OF UKRAINE
“IGOR SIKORSKY KYIV POLYTECHNIC INSTITUTE”
DEPARTMENT OF BIOMEDICAL ENGINEERING

**Danilova V. A.**

# OBJECT-ORIENTED PROGRAMMING.
## WORKSHOP

*Workshop on discipline for students of specialties 163 "Biomedical Engineering"*

Kyiv
Igor Sikorsky Kyiv Polytechnic Institute
2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

**В.А. Данілова**

# ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ. ПРАКТИКУМ

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського*
*як навчальний посібник для здобувачів ступеня бакалавра*
*за спеціальністю 163 «Біомедична інженерія»*

Київ
КПІ ім. Ігоря Сікорського
2021

| | |
|---|---|
| Рецензент | *Богомолов М.Ф.,* к.т.н., доцент кафедри БМІ КПІ ім. Ігоря Сікорського,<br>*Дубко А.Г.,* к.т.н., доцент, наук. співроб. відд. зварювання та споріднених технологій в медицині та екології Інституту електрозварювання ім.Є.О.Патона |
| Відповідальний редактор | *Зубчук В.І.*, к.т.н., доц., доцент кафедри біомедичної інженерії КПІ ім. Ігоря Сікорського |

*Гриф надано Методичною радою КПІ ім. Ігоря Сікорського*
*(протокол № 1 від 16.09.2021 р.)*
*за поданням Вченої ради факультету біомедичної інженерії*
*(протокол № 16 від 30.08.2021 р.)*

*Данілова Валентина Анатоліївна, старший викладач*

# ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ. ПРАКТИКУМ

«Об'єктно-орієнтоване програмування. Практикум»: навч. посіб. для студентів спеціальності 163 «Біомедична інженерія» / КПІ ім. Ігоря Сікорського; уклад. В.А. Данілова.– КПІ ім. Ігоря Сікорського.- 2021. – 105 с.

Навчальний посібник розроблено для отримання студентами практичних навичок з програмування мовою С++. Навчальне видання призначене для студентів, які навчаються за спеціальністю 163 – «Біомедична інженерія» факультету біомедичної інженерії КПІ ім. Ігоря Сікорського.

Object-Oriented Programming. Workshop: workshop on discipline for students of specialties 163 «Biomedical Engineering» / V.A. Danilova.– Kyiv: Igor Sikorsky Kyiv Polytechnic Institute, 2021. – 105 p.

Educational and methodical edition

# OBJECT-ORIENTED PROGRAMMING. WORKSHOP

Workshop on discipline for students of specialties 163 "Biomedical Engineering"

Author: ***Valentyna Danilova***, Senior Lecturer, Department of Biomedical Engineering

Editor-in-Chief: *Viktor Zubchuk*, Ph.D., Associate Professor

Reviewers: *M.F.Bogomolov*, Ph.D., Associate Professor,
*A.G Dubko*, Ph.D., Associate Professor,
Researcher of Department of Welding and Related Technologies in Medicine and Ecology

*Edited by the authors*

# CONTENT

# INTRODUCTION

The main task of this workshop is to teach students to develop programs in C ++ using structural programming technology. It's no secret that over the last few decades, the software market has undergone significant changes that are obvious even to a non-professional. In particular, the C ++ language has become a universal high-level programming language with support for several modern programming paradigms: object-oriented, generalized and procedural.

The course "Object-Oriented Programming" is designed to master the technique of object-oriented programming based on the C ++ programming language.

The course introduces students to modern methods and principles of PC software development: Windows operating system, integrated Visual Studio.NET development environment, object-oriented programming and C ++.

In this guide, the study of programming technology begins with the simplest examples of program development, gradually complicating them, and ending with full-fledged structural programs.

The guidelines include a number of practical works during which students have the opportunity to gain experience working with the integrated development environment Visual Studio .NET.

Before performing practical work, students must: read the guidelines; repeat lecture material related to practical work; prepare answers to the questions given in the guidelines at the end of each practical work.

After completing these tasks, the student must demonstrate to the teacher the work on the computer, draw up a report on the results of this practical work, defend it and pass it to the teacher.

# PRACTICAL WORK №1

# Topic: Work in the integrated development environment of Microsoft Visual Studio

**Purpose:** formation of skills of the organization of work in the environment of Microsoft Visual Studio at writing and debugging of programs on C ++.

**Theoretical information**

**Microsoft Visual Studio** is a single-shell integrated tool that allows you to create, open, view, edit, save, compile, debug, and run C ++ programs. Any program created in the Microsoft Visual Studio environment is designed as a separate project.

**A project** is a set of interconnected source files designed to solve a specific problem. Their compilation and layout allows you to get a ready-to-run program. The project includes both files created directly by the programmer and files created and edited by the environment itself.

After starting Microsoft Visual Studio , the main program window opens, shown in Figure 1.1 (depending on the settings, its appearance may differ slightly from the above).

The Microsoft Visual Studio desktop consists of three windows:

1. *Project Workspace* window, which displays a list of current project files;

2. the editor ( *Editor* ) to enter and edit the source code in C ++;

3. Output window ( *Output* ), which displays messages about the progress of compilation and layout of the program. In particular, this window displays all error messages at the stages of compiling and compiling the program.

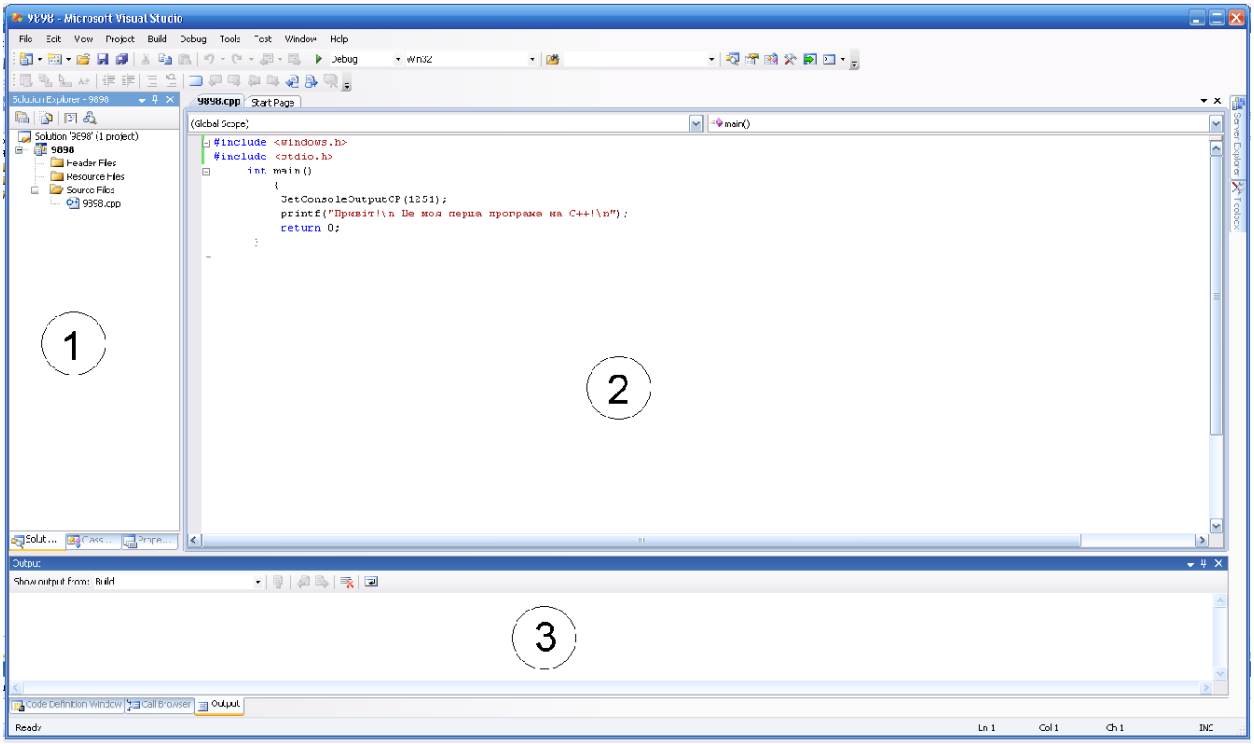


Fig.1.1 View of the Microsoft Visual Studio environment window after creating a new project.

The environment allows you to build both projects - *Windows applications* with a graphical interface, and projects - *console applications* .

**A console** is an interface used by a program that works in text mode. The program has input (connected to the keyboard) and output (connected to the monitor screen) buffers and interacts with Windows through a console, which creates a window whose properties are similar to a normal Windows window. The project - console application is the most convenient for learning the C ++ programming language.

To create a new project such as a *console application*, *you* must perform the following steps:

- in the menu bar of the main window of Microsoft Visual Studio select the command *File-New-Projects* ;

- in the *New Projects* dialog box, select the project type: *Visual C ++ - Win32* ;

- specify the type of the created project - *Win32 Console Application* ;

- in the field *Project Name* enter the project name (for example *Lab1* );

- in the *Locations* field, select the directory where it will be stored ( *Browse* dialog box );

- click the *OK* button ;

- in the *Win32 Console Application* application wizard window, select the *Application Settings* menu item and specify an empty project ( *Empty project* );

- click on the *Finish* button .

As a result, an empty console project will be created. It is necessary to add a new or existing file with the text of the program in C ++. To add a new file to the project, you can use the command *Project - Add New Item* : in the *Name* field enter the file name (for convenience, it is desirable to specify a name that matches the name of the project itself) and click the *Add* button .

In order to add an existing file with the program in C ++, you must first copy it to the working folder of the project, use the command *Project - Add Existing Item* and select the appropriate file.

A project folder can contain multiple files and two subfolders.

**Files and folders are intended for:**

- file with the extension *.dsw* - a project file that combines all the files that are part of the project (it may not be);

- file with *.dsp* extension - to build a separate project or subproject;

- file with *.opt* extension - contains all the settings of this project;

- file with extension *.ncb* - service file;

- *Debug* - a folder that contains files created at the stage of compiling and compiling the program; executable file with *.exe* extension (if the program was compiled and compiled successfully);

- The folder with the name of the program in C ++ includes a file with the extension *.cpp* - the text file of the program in C ++.

You can compile and compile a project (programs in C ++) using the menu in the main *Build* window or using the appropriate toolbar buttons.

**The main commands of the *Build* menu are:**

- *Compile* ( *Ctrl + F7* ) - compile a file with the program C ++, which is part of the project. Compilation error messages are displayed in the *Output* window. If no compilation errors are detected, an object file with the *.obj* extension will be created;

- *Build Solution* ( *F7* ) - project layout. All files that have changed since the last layout are compiled. After compiling runs arrangement ( *link* ) all object modules, including the library and create an executable file. Error messages during the layout phase are displayed in the *Output* window. If both phases completed without errors, it will create an executable file with the extension *.ehe* that can be run to perform. You can start the project by using the menu of the main *Debug* window (or by using the appropriate toolbar buttons):

- *Start Without Debugging* ( *Ctrl + F5* ) - start the executable file created as a result of the project layout. As a result, a console window will open to work with the program. To open a project you've worked on before, choose *File-Open-Project / Solution* , find the folder with your project, and select the file with the *.dsw* extension .

## Alphabet and language syntax

In the natural language of communication there are four main elements: symbol, word, phrase and sentence. Similar elements exist in algorithmic language, only words are called tokens, phrases - expressions, and sentences - operators. Tokens are created from symbols, expressions - from tokens and symbols, operators - from symbols, expressions and tokens.

**The C ++ alphabet includes:**

- uppercase (AZ) and lowercase (a-z) letters of the Latin alphabet and the underscore ( _ );
- Arabic numerals from 0 to 9;
- signs of arithmetic operations +, -, *, /,%, ++, -;
- signs of bitwise operations <<, >>, &, |, ~, ^;
- signs of relations <, <=, ==,! =,>,> =;
- signs of logical operations &&, ||,!;

- punctuation ,; : pass;
- special characters ., =, ->,?, \, $, #, ', “;
- parentheses (,), [,], {,}.

Other symbols, as well as Cyrillic letters, are not used to construct the basic elements of the language or for their section, but they can be used in symbolic constants and comments.

**Tokens** , ie the basic elements of language with a certain independent meaning, consist of symbols of the alphabet. These include identifiers, keywords, transaction signs, constants, delimiters (parentheses, semicolons, commas, spaces). The boundaries of tokens are determined by other token-separators or signs of operations.

***An identifier*** , that is, the name of a program object, is any sequence of letters of the Latin alphabet, numbers, and underscores, provided that the first is a letter or underscore, not a number.

There are two types of identifiers:

- *standard* , for example, the names of all functions built into the language;
- *custom* .

Characteristically, the C ++ language is case sensitive, so the compiler recognizes uppercase and lowercase letters of the Latin alphabet as different characters. This makes it possible to create identifiers that are the same read, but differ in the inscription of one or more characters. For example, the identifiers "Sigma", "sigma" and "sigMa" are considered different.

Identifiers can be of any length, but no more than 31 characters from the beginning of the identifier are significant, and in some compilers this restriction is even more stringent (no more than 8 characters). The names of program objects are created at the stage of data declaration, after which they can be used in various operators of the program.

***Keywords*** (service) are a number of reserved identifiers that are used to construct language structures and have a fixed meaning. According to the semantic load, service words are divided into the following main groups:

- type specifiers - char, int, long, typedef, short, float, double, enum, struct, union, signed, unsigned, void;
- qualifiers of types - const;
- memory classes - auto, extern, register, static;
- for construction of operators - for, while, do, if, else, switch, case, continue, goto, break, return, default, sizeof.

In the table. 1.1 shows a list of C ++ keywords.

Table 1.1

**Reserved Keywords**

| | | | | |
|---|---|---|---|---|
| **asm** | **delete** | **goto** | **register** | **throw** |
| **auto** | **do** | **if** | **return** | **try** |
| **break** | **double** | **inline** | **short** | **typedef** |
| **case** | **else** | **int** | **signed** | **typename** |
| **catch** | **enum** | **long** | **sizeof** | **union** |
| **char** | **explicit** | **new** | **static** | **unsigned** |
| **class** | **extern** | **operator** | **struct** | **virtual** |
| **const** | **float** | **private** | **switch** | **void** |
| **continue** | **for** | **protected** | **template** | **volatile** |
| **default** | **friend** | **public** | **this** | **while** |

The following symbols are used as token separators: space, tab, newline character, comment. Any number of delimiters is allowed between any two tokens. In addition, some tokens ( "*", "+", ",", "", "" ("," -> " , etc.) are themselves separators and it is not necessary to separate them from other tokens by delimiters.

## Tasks

1. Start the Microsoft Visual Studio integrated development environment and create a new console application. Save the project to disk in your folder. Add a new file to the project and enter the program in C ++:

*#include <iostream>*

```
int main(){
cout<< “Hello”;
return 0;
}
#include <iostream>
int main(){
cout<< “Hello”;
cout<<endl;
return 0;
}
```

2. Compile the entered program by setting the appropriate command. If the program is typed correctly, the compilation output window will indicate that there are no errors (s) and warnings (warnings (s) ) and that an executable file has been created ( Build: 0 succeeded, 0 failed, 1 up-to-date, 0 skipped). If mistakes were made while typing the program, the corresponding messages will be issued. If the output window (Output) to select a message and press the Enter (or double click the left mouse button), then the window line of the error will be marked marker and the cursor will be placed in this line. The error message is followed by the error number and a brief description. The compiler may not always be able to accurately locate the error position, especially if parentheses or semicolons are omitted - the error may be slightly higher than the specified location. Warnings (s) are usually not critical, and the executable file can be created and executed (preferably not).

3. Let's modify our program to look like:

```
#include <iostream.h>

#include <conio.h>

int main()
{
char name[20];
```

```
cout<< "What is your name:\n";
cin >> name;
cout<< "Hello: " <<name << endl;
getch();
}
```

#include <conio.h> is a directive for working with the screen

getch () function to delay the screen before pressing any key.

After starting, it should display the question *What is your name:*, we should enter the name, for example *Ivan* and press Enter, the program should display: *Hello: Ivan.*

4. Write in the report a modified program with a comment next to each line of the program.

For example:

# include <iostream.h> // connection libraries for input-output

**Control questions:**

1. What is the alphabet of C ++?
2. What reserved keywords does the C ++ language use?

# PRACTICAL WORK №2

# Topic: Input and output of information in C ++. Streaming operations of the C ++ language

**Purpose:** Formation of skills and abilities of the organization of operations of format input-output of information by means of C ++.

**Theoretical information**

The main parts of the typical structure of the program in C ++ are as follows:

- preprocessor processing directives;
- description of external variables (source data and results) and functions;
- program functions;
- main function - the program **main ()** , which has the form:

```
main ()
{
description of variables;
executive operators;
}
```

The program always starts with preprocessor directives (***#)*** , which are processed by it before compilation. The task of the preprocessor is to supplement the text of the program with library functions or C ++ objects, described in the corresponding blank (*header*) files. The most frequently used header files and their purpose are given in Appendix №1.

The program in C++ consists of a set of separate functions. The ***main ()*** function is mandatory - the entry point of the program (its location in the program does not matter):

```
# preprocessor directives
. . .
# preprocessor directives
```

*function a ()*

*{*

*function body a*

*}*

*function b ()*

*{*

*function body b*

*}*

*. . .*

*void main ()*

*/ * function to start with*

*program execution. * /*

*// void - does not return any values*

*{*

*body function*

*}*

The text between the parentheses **/ *** and ***** **/** is a comment to the program and is ignored at the compilation stage. The text following **//** is a one-line comment (to the end of this line). Such comments can be used anywhere in the program to "document" it.

The syntax for declaring variables is as follows:

**<*type*> <*name*> = <*value*>**

where:

**<*type*>** - variable type (one of the basic);

**<*name*>** - variable identifier;

**<*value*>** - the value that will be assigned to the variable after its identification (optional).

Example:

*int a;*

*unsigned int b = 2300;*

*double x, y;*

*double N_Avag = 6.022045e23;*

*char c1 = 'a', c2 = 'A';*

*int c3 = 5 + 5 * 2;*

*const float pi = 3.14l5926;*

The ***const*** keyword indicates a value that does not change its value during program execution. Each variable has a certain scope. If a variable is declared outside of any function, it is called global and can be used (is visible) anywhere in the program. If a variable is declared in a program block (or some function), it is local and can be used (visible) only within the given block (function).

C ++ does not have built-in tools for I/O. They are performed using I/O functions, the description of which is contained in the header files ***stdio.h*** and ***iostream***, which are connected at the beginning of the program using the preprocessor directive ***#include*** .

When using the ***stdio.h*** file (C-style I/O), the function is used for input

***scanf (<format list>, <input list>)***

where:

***scanf*** - the name of the format input function;

**<*list of formats*>** - specifiers that specify the format of data entry of a particular type (see Appendix №3);

**<*input list*>** - comma addresses of variables ( ***&*** **<*input list*>** ) into which data from the keyboard is entered. The number of format specifiers and their type must match the number and type of variable addresses whose values must be entered. The **&** symbol means a unary operation to obtain the address of a variable. Not used to enter values for string variables **&** .

Input of symbolic data from the keyboard is carried out by means of the following functions:

***getchar ()*** - with the display of the entered character on the screen;

***getch ()*** - without displaying the entered character on the screen.

Example:

```
#include <stdio.h> // file connection
void main ()
{
int k, m; // description of variables
scanf ("% d% d", & k, & m); // data entry
}
```

Data output is performed using the ***printf*** function :

***printf (<format list>, <output list>)***

The ***printf*** function uses the same formats as the **<*format list*>** as the ***scanf*** input function (see Appendix №3). Additionally, you can use the following control characters (escape characters):

\ ***n*** - go to a new line;

\ ***r*** - return to the beginning of the line;

\ ***v*** - vertical tab;

\ ***t*** - horizontal tab.

In the **<*output list*>** a comma identifies the variables whose values are displayed. The ***putchar ()*** function is used to display the symbol on the screen .

Example program illustrating the use of I / O operations:

```
#include <stdio.h>
#include <windows.h>
void main ()
{
char ch = '1';
int k, m; // description of variables
float a; // description of variables
SetConsoleOutputCP (1251);
SetConsoleCP (1251);
putchar (ch); // display the character on the screen
printf ("\ n enter data \ n"); // output data to the screen
scanf ("% i% i% f", & k, & m, & a); // data entry
```

*printf ("\ nk =% d \ tm =% d \ ta =% f \ n", k, m, a); / * display data * /*

*}*

The result of the program:

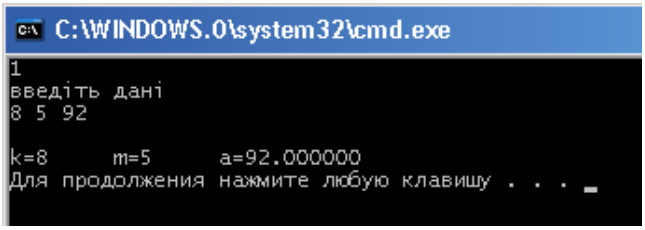


Fig. 2.1 - The result of the program

I / O operations can also be implemented using the ***iostream*** I / O ***library*** (I / O functions using C ++ classes):

- data entry from the keyboard: ***cin*** >> <***variable identifier***> ;
- data output to the screen: ***cout*** << <***expression***>**.**

here <***expression***> **-** variable identifier, string of characters or arithmetic expression.

To include a new string character (equivalent \ ***n*** ) in the stream, ***endl*** is ***used*** .

Example program illustrating the use of streaming I / O operations:

*#include <iostream>*

*#include <windows.h>*

*using namespace std;*

*void main ()*

*{*

*char ch = '1';*

*int k, m; // description of variables*

*float a; // description of variables*

*SetConsoleOutputCP (1251);*

*SetConsoleCP (1251);*

*cout << ch; // display the character on the screen*

*cout << endl << "enter data k, m, a" << endl;*

*cin >> k >> m >> a; // data entry*

*// output data and move to a new line*

*cout << "k =" << k << "\ tm =" << m << "\ ta =" << a*

*<< endl;*

*}*

The result of the program:

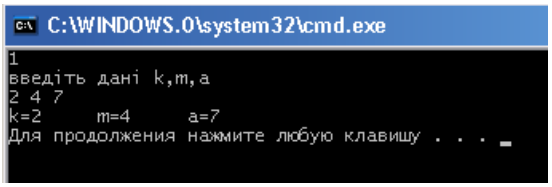


Fig. 2.2 - The result of the program

The previous example in the absence of such a directive will look like (here also instead of a streaming character of a new line **endl** the control character of function **printf** of transition to a new line is used \ **n** ):

*#include <iostream>*

*#include <windows.h>*

*void main ()*

*{*

*char ch = '1';*

*int k, m; // description of variables*

*float a; // description of variables*

*SetConsoleOutputCP (1251);*

*SetConsoleCP (1251);*

*std :: cout << ch; // display the character on the screen*

*std :: cout << "\ nentry data k, m, a \ n";*

*// output data to the screen*

*std :: cin >> k >> m >> a; // data entry*

*std :: cout << "k =" << k << "\ tm =" << m << "\ ta =" << a << "\ n";*

*// output data to the screen*

*}*

The result of the program:

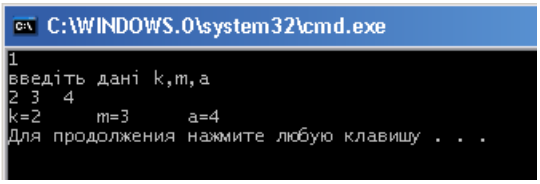


Fig. 2.3 - The result of the program

When creating a program take into account the following basic requirements:

- all used constants, variables, functions and non-standard types must be declared (described) before their first use, and these declarations can be placed anywhere in the program;
- each language operator ends with the symbol **";"**;
- curly braces ( **"{"** and **"}"** ) distinguish the compound operator and everything that is given between such parentheses is syntactically perceived as one operator;
- nested blocks should have an indentation of 3-4 characters, with blocks of one level of nesting should be aligned vertically.

## Tasks

Write programs that input and output variables of all standard types (see Appendix №2). The first program is using the ***#include <stdio.h>*** preprocessor directive, and the second is using the ***#include <iostream>*** directive. Also use format output and stream redirection operations to work with files.

**Note:** if an **exe** file is run for execution, then in order to be able to view the results of the program execution, it is necessary to stop closing the console window with the **system ("pause")** command, adding it at the end of the program.

**Control questions:**

1. What is the structure of the program?

2. List the basic requirements that should be considered when creating programs in C ++.

# PRACTICAL WORK №3
# Topic: Linear programs. Calculation of arithmetic expressions and mathematical functions

**Purpose:** Formation of skills and abilities of programming of arithmetic expressions, calculation of mathematical functions and writing of simple linear programs.

**Theoretical information**

Assignment operator = allows you to replace the value of the operand to the left of the equals sign with the value of the expression calculated to the right of it.

Assignment operator syntax:

***<variable identifier> = <expression>***

It is permissible to record the species:

***<variable identifier1> = <variable identifier2> = <expression>***

The main mathematical operations of C ++ are given in Appendix №4. In complex expressions, the order of operations is determined by parentheses and the priority of operations. You can use several levels of parentheses: the calculation goes from the inner parentheses to the outer ones. The order of operations in an expression is important if there are several operations with different priorities: operations with the same priority are performed before operations with a lower priority, regardless of where they are in the expression.

Operations are performed from left to right in order of priority from highest to lowest:

1. challenge of mathematical functions;
2. **++, -** ;
3. ***, /,%*** ;
4. **+, -** ;
5. assignment operations: **=, + =, - =, * =, / =,% =** .

In the assignment operator, the variables that make up the expression must be of the same type. If they are of different types, it is necessary to convert variables to one type. The conversion is performed so that the variables occupying a smaller amount of RAM were converted to the type of variables occupying a larger volume.

Example:

*int a;*

*float y, b;*

*y = a + b;*

In the assignment operator ***y = a + b*** it is necessary to write:

*y = float (a) + b* .

The ***float (a)*** operation is called a type cast operation: it converts an integer variable to a valid type. When performing a cast operation, information may be lost.

For example, as a result of executing the following program fragment, the variable ***k*** will take the value 4.

*double a = 4.24;*

*int k;*

*k = int (a);*

For the "binding" several expressions used following **,** (comma). Expressions separated by commas are calculated from left to right.

Example:

*a = 4, b = b + a + 5, c = b / 5;*

Mathematical functions are located in the math library , which is connected at the beginning of the program by the preprocessor directive #include <math.h> . It should be remembered that all trigonometric functions in C ++ work with angular values given in radians.

C++ being superset of C, supports large number of useful mathematical functions.

These functions are available in standard C++ and C to support various mathematical calculations. In order to use these functions you need to include header file- **<math.h>** or **<cmath>**.

• **double sin(double)** : This function takes angle (in degree) as an argument and return its sine value that could be verified using sine curve.

• **double cos(double)** : This function takes angle (in degree) as an argument and return its cosine value that could be verified using cosine curve.

• **double tan(double)** : This function takes angle (in degree) as an argument and return its tangent value. This could also be verified using Trigonometry as Tan(x) = Sin(x)/Cos(x).

• **double sqrt(double)** : This function takes number as an argument and return its square root value. Number can not be negative value.

• **int abs(int)** : This function takes integer number as an argument and return its absolute value. It means, the output will always be positive regardless of sign of input.

• **double pow(double, double)** : This function takes one argument as base and other as exponent.

• **double hypot(double, double)** : This function requires two sides of the right angled triangle to give output as its hypotenuse.

• **double floor(double)** : This functions returns the integer value lesser or equal to argument passed in the function.

• **double fabs(double)** : This function returns the absolute value of any number.

• **double acos(double)** : This function returns the arc cosine of argument. The argument to acos() must be in the range -1 to 1 ; otherwise, a domain error occurs.

• **double asin(double)** : This function returns the arc sine of argument. The argument to asin() must be in the range -1 to 1 ; otherwise, a domain error occurs.

• **double atan(double)** : This function returns the arc tangent of arg.

• **double atan2(double, double)** : This function returns the arc tangent of (double a)/(double b).

• **double ceil(double)** : This function returns the smallest integer as double not less than the argument provided.

• **double cosh(double)** : This function returns the hyperbolic cosine of argument provided. The value of argument provided must be in radians.

• **double tanh(double)** : This function returns the hyperbolic tangent of argument provided. The value of argument provided must be in radians.

• **double log(double)** : This function takes a number and returns the natural log of that number.

**Example:**

Write a program to calculate the arithmetic expression and output the result to the screen using streaming I/O.

$$h = \frac{x^{2y} + e^{y-1}}{1 + x|y - tg(z)|} + 10 \cdot \sqrt[3]{x} - \ln(z)$$

```
#include <iostream>
#include <math.h>
#include <windows.h>
using namespace std;
int main()
{
double x,y,z,a,b,c,h;
SetConsoleOutputCP(1251);
SetConsoleCP(1251);
cout << "Enter x:";
cin >> x;
cout << "Enter y:";
cin >> y;
cout << "Enter z:";
cin >> z;
a=pow(x,2*y)+exp(y-1);
b=1+x*fabs(y-tan(z));
c=10*pow(x,1/3.0)-log(z);
h=a/b+c;
```

```
cout << " Calculation result h= "<<h<<endl;
return 0;
}
```

## Tasks

Writing a program, use I /O streaming operations and explanatory text information for easy visual perception of the results.

1) $\dfrac{|x\ln x - 4/7|\cdot\sqrt{x}}{\sqrt[5]{e^{4x-1.1}}}$;

2) $\sqrt[3]{\pi^2 - x^2 + e^{-1}} + \operatorname{tg}\dfrac{x-1}{x} + \dfrac{1}{7}$

3) $\sqrt{e^{2.2x}} - \left|\sin\dfrac{\pi x}{x+2/3}\right| + 1.7$;

4) $\sqrt{e^{2x}\sqrt{x} - \dfrac{x+1/3}{x}}\cdot|\cos 2.5x|$;

5) $\sqrt{\sqrt[5]{x^4} + \sqrt[5]{x^{4\ x}}} + \ln|x - 20.5|$

6) $\dfrac{|7.2-10x|}{\sqrt[3]{\frac{x}{9}+e^{2x}}}\cdot\operatorname{arctg}\dfrac{4\operatorname{tg}2x}{\sqrt{1.1x^3}}$;

7) $\dfrac{\sqrt{\sin^3\frac{x}{2}} + \sqrt[3]{e^{1.3x}+e^{-1.3x}}}{|x-7/9|}$;

8) $\dfrac{\left|\ln x^2\right| + 1/3}{\sqrt{e^{x/\pi}+\sqrt[3]{x}+1.4}}$;

9) $\dfrac{\sqrt[5]{e^{2/3-x}}}{\sqrt{x^2+x^4+\ln|x-3.4|}}$;

10) $\dfrac{\ln(x^2)+\pi}{e^{5/3}} - x\cdot\operatorname{arctg}\dfrac{x}{\sqrt{e}} + 1.4$;

11) $\left(\dfrac{1}{7} + \ln\sqrt{x}\right)\cdot e^{\sqrt{x-2}}$;

12) $\dfrac{x^3}{\sqrt{3}} - e^x\ln\left|1.37^3 + x^3\right| + \dfrac{4}{3}$;

13) $\dfrac{\sqrt{x}\sin\frac{x^2}{2} - 1.3}{\sqrt[5]{x}+e^{3x}+|\cos x|}$;

14) $\dfrac{\ln\sqrt{\pi+|2-x|}}{3-1/x} + \sqrt[3]{x^2}\cdot\sin 1.4x$;

15) $\sqrt{e^{|\sin^3 x|}} + 2\ln 3x - \dfrac{1}{9}$;

16) $\left(\sqrt[3]{\ln^2 x} + \operatorname{tg}\cos\pi x\right)\cdot\left|\ln\dfrac{x}{10.5}\right|$ ;

# PRACTICAL WORK №4

# Topic: If else Statement in C++

**Purpose:** Formation of skills and abilities to develop branched programs for the organization of alternative calculations.

**Theoretical information**

Sometimes we need to execute a block of statements only when a particular condition is met or not met. This is called decision making, as we are executing a certain code after making a decision in the program logic. For decision making in C++, we have four types of control statements (or control structures), which are as follows:

a) if statement

b) nested if statement

c) if-else statement

d) if-else-if statement

If statement consists a condition, followed by statement or a set of statements as shown below:

```
if(condition){
 Statement(s);
}
```

The statements inside if parenthesis (usually referred as if body) gets executed only when the given condition is true. If the condition is false then the statements inside if body are completely ignored.

**Example of if statement**

```
#include <iostream>
using namespace std;
int main(){
  int num=70;
```

```
  if( num < 100 ){
    /* This cout statement will only execute,
     * if the above condition is true
     */
    cout<<"number is less than 100";
  }
  if(num > 100){
    /* This cout statement will only execute,
     * if the above condition is true
     */
    cout<<"number is greater than 100";
  }
  return 0;
}
```

***Output:***

*number is less than 100*

When there is an if statement inside another if statement then it is called the nested if statement.

The structure of nested if looks like this:

```
if(condition_1) {
  Statement1(s);
  if(condition_2) {
    Statement2(s);
  }
}
```

Statement1 would execute if the condition_1 is true. Statement2 would only execute if both the conditions( condition_1 and condition_2) are true.

Sometimes you have a condition and you want to execute a block of code if condition is true and execute another piece of code if the same condition is false. This can be achieved in C++ using if-else statement.

This is how an if-else statement looks:

```
if(condition) {
  Statement(s);
}
else {
  Statement(s);
}
```

The statements inside “if” would execute if the condition is true, and the statements inside “else” would execute if the condition is false.

**Program for calculating a function**

$$y(x)=\begin{cases}-1 \; npu \; x<0\\ 0 \; npu \; x=0\\ +1 \; npu \; x>0\end{cases}$$

```
#include "pch.h"
#include <iostream>
#include <windows.h>
using namespace std;

int main()
{
  int x, y;
  SetConsoleOutputCP(1251);
  SetConsoleCP(1251);
  cout << "Enter x \n";
```

```
    cin >> x;
    cout << x = " << x;
    if (x < 0) y = -1; else if (x == 0) y = 0; else y = 1;
    cout << "\nx = " << x << "\t " << "y = " << y << "\n";
}
```

## Tasks

1. Write a C++ program to find maximum between two numbers.
2. Write a C++ program to find maximum between three numbers.
3. Write a C++ program to check whether a number is negative, positive or zero.
4. Write a C++ program to check whether a number is divisible by 2 or not.
5. Write a C++ program to find maximum between three numbers.
6. Write a C++ program to find maximum between two numbers.
7. Write a C++ program to check whether a number is negative, positive or zero.
8. Write a C++ program to check whether a number is divisible by 3 or not.
9. Write a C++ program to find maximum between two numbers.
10. Write a C++ program to find maximum between two numbers.
11. Write a C++ program to check whether a number is negative, positive or zero.
12. Write a C++ program to check whether a number is divisible by 5 or not.

**Control questions:**

1. What is the conditional operator used for?
2. Name two forms of conditional operator?
3. Name the comparison operations in C ++?
4. What is the keyword "else" used for?

# PRACTICAL WORK №5

# Topic: For loop

**Purpose:** To develop skills in the development of simple programs usig for loop.

**Theoretical information**

The loops are needed to execute a block of code many of the times. In this, basically the statements are executed sequentially which means the first statement in a function is executed first than the second statement and so on. The loops are used to execute a single statement or the group of statements multiple times.

A loop is used for executing a block of statements repeatedly until a particular condition is satisfied. For example, when you are displaying number from 1 to 100 you may want set the value of a variable to 1 and display it 100 times, increasing its value by 1 on each loop iteration.

The **for loop** is used as a repetition control statement that allows you to write a loop that will execute a specific number of times. The for loop is used to execute the block of statements again and again till the condition returns false. In for loop, the statement is executed till the Boolean condition is not true, as the condition is true it will terminate the loop. The for loop consists of three parts first is the initialization, second is the condition and third is increment or decrement statement.

**The for loop syntax is like this:**

**for** (initialization; condition; increment **or** decrement )

{ //statement;

}

First step: In for loop, initialization happens first and only once, which means that the initialization part of for loop only executes once.

Second step: Condition in for loop is evaluated on each loop iteration, if the condition is true then the statements inside for for loop body gets executed. Once the

condition returns false, the statements in for loop does not execute and the control gets transferred to the next statement in the program after for loop.

Third step: After every execution of for loop's body, the increment/decrement part of for loop executes that updates the loop counter.

Fourth step: After third step, the control jumps to second step and condition is re-evaluated.

The steps from second to fourth repeats until the loop condition returns false.

**Example of a Simple For loop in C++**

Here in the loop initialization part I have set the value of variable i to 1, condition is i<=6 and on each loop iteration the value of i increments by 1.

```
#include <iostream>
using namespace std;
int main(){
  for(int i=1; i<=6; i++){
    /* This statement would be executed
     * repeatedly until the condition
     * i<=6 returns false.
     */
    cout<<"Value of variable i is: "<<i<<endl;
  }
  return 0;
}
```

Output:

```
Value of variable i is: 1
Value of variable i is: 2
Value of variable i is: 3
Value of variable i is: 4
Value of variable i is: 5
Value of variable i is: 6
```

## Tasks

1. Write a program in C++ to find the first 10 natural numbers. Go to the editor

Sample output:

The natural numbers are:

1 2 3 4 5 6 7 8 9 10

2. Write a program in C++ to find the sum of first 10 natural numbers. Go to the editor

Sample Output:

Find the first 10 natural numbers:

---

The natural numbers are:

1 2 3 4 5 6 7 8 9 10

The sum of first 10 natural numbers: 55

3. Write a program in C++ to display n terms of natural number and their sum. Go to the editor

Sample Output:

Input a number of terms: 7

The natural numbers upto 7th terms are:

1 2 3 4 5 6 7

The sum of the natural numbers is: 28

4. Write a program in C++ to check whether a number is prime or not. Go to the editor

Sample Output:

Input a number to check prime or not: 13

The entered number is a prime number.

5. Write a program in C++ to find prime number within a range. Go to the editor

Input number for starting range: 1

Input number for ending range: 100

The prime numbers between 1 and 100 are:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

The total number of prime numbers between 1 to 100 is: 25

6. Write a program in C++ to find the factorial of a number. Go to the editor

Sample output:

Input a number to find the factorial: 5

The factorial of the given number is: 120

**Control questions**

1. What is a loop?
2. What operators are called loop operators?
3. Name the operators of  loop C ++.
4. Give the general syntax of the for loop.
5. What variable is called a loop counter?

# PRACTICAL WORK №6

# Topic: WHILE LOOP and DO-WHILE LOOP

**Purpose:** To develop skills in the development of simple loop programs with an unknown number of repetitions for the implementation of basic data processing algorithms.

**Theoretical information**

The while loop is basically most basic loop statement, it repeats the statement till the condition is true. In this, there can be a single statement or block of statements and the condition can be any expression and any non zero value is true. It basically targets the statement as long as the given condition is true.

**Let us have a look at the syntax:**

```
while(condition)
{   //statements
}
```

In while loop, condition is evaluated first and if it returns true then the statements inside while loop execute, this happens repeatedly until the condition returns false. When condition returns false, the control comes out of loop and jumps to the next statement in the program after while loop.

Note: The important point to note when using while loop is that we need to use increment or decrement statement inside while loop so that the loop variable gets changed on each iteration, and at some point condition returns false. This way we can end the execution of while loop otherwise the loop would execute indefinitely.

While Loop example in C++:

```
#include <iostream>
using namespace std;
int main(){
  int i=1;
```

```
  /* The loop would continue to print
   * the value of i until the given condition
   * i<=6 returns false.
   */
  while(i<=6){
    cout<<"Value of variable i is: "<<i<<endl; i++;
  }
}
```

Output:

Value of variable i is: 1

Value of variable i is: 2

Value of variable i is: 3

Value of variable i is: 4

Value of variable i is: 5

Value of variable i is: 6

As you have study about the for loop and while loop in which they test the loop condition at the top of the loop whereas do-while loop check the condition at the bottom of the loop. In do-while loop, the program will execute the body of the loop first and at the end of the loop, the test condition in the while statement is executed; if condition is true then program continues to execute loop body otherwise it will come out of the loop.

**Syntax of do-while loop**

```
do
{
  statement(s);
}while(condition);
```

First, the statements inside loop execute and then the condition gets evaluated, if the condition returns true then the control jumps to the “do” for further repeated execution of it, this happens repeatedly until the condition returns false. Once

condition returns false control jumps to the next statement in the program after do-while.

**Do-while loop example in C++:**

```
include <iostream>
using namespace std;
int main(){
  int num=1;
  do{
    cout<<"Value of num: "<<num<<endl;
    num++;
  }while(num<=6);
  return 0;
}
```

**Output:**

Value of num: 1

Value of num: 2

Value of num: 3

Value of num: 4

Value of num: 5

Value of num: 6

**Continue statement** is used inside loops. Whenever a continue statement is encountered inside a loop, control directly jumps to the beginning of the loop for next iteration, skipping the execution of statements inside loop's body for the current iteration.

**Syntax of continue statement**

```
continue;
```

**The break statement** is used in following two scenarios:

a) Use break statement to come out of the loop instantly. Whenever a break statement is encountered inside a loop, the control directly comes out of loop

terminating it. It is used along with if statement, whenever used inside loop(see the example below) so that it occurs only for a particular condition.

b) It is used in switch case control structure after the case blocks. Generally all cases in switch case are followed by a break statement to avoid the subsequent cases (see the example below) execution. Whenever it is encountered in switch-case block, the control comes out of the switch-case body.

**Syntax of break statement**

*break;*

**The goto statement** is used for transferring the control of a program to a given label. The syntax of goto statement looks like this:

**goto label_name;**

Program structure:

**label1:**

**...**

**...**

**goto label2;**

**...**

**..**

**label2:**

**...**

In a program we have any number of goto and label statements, the goto statement is followed by a label name, whenever goto statement is encountered, the control of the program jumps to the label specified in the goto statement.

Goto statements are almost never used in any development as they are complex and makes your program much less readable and more error prone. In place of goto, you can use continue and break statement.

## Examples

Write a program in C++ to find the sum of digits of a given number.

Sample Output:

Input a number: 1234

The sum of digits of 1234 is: 10

```
#include <iostream>
using namespace std;
int main()
{
   int num1, num2, r, sum;
   cout << "\n\n Find the sum of digits of a given number:\n";
   cout << "-------------------------------------------\n";
   cout << " Input a number: ";
   cin >> num1;
   num2 = num1;
   while (num1 > 0)
   {
      r = num1 % 10;
      num1 = num1 / 10;
      sum = sum + r;
   }
   cout << " The sum of digits of " << num2 << " is: " << sum << endl;
}
```

Sample Output:

```
 Find the sum of digits of a given number:
-------------------------------------------
 Input a number: 1234
 The sum of digits of 1234 is: 10
```

## Tasks

1. Write a program in C++ to find the sum of the series 1 + 1/2^2 + 1/3^3 + ..+ 1/n^n.

Sample Output:

Input the value for nth term: 5

1/1^1 = 1

1/2^2 = 0.25

1/3^3 = 0.037037

1/4^4 = 0.00390625

1/5^5 = 0.00032

The sum of the above series is: 1.29126

2. Write a program in C++ to calculate the sum of the series (1*1) + (2*2) + (3*3) + (4*4) + (5*5) + ... + (n*n).

Sample Output:

Input the value for nth term: 5

1*1 = 1

2*2 = 4

3*3 = 9

4*4 = 16

5*5 = 25

The sum of the above series is: 55

3. Write a program in C++ to calculate the series (1) + (1+2) + (1+2+3) + (1+2+3+4) + ... + (1+2+3+4+...+n).

Sample Output:

Input the value for nth term: 5

1 = 1

1+2 = 3

1+2+3 = 6

1+2+3+4 = 10

1+2+3+4+5 = 15

The sum of the above series is: 35

4. Write a program in C++ to find the sum of series 1 - X^2/2! + X^4/4!-.... upto nth term.

Sample Output:

Input the value of X: 3

Input the value for nth term: 4

term 1 value is: 1

term 2 value is: -4.5

term 3 value is: 3.375

term 4 value is: -1.0125

The sum of the above series is: -1.1375

5. Write a program in C++ to asked user to input positive integers to process count, maximum, minimum, and average or terminate the process with -1.

Sample Output:

Your input is for termination. Here is the result below:

Number of positive integers is: 4

The maximum value is: 9

The minimum value is: 3

The average is 6.00

6. Write a program in C++ to list non-prime numbers from 1 to an upperbound.

Sample Output:

Input the upperlimit: 25

The non-prime numbers are:

4 6 8 9 10 12 14 15 16 18 20 21 22 24 25

Click me to see the sample solution

7. Write a program in C++ to print a square pattern with # character. Sample Output:

Print a pattern like square with # character:

---

Input the number of characters for a side: 4

# # # #

# # # #

# # # #

# # # #

## Control questions

1. Give the general syntax of the while loop.
2. Give the general syntax of the do- while loop.
3. What is the difference between a counter-controlled cycle and a control value-controlled cycle?

# PRACTICAL WORK №7

# Topic: One-dimensional arrays

**Purpose:** to consolidate the skills of working with arrays, mastering the simplest sorting algorithms.

**Theoretical information**

**Array** – a set of variables of the same type. Access to these variables is carried out by the same name. This name is called the name of the array. Arrays are used for grouping related variables between themselves.

An array is a collection of similar elements of same data type at a contiguous memory location. The array is a data structure which stores the fixed size of elements in sequence.

The size and the data type of an array cannot be changed after the declaration. The length of the array is defined when the array is created and the length of the array is fixed it cannot be changed.

In arrays, each value is an element and these elements are accessed through an index.

Arrays can be one-dimentional and multidimentional. In the one-dimensional arrays, only one index is used to access an array element. In multidimensional arrays to access an element of the array are used multiple indexes.

**The general form of the description of one-dimensional array:**

*type array_name[size];*

In the definition above:

type – the type of array items. It is also called a base type. The base type determines the amount of data of each item that is an array. Array type can be both a base type and the aggregate type (e.g., structure).

size – number of items in array;

array_name – directly the array name that provides access to the array elements.

After describing the array, elements value may be zero or undefined.

For example, take an integer array 'n'.

*int n[6];*

n[ ] is used to denote an array named 'n'.

So, n[6] means that 'n' is an array of 6 integers. Here, 6 is the size of the array i.e., there are 6 elements of array 'n'.

Giving array size i.e. 6 is necessary because the compiler needs to allocate space to that many integers. The compiler determines the size of an array by calculating the number of elements of the array.

Here 'int n[6]' will allocate space to 6 integers.

We can also declare an array by another method.

*int n[ ] = { 2,3,15,8,48,13 };*

In this case, we are declaring and assigning values to the array at the same time. Hence, no need to specify the array size because the compiler gets it from

{ 2,3,15,8,48,13 }.

Following is the pictorial view of the array.

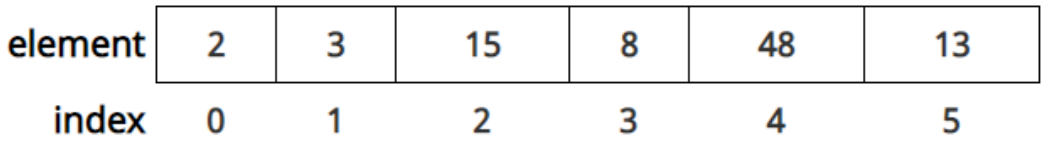


Fig.7.1 One-dimensional array.

0,1,2,3,4 and 5 are the indices. It is like these are the identities of 6 different elements of the array. Index starts from 0. So, the first element has index 0. We access the elements of an array by writing *array_name[index]*.

By writing int n[ ]={ 2,4,8 }; , we are initializing the array.

But when we declare an array like int n[3]; , we need to assign the values to it separately. Because 'int n[3];' will definitely allocate the space of 3 integers in the memory but there are no integers in that.

To assign values to the array, assign a value to each of the element of the array.

*n[0] = 2;*

*n[1] = 4;*

*n[2] = 8;*

It is just like we are declaring some variables and then assigning the values to them.

*int x,y,z;*

*x=2;*

*y=4;*

*z=8;*

Thus, the two ways of initializing an array are:

*int n[ ]={ 2,4,8 };*

and the second method is declaring the array first and then assigning the values to its elements.

*int n[3];*

*n[0] = 2;*

*n[1] = 4;*

*n[2] = 8;*

You can understand this by treating n[0], n[1] and n[2] as different variables you used before.

Just like a variable, an array can be of any other data type also.

*float f[ ]= { 1.1, 1.4, 1.5};*

Here, f is an array of floats.

The array elements are accessed by indexing the array name and it can be done by putting the index of the element within the square brackets after the name of the array.

You can access elements of an array by using indices.

Suppose you declared an array mark as above. The first element is mark[0], second element is mark[1] and so on.

**Few key notes:**

- Arrays have 0 as the first index not 1. In this example, mark[0] is the first element.
- If the size of an array is n, to access the last element, (n-1) index is used. In this example, mark[4] is the last element.

***Example 1.*** First, let's see the example to calculate the average of the marks of 3 students. Here, marks[0], marks[1] and marks[2] represent the marks of the first, second and third student respectively.

```
#include <iostream>
int main(){
  using namespace std;
  int marks[3];
  float average;
  cout << "Enter marks of first student" << endl;
  cin >> marks[0];
  cout << "Enter marks of second student" << endl;
  cin >> marks[1];
  cout << "Enter marks of third student" << endl;
  cin >> marks[2];
  average = ( marks[0] + marks[1] + marks[2] )/ 3.0;
  cout << "Average marks : " << average << endl;
  return 0;
}
```

In the above example, two points should be kept in mind. The average value should be of type 'float' because the average of integers can be float also.

Secondly, while taking out the average, sum of the numbers should be divided by 3.0 and not 3, otherwise, you will get the average value as integer and not float.

***Example 2.*** This program calculates the average if the number of data are from 1 to 100.

```
#include <iostream>
using namespace std;
int main()
{
   int n, i;
   float num[100], sum=0.0, average;
   cout << "Enter the numbers of data: ";
   cin >> n;
   while (n > 100 || n <= 0)
   {
      cout << "Error! number should in range of (1 to 100)." << endl;
      cout << "Enter the number again: ";
      cin >> n;
   }
   for(i = 0; i < n; ++i)
   {
      cout << i + 1 << ". Enter number: ";
      cin >> num[i];
      sum += num[i];
   }
   average = sum / n;
   cout << "Average = " << average;
   return 0;
}
```

Output

Enter the numbers of data: 6

1. Enter number: 45.3

2. Enter number: 67.5

3. Enter number: -45.6

4. Enter number: 20.34

5. Enter number: 33

6. Enter number: 45.6

Average = 27.69

If user enters value of n above 1 or below 100 then, while loop is executed which asks user to enter value of n until it is between 1 and 100.

***Example 3:*** Display Largest Element of an array

```
#include <iostream>
using namespace std;
int main()
{
   int i, n;
   float arr[100];
   cout << "Enter total number of elements(1 to 100): ";
   cin >> n;
   cout << endl;
   // Store number entered by the user
   for(i = 0; i < n; ++i)
   {
     cout << "Enter Number " << i + 1 << " : ";
     cin >> arr[i];
   }
   // Loop to store largest number to arr[0]
   for(i = 1;i < n; ++i)
   {
     // Change < to > if you want to find the smallest element
     if(arr[0] < arr[i])
        arr[0] = arr[i];
   }
```

```
    cout << "Largest element = " << arr[0];
    return 0;
}
```

Output

```
Enter total number of elements: 8
Enter Number 1: 23.4
Enter Number 2: -34.5
Enter Number 3: 50
Enter Number 4: 33.5
Enter Number 5: 55.5
Enter Number 6: 43.7
Enter Number 7: 5.7
Enter Number 8: -66.5
Largest element = 55.5
```

This program takes n number of elements from user and stores it in array arr[].

To find the largest element, the first two elements of array are checked and largest of these two element is placed in arr[0].

Then, the first and third elements are checked and largest of these two element is placed in arr[0].

This process continues until and first and last elements are checked.

After this process, the largest element of an array will be in arr[0] position.

**Bubble Sort**

In the bubble sort, as elements are sorted they gradually "bubble" (or rise) to their proper location in the array, like bubbles rising in a glass of soda. The bubble sort repeatedly compares adjacent elements of an array. The first and second elements are compared and swapped if out of order. Then the second and third elements are compared and swapped if out of order. This sorting process continues until the last two elements of the array are compared and swapped if out of order.

When this first pass through the array is complete, the bubble sort returns to elements one and two and starts the process all over again. So, when does it stop? The bubble sort knows that it is finished when it examines the entire array and no "swaps" are needed (thus the list is in proper order). The bubble sort keeps track of occurring swaps by the use of a flag.

The table below follows an array of numbers before, during, and after a bubble sort for descending order. A "pass" is defined as one full trip through the array comparing and if necessary, swapping, adjacent elements. Several passes have to be made through the array before it is finally sorted.

The bubble sort is an easy algorithm to program, but it is slower than many other sorts. With a bubble sort, it is always necessary to make one final "pass" through the array to check to see that no swaps are made to ensure that the process is finished. In actuality, the process is finished before this last pass is made.

```
#include<iostream>
 using namespace std;
 int main()
{
  int a[50],n,i,j,temp;
  cout<<"Enter the size of array: ";
  cin>>n;
  cout<<"Enter the array elements: ";
        for(i=0;i<n;++i)
        cin>>a[i];
  for(i=1;i<n;++i)
  {
        for(j=0;j<(n-i);++j)
              if(a[j]>a[j+1])          // ascending order simply changes to <
              {
                    temp=a[j];  // swap elements
                    a[j]=a[j+1];
```

```
                a[j+1]=temp;
            }
        }
        cout<<"Array after bubble sort:";
        for(i=0;i<n;++i)
            cout<<" "<<a[i];
        return 0;
    }
```

## Tasks

1(6).Write a C++ program to find the two repeating elements in a given array of integers.

2(7). Write a C++ program to find the number of pairs of integers in a given array of integers whose sum is equal to a specified number.

3(8). Write a C++ program to find the largest element of a given array of integers.

4(9). Write a C++ program to sort 10 integer values (reading from keyboard) in ascending and descending order.

5(10). Write a C++ program to delete an element from an array at specified position.

### Control questions

1. What is an array?
2. Provide the syntax for the one-dimensional array declaration.
3. What is an array element index?
4. How can I access array elements?
5. Give the syntax of the reference to a specific element of the array.
6. How can I set the initial values of the elements of the array?

# PRACTICAL WORK № 8

# Topic: Development of programs using two-dimensional arrays

**Purpose:** to master the technique of using two-dimensional arrays in C++ programming, to learn to build a mathematical model of the problem.

**Theoretical information**

Multidimensional array

The multi-dimensional array is that array in which data is arranged in the form of array of arrays. The multi-dimensional array can have as many dimensions as it required.

So, two dimensional and three dimensional arrays are commonly used.

Multidimensional arrays are also known as array of arrays.

**2-dimensional arrays** also exist and are generally known as matrix. These consist of rows and columns.

Before going into its application, let's first see how to declare and initialize a 2 D array.

Similar to one-dimensional array, we define 2-dimensional array as below.

*int a[2][4];*

Here, a is a 2-D array of type int which consists of 2 rows and 4 columns.

Same as in one-dimensional array, we can assign values to a 2-dimensional array in 2 ways as well.

In the first method, just assign a value to the elements of the array. If no value is assigned to any element, then its value is assigned zero by default.

Suppose we declared a 2-dimensional array a[2][2]. Then to assign it values, we need to assign a value to its elements.

*int a[2][2];*

*a[0][0]=1;*

*a[0][1]=2;*

*a[1][0]=3;*

*a[1][1]=4;*

The second way is to declare and assign values at the same time as we did in one-dimensional array.

*int a[2][3] = { 1, 2, 3, 4, 5, 6 };*

Here, value of a[0][0] is 1, a[0][1] is 2, a[0][2] is 3, a[1][0] is 4, a[1][1] is 5 and a[1][2] is 6.

We can also write the above code as:

```
int a[2][3] = {
   {1, 2, 3},
   {4, 5, 6 }
};
```

While assigning values to an array at the time of declaration, there is no need to give dimensions in one-dimensional array, but in 2 D array, we need to give at least the second dimension.

Let's consider different cases of initializing an array.

```
int a[2][2] = { 1, 2, 3, 4 }; /* valid */
int a[ ][2] = { 1, 2, 3, 4 }; /* valid */
int a[2][ ] = { 1, 2, 3, 4 }; /* invalid */
int a[ ][ ] = { 1, 2, 3, 4 }; /* invalid */
```

*int arr[2][3];*

This array has total 2*3 = 6 elements.

Accessing array elements:

*arr[0][0] – first element*

*arr[0][1] – second element*

*arr[0][2] – third element*

*arr[1][0] – fourth element*

*arr[1][1] – fifth element*

*arr[1][2] – sixth element*

**Example: Two dimensional array in C++**

```
#include <iostream>
using namespace std;
int main(){
  int arr[2][3] = {{11, 22, 33}, {44, 55, 66}};
  for(int i=0; i<2;i++){
    for(int j=0; j<3; j++){
      cout<<"arr["<<i<<"]["<<j<<"]: "<<arr[i][j]<<endl;
    }
  }
  return 0;
}
```

**Output:**

```
arr[0][0]: 11
arr[0][1]: 22
arr[0][2]: 33
arr[1][0]: 44
arr[1][1]: 55
arr[1][2]: 66
```

**A Simple C++ program to add two Matrices:**

Here we are asking user to input number of rows and columns of matrices and then we ask user to enter the elements of both the matrices, we are storing the input into a multidimensional array for each matrix and after that we are adding corresponding elements of both the matrices and displaying them on screen.

```
#include<iostream>
using namespace std;
int main()
{
```

```
  int row, col, m1[10][10], m2[10][10], sum[10][10];
  cout<<"Enter the number of rows(should be >1 and <10): ";
  cin>>row;
  cout<<"Enter the number of column(should be >1 and <10): ";
  cin>>col;
  cout << "Enter the elements of first 1st matrix: ";
  for (int i = 0;i<row;i++ ) {
    for (int j = 0;j < col;j++ ) {
      cin>>m1[i][j];
    }
  }
cout << "Enter the elements of first 1st matrix: ";
  for (int i = 0;i<row;i++ ) {
    for (int j = 0;j<col;j++ ) {
      cin>>m2[i][j];
    }
  }
  cout<<"Output: ";
  for (int i = 0;i<row;i++ ) {
     for (int j = 0;j<col;j++ ) {
       sum[i][j]=m1[i][j]+m2[i][j];
       cout<<sum[i][j]<<" ";
     }
  }
  return 0;
}
```

Output:

Enter the number of rows(should be >1 and <10): 2

Enter the number of column(should be >1 and <10): 3

Enter the elements of  first 1st matrix: 1 2 3 4 5 6

Enter the elements of  first 1st matrix: 5 5 5 5 5 5

Output: 6 7 8 9 10 11

**Tasks**

1. Write a C++ program to subtract two matrices.
2. Write a C++ program to perform Scalar matrix multiplication.
3. Write a C++ program to multiply two matrices.
4. Write a C++ program to check whether two matrices are equal or not.
5. Write a C++ program to find sum of main diagonal elements of a matrix.
6. Write a C++ program to find sum of minor diagonal elements of a matrix.
7. Write a C++ program to find sum of each row and column of a matrix.
8. Write a C++ program to interchange diagonals of a matrix.
9. Write a C++ program to find transpose of a matrix.
10. Write a C++ program to check Symmetric matrix.

## Control questions

1. Give the syntax for referring to a specific element of a two-dimensional array.
2. How can you  set the initial values of the elements of the array?
3. How do you  initialize a 2D array using an initialization list?
4. How to initialize a two-dimensional array using the for loop?
5. What values initialize array elements if the initial values are less than array elements?
6. What are the initial values of the elements of a two-dimensional array in the following case of initialization: array [2] [3] = {1, 2, 3, 4, 5, 6};
7. What are the initial values of the elements of a two-dimensional array in the following case of initialization: array [2] [3] = {{1, 2}, {4}};
8. What determines the amount of memory allocated for a multidimensional array?

# PRACTICAL WORK №9

## Topic: Functions

**Purpose:** development of software for the implementation of algorithms based on functions.

**Theoretical information**

Functions are used for the structured programming which follows top-down approach. Function is a collection of statements or set of statements that perform the task together.

In C++, every program consists of a single function that is a main( ) function. The functions take the inputs and perform the task on those inputs and provide output.

It provides standard to a program and while creating a program we use functions which makes program easier to understand, edit and check the errors, etc.

We have two types of function in C++:

1) Built-in functions
2) User-defined functions

Built-in functions are also known as library functions. We need not to declare and define these functions as they are already written in the C++ libraries such as iostream, cmath etc. We can directly call them when we need.

User defined functions are those functions which are defined by the user. These functions can be created by the user. It allows performing the additional functions besides the in-built functions. Some of the examples of user defied functions are: add( ), multiply( ), divide( ) etc.

When a program begins running, the system calls the main() function, that is, the system starts executing codes from main() function.

When control of the program reaches to function_name() inside main(), it moves to void function_name() and all codes inside void function_name() is executed.

Then, control of the program moves back to the main function where the code after the call to the function_name() is executed as shown in figure above.

**Example: User Defined Function**

C++ program to add two integers. Make a function add() to add integers and display sum in main() function.

```
#include <iostream>
using namespace std;
// Function prototype (declaration)
int add(int, int);
int main()
{
   int num1, num2, sum;
   cout<<"Enters two numbers to add: ";
   cin >> num1 >> num2;
   // Function call
   sum = add(num1, num2);
   cout << "Sum = " << sum;
   return 0;
}
// Function definition
int add(int a, int b)
{
   int add;
   add = a + b;
   // Return statement
   return add;
}
```

Output

Enters two integers: 8

-4

Sum = 4

SOME OF THE ADVANTAGES OF USING FUNCTIONS IN C++ PROGRAM ARE:

- It makes the program clear and easy to understand.
- Single functions can be tested easily for errors.
- It saves time from typing the same functions again and again.
- It helps to modify the program easily without changing the structure of a program.

If a user-defined function is defined after main() function, compiler will show error. It is because compiler is unaware of user-defined function, types of argument passed to function and return type.

In C++, function prototype is a declaration of function without its body to give compiler information about user-defined function. **Function prototype** in the above example is:

*int add(int, int);*

You can see that, there is no body of function in prototype. Also, there are only return type of arguments but no arguments. You can also declare function prototype as below but it's not necessary to write arguments.

*int add(int a, int b);*

Note: It is not necessary to define prototype if user-defined function exists before main() function.

**Function Call**

- To execute the codes of function body, the user-defined function needs to be invoked(called).
- In the above program, add(num1,num2); inside main() function calls the user-defined function.
- The function returns an integer which is stored in variable add.

The general form of a C++ **function definition** is as follows −

*return_type function_name( parameter list ) {*

*body of the function*

```
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

Return Type − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body − The function body contains a collection of statements that define what the function does.

The function itself is referred as function definition. Function definition in the above program is:

```
// Function definition
int add(int a,int b)
{
   int add;
   add = a + b;
   return add;
}
```

When the function is called, control is transferred to the first statement of the function body.

Then, other statements in function body are executed sequentially.

When all codes inside function definition is executed, control of program moves to the calling program.

A function can return a single value to the calling program using return statement.

In the above program, the value of add is returned from user-defined function to the calling program using statement below:

*return add;*

If a function doesn't return anything, then its return type is written as void as in the example below.

```
#include <iostream>
using namespace std;
void display( int n ) /* function */
{
        cout << "Number is " << n << endl;
}
int main(){
  int n;
  cout << "Enter number" << endl;
  cin >> n;
  display(n); /* calling the function display*/
  return 0;
}
```

Output

Enter a number

5

Number is 5

'void' means that function will not return anything.

The default arguments are used when you provide no arguments or only few arguments while calling a function.

The default arguments are used during compilation of program.

For example, lets say you have a user-defined function sum declared like this: int sum(int a=10, int b=20), now while calling this function you do not provide any arguments, simply called sum(); then in this case the result would be 30, compiler

used the default values 10 and 20 declared in function signature. If you pass only one argument like this: sum(80) then the result would be 100, using the passed argument 80 as first value and 20 taken from the default argument.

As you have seen in the above example that I have assigned the default values for only two arguments b and c during function declaration. It is up to you to assign default values to all arguments or only selected arguments but remember the following rule while assigning default values to only some of the arguments:

If you assign default value to an argument, the subsequent arguments must have default values assigned to them, else you will get compilation error.

Yes, we can call a function inside another function. We have already done this. We were calling our functions inside the main function. Now look at an example in which there are two user defined functions. And we will call one inside another.

```
#include <iostream>
using namespace std;
int div_2(int a){
  if(a%2==0){
  return 1;
  }
  else{
    return 0;
  }
}
void div_6(int b){
  if( div_2(b)==1 && b%3 == 0 ){
    cout << "Yes, the number is divisible by 6." << endl;
  }
  else{
    cout << "No, the number is not divisible by 6." << endl;
  }
}
```

```
int main(){
  div_6(12);
  div_6(25);
  return 0;
}
```

Output

Yes, the number is divisible by 6.

No, the number is not divisible by 6.

A number is divisible by 6, if it is divisible by both 2 and 3. We have a function div_2 which will return 1 if the given number is divisible by 2. Another function that we have defined is div_6 which calls div_2 inside itself.

if( div_2(b)==1 && b%3 == 0 ) - In our case, b and thus a are 12 in the first case and 25 is the next case. So, if div_2 returns 1, the number is divisible by 2. And if b%3==0 is true, 'b' is divisible by 3. So if the number is divisible by both 2 and 3, then it is divisible by 6 also.

The process in which a function calls itself is known as **recursion** and the corresponding function is called the recursive function. The popular example to understand the recursion is factorial function.

Factorial function: f(n) = n*f(n-1), base condition: if n<=1 then f(n) = 1.

**C++ recursion example: Factorial**

```
#include <iostream>
using namespace std;
//Factorial function
int f(int n){
   /* This is called the base condition, it is
    * very important to specify the base condition
    * in recursion, otherwise your program will throw
    * stack overflow error.
    */
```

```
  if (n <= 1)
      return 1;
  else
     return n*f(n-1);
}
int main(){
  int num;
  cout<<"Enter a number: ";
  cin>>num;
  cout<<"Factorial of entered number: "<<f(num);
  return 0;
}
```

Output:

Enter a number: 5

Factorial of entered number: 120

**Base condition**

In the above program, you can see that I have provided a base condition in the recursive function.

The condition is: if (n <= 1)  return 1;

The purpose of recursion is to divide the problem into smaller problems till the base condition is reached.

For example in the above factorial program I am solving the factorial function f(n) by calling a smaller factorial function f(n-1), this happens repeatedly until the n value reaches base condition(f(1)=1).

**Direct recursion:** When function calls itself, it is called direct recursion, the example we have seen above is a direct recursion example.

**Indirect recursion:** When function calls another function and that function calls the calling function, then this is called indirect recursion. For example: function A calls function B and Function B calls function A.

## Tasks

1. Write a program to find cube of any number using function.
2. Write a program to find diameter, circumference and area of circle using functions.
3. Write a program to find maximum and minimum between two numbers using functions.
4. Write a program to check whether a number is even or odd using functions.
5. Write a program to check whether a number is prime using functions.
6. Write a C++ program to find power of any number using recursion.
7. Write a C++ program to print all natural numbers between 1 to n using recursion.
8. Write a C++ program to print all even or odd numbers in given range using recursion.
9. Write a C++ program to find sum of all natural numbers between 1 to n using recursion.
10. Write a C++ program to find sum of all even or odd numbers in given range using recursion.

## Control questions

1. What is a function?
2. What is the reason for building programs based on functions?
3. How to declare functions?
4. How to define functions?
5. What is called a body function?
6. What is a list of function parameters?
7. What parameters are called formal?
8. What is a function prototype?
9. What is the type of return value?
10. How to return the value of a function?
11. How do I declare a function that does not return a value?

# PRACTICAL WORK №10

# Тема: Strings

**Topic:** mastering the techniques of working with symbols and strings, the use of library functions for processing strings.

**Theoretical information**

We know that string is a collection of characters. Let's again have a look at string and learn more about it.

**There are two different types of strings in C++.**

- C-style string
- std::string (part of the standard library).

In C programming, the collection of characters is stored in the form of arrays, this is also supported in C++ programming. Hence it's called C-strings.

C-strings are arrays of type char terminated with null character, that is, \0 (ASCII value of null character is 0).

How to define a C-string?

*char str[] = "C++";*

In the above code, str is a string and it holds 4 characters.

Although, "C++" has 3 character, the null character \0 is added to the end of the string automatically.

Alternative ways of defining a string

*char str[4] = "C++";*

*char str[] = {'C','+','+','\0'};*

*char str[4] = {'C','+','+','\0'};*

Like arrays, it is not necessary to use all the space allocated for the string. For example:

```
char str[100] = "C++";
```

Let's see two examples to print a string, one without and the other with for loop.

```
#include <iostream>
using namespace std;
int main(){
  char str[ ] = "Hello";
  cout << str << endl;
  return 0;
}
```

Output

Hello

In the above example, we printed the whole string at once. Now, let's see the same example of printing individual characters of the string using for loop.

```
#include <iostream>
using namespace std;
int main(){
  char str[ ] = "Hello";
  int i;
  for( i=0; i<6; i++)
  {
        cout << str[i];
  }
  return 0;
}
```

Output

Hello

In the first example, we printed the whole string at once. Whereas in the second example, we printed a character each time.

Now let's see how to input string from the user with an example.

```
#include <iostream>

int main()

{

  using namespace std;

  char name[20];       //declaring string 'name'

  cin >> name; //taking string input

  cout << name << endl; //printing string

  return 0;

}
```

Output

```
Peter
Peter
```

char name[20]; - By writing this statement, we declared an array of characters named 'name' and gave it an array size of 20 because we don't know the exact length of the string. Thus, it occupied a space in the memory having a size that is required by 20 characters. So, our array 'name' cannot store more than 20 characters. cin >> name; - This is used to simply input a string from the user as we do for other datatypes and there is nothing new in this.

The above code takes only one word from user to a string variable. It terminates with any white space. Try to write something after space and it will take only the first word.

For example, if in the above example, we input Sam Brad as the name, then the output will only be Sam because the code considers only one word and terminates after the first word (after a whitespace).

We can take input of a string that consists of more than one word by using cin.getline. Let's see an example:

```
#include <iostream>
int main()
{
```

```
 using namespace std;
 char name[20];      //declaring string 'name'
 cin.getline(name, sizeof(name)); //taking string input
 cout << name << endl; //printing string
 return 0;
}
```

Output

Sam Bard

Sam Bard

cin.getline(name, sizeof(name)); - cin.getline takes two arguments, the string variable and the size of that variable. We have used sizeof operator to get the size of string variable 'name'.

C++ program to read and display an entire line entered by user.

```
#include <iostream>
using namespace std;
int main()
{
   char str[100];
   cout << "Enter a string: ";
   cin.get(str, 100);
   cout << "You entered: " << str << endl;
   return 0;
}
```

Output

Enter a string: Programming is fun.

You entered: Programming is fun.

To read the text containing blank space, cin.get function can be used. This function takes two arguments.

First argument is the name of the string (address of first element of string) and second argument is the maximum size of the array.

In the above program, str is the name of the string and 100 is the maximum size of the array.

C++ program to read and display an entire line entered by user.

```
#include <iostream>
using namespace std;
int main()
{
   char str[100];
   cout << "Enter a string: ";
```

```
    cin.get(str, 100);
    cout << "You entered: " << str << endl;
    return 0;
}
```

Output

Enter a string: Programming is fun.

You entered: Programming is fun.

To read the text containing blank space, cin.get function can be used. This function takes two arguments.

First argument is the name of the string (address of first element of string) and second argument is the maximum size of the array.

In the above program, str is the name of the string and 100 is the maximum size of the array.

**Predefined string functions**

We can perform different kinds of string functions like joining of 2 strings, comparing one string with another or finding the length of the string. Let's have a look at the list of such functions.

| Function | Use |
|---|---|
| strlen | calculates the length of string |
| strcat | Appends one string at the end of another |
| strncat | Appends first n characters of a string at the end of another |
| strcpy | Copies a string into another |
| strncpy | Copies first n characters of one string into another |
| strcmp | Compares two strings |
| strncmp | Compares first n characters of two strings |
| strchr | Finds first occurrence of a given character in a string |
| strrchr | Finds last occurrence of a given character in a string |
| strstr | Finds first occurrence of a given string in another string |

Fig.10.1 Predefined string functions.

These predefined functions are part of the cstring library. Therefore, we need to include this library in our code by writing

```
#include <cstring>
```

We will see some examples of strlen, strcpy, strcat and strcmp as these are the most commonly used.

**strlen(s1)** calculates the length of string s1.

Whitespace is also calculated in the length of the string.

```
#include <iostream>
#include <cstring>
using namespace std;
int main(){
  char name[ ]= "Hello";
  int len1, len2;
  len1 = strlen(name);
  len2 = strlen("Hello World");
  cout << "Length of " << name << " = " << len1 << endl;
  cout << "Length of " << "Hello World" << " = " << len2 << endl;
  return 0;
}
```

Output

Length of Hello = 5

Length of Hello World = 11

strlen doesn't count '\0' while calculating length of string.

**2 D Array of Characters**

Same as 2 D array of integers and other data types, we have 2 D array of characters also.

For example, we can write

```
char names[4][10] = {
                    "Andrew",
                    "Kary",
```

```
    "Brown",
    "Lawren"
};
```

**std::string in C++**

Since string is used extensively, C++ provides a built-in string data type. Just like int, float or other data types, we can use string data type also. It simply makes using strings easier.

Same as cin and cout, string is also defined in the std namespace. To use strings in this way, we need to include the header since it is declared in the header. We include it by writing

*#include <string>*

We declare variables of type std::string as follows.

*std::string name;*

Here name is a string variable just like we have int variables, float variables or variables of other data types.

We assign value to a string variable just as we assign value to a variable of any other data type as follows.

*name = "Hall";*

We can also assign value to a string variable at the time of declaring it.

*std::string name("Hall");*

A string variable is just like any other variable.

**Example 1: From a C-style string**

This program takes a C-style string from the user and calculates the number of vowels, consonants, digits and white-spaces.

```
#include <iostream>

using namespace std;

int main()

{
```

```
char line[150];
int vowels, consonants, digits, spaces;
vowels =  consonants = digits = spaces = 0;
cout << "Enter a line of string: ";
cin.getline(line, 150);
for(int i = 0; line[i]!='\0'; ++i)
{
   if(line[i]=='a' || line[i]=='e' || line[i]=='i' ||
      line[i]=='o' || line[i]=='u' || line[i]=='A' ||
      line[i]=='E' || line[i]=='I' || line[i]=='O' ||
      line[i]=='U')
   {
      ++vowels;
   }
   else if((line[i]>='a'&& line[i]<='z') || (line[i]>='A'&& line[i]<='Z'))
   {
      ++consonants;
   }
   else if(line[i]>='0' && line[i]<='9')
   {
      ++digits;
   }
   else if (line[i]==' ')
   {
      ++spaces;
```

```
        }
    }
    cout << "Vowels: " << vowels << endl;
    cout << "Consonants: " << consonants << endl;
    cout << "Digits: " << digits << endl;
    cout << "White spaces: " << spaces << endl;
    return 0;
}
```

Output

Enter a line of string: This is 1 hell of a book.

Vowels: 7

Consonants: 10

Digits: 1

White spaces: 6

**Example 2: From a String Object**

This program takes a string object from the user and calculates the number of vowels, consonants, digits and white-spaces.

```
#include <iostream>
using namespace std;
int main()
{
    string line;
    int vowels, consonants, digits, spaces;
    vowels =  consonants = digits = spaces = 0;

    cout << "Enter a line of string: ";
```

```
getline(cin, line);
for(int i = 0; i < line.length(); ++i)
{
   if(line[i]=='a' || line[i]=='e' || line[i]=='i' ||
      line[i]=='o' || line[i]=='u' || line[i]=='A' ||
      line[i]=='E' || line[i]=='I' || line[i]=='O' ||
      line[i]=='U')
   {
      ++vowels;
   }
   else if((line[i]>='a'&& line[i]<='z') || (line[i]>='A'&& line[i]<='Z'))
   {
      ++consonants;
   }
   else if(line[i]>='0' && line[i]<='9')
   {
      ++digits;
   }
   else if (line[i]==' ')
   {
      ++spaces;
   }
}
cout << "Vowels: " << vowels << endl;
cout << "Consonants: " << consonants << endl;
```

```
    cout << "Digits: " << digits << endl;
    cout << "White spaces: " << spaces << endl;
    return 0;
}
```

Output

```
Enter a line of string: I have 2 C++ programming books.
Vowels: 8
Consonants: 14
Digits: 1
White spaces: 5
```

**Tasks**

1. Write a C++ program to search all occurrences of a character in given string.
2. Write a C++ program to convert lowercase string to uppercase.
3. Write a C++ program to toggle case of each character of a string.
4. Write a C++ program to count total number of vowels and consonants in a string.
5. Write a C++ program to count total number of words in a string.
6. Write a C++ program to find reverse of a string.
7. Write a C++ program to check whether a string is palindrome or not.
8. Write a C++ program to remove all repeated characters from a given string.
9. Write a C++ program to replace all occurrences of a character with another in a string.
10. Write a C++ program to remove all occurrence of a word in given string.

**Control questions**

1. What is the specificity of character arrays?
2. What is a null-terminated string?

# PRACTICAL WORK № 11

## Topic: Structures. Arrays of structures

**Purpose:** to master practical skills in creating and using structures, transferring structures to functions as parameters; learn to use arrays of structures and perform operations with fields of structures; learn to compose programs using structures.

### Theoretical information

Structure is a compound data type that contains different variables of different types.

Structure is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collection of data of different data types.

For example: You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables name, citNo, salary to store these information separately.

However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2, salary2

You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.

A better approach will be to have a collection of all related information under a single name Person, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

This collection of all related information under a single name Person is a structure.

The struct keyword defines a structure type followed by an identifier (name of the structure).

Then inside the curly braces, you can declare one or more members (declare variables inside curly braces) of that structure. For example:

```
struct Person
{
   char name[50];
   int age;
   float salary;
};
```

Here a structure person is defined which has three members: name, age and salary.

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member, for your program. The format of the struct statement is this −

```
struct [structure tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
```

*} book;*

Once you declare a structure person as above. You can define a structure variable as:

*Person bill;*

Here, a structure variable bill is defined which is of type structure Person.

When structure variable is defined, only then the required memory is allocated by the compiler.

Considering you have either 32-bit or 64-bit system, the memory of float is 4 bytes, memory of int is 4 bytes and memory of char is 1 byte.

Hence, 58 bytes of memory is allocated for structure variable bill.

**The members of structure variable is accessed using a dot (.) operator.**

Suppose, you want to access age of structure variable bill and assign it 50 to it. You can perform this task by using following code below:

bill.age = 50;

**Example: C++ Structure**

C++ Program to assign data to members of a structure variable and display it.

```
#include <iostream>
using namespace std;
struct Person
{
   char name[50];
   int age;
   float salary;
};
int main()
{
   Person p1;
      cout << "Enter Full name: ";
   cin.get(p1.name, 50);
   cout << "Enter age: ";
```

```
    cin >> p1.age;
    cout << "Enter salary: ";
    cin >> p1.salary;
    cout << "\nDisplaying Information." << endl;
    cout << "Name: " << p1.name << endl;
    cout <<"Age: " << p1.age << endl;
    cout << "Salary: " << p1.salary;
    return 0;
}
```

Output

```
Enter Full name: Magdalena Dankova
Enter age: 27
Enter salary: 1024.4
Displaying Information.
Name: Magdalena Dankova
Age: 27
Salary: 1024.4
```

Here a structure Person is declared which has three members name, age and salary.

Inside main() function, a structure variable p1 is defined. Then, the user is asked to enter information and data entered by user is displayed.

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use struct keyword to define variables of structure type. Following is the example to explain usage of structure.

We can also make an array of structures. In the first example in structures, we stored the data of 3 students. Now suppose we need to store the data of 100 such children. Declaring 100 separate variables of the structure is definitely not a good option. For that, we need to create an array of structures.

Let's see an example for 5 students.

```
#include <iostream>
#include <cstring>
using namespace std;
struct student
{
  int roll_no;
  string name;
  int phone_number;
};
int main(){
  struct student stud[5];
  int i;
  for(i=0; i<5; i++){                          //taking values from user
          cout << "Student " << i + 1 << endl;
          cout << "Enter roll no" << endl;
          cin >> stud[i].roll_no;
          cout << "Enter name" << endl;
          cin >> stud[i].name;
          cout << "Enter phone number" << endl;
          cin >> stud[i].phone_number;
  }
  for(i=0; i<5; i++){                                    //printing values
          cout << "Student " << i + 1 << endl;
          cout << "Roll no : " << stud[i].roll_no << endl;
          cout << "Name : " << stud[i].name << endl;
          cout << "Phone no : " << stud[i].phone_number << endl;
  }
  return 0;
}
```

Here we created an array named stud having 5 elements of structure student. Each of the element stores the information of a student. For example, stud[0] stores the information of the first student, stud[1] for the second and so on.

Structure variables can be passed to a function and returned in a similar way as normal arguments.

**Passing structure to function in C++**

A structure variable can be passed to a function in similar way as normal argument. Consider this example:

**Example: C++ Structure and Function**

```
#include <iostream>
using namespace std;
struct Person
{
   char name[50];
   int age;
   float salary;
};
void displayData(Person);   // Function declaration
int main()
{
   Person p;
   cout << "Enter Full name: ";
   cin.get(p.name, 50);
   cout << "Enter age: ";
   cin >> p.age;
   cout << "Enter salary: ";
   cin >> p.salary;
   // Function call with structure variable as an argument
   displayData(p);
   return 0;
```

```
}
void displayData(Person p)
{
   cout << "\nDisplaying Information." << endl;
   cout << "Name: " << p.name << endl;
   cout <<"Age: " << p.age << endl;
   cout << "Salary: " << p.salary;
}
```

Output

```
Enter Full name: Bill Jobs
Enter age: 55
Enter salary: 34233.4
Displaying Information.
Name: Bill Jobs
Age: 55
Salary: 34233.4
```

In this program, user is asked to enter the name, age and salary of a Person inside main() function.

Then, the structure variable p is to passed to a function using.

displayData(p);

The return type of displayData() is void and a single argument of type structure Person is passed.

Then the members of structure p is displayed from this function.

**C++ Program to Store Information of a Student in a Structure**

This program stores the information (name, roll and marks entered by the user) of a student in a structure and displays it on the screen.

To understand this example, you should have the knowledge of following C++ programming topics:

- C++ Structures
- C++ Strings

In this program, a structure(student) is created which contains name, roll and marks as its data member. Then, a structure variable(s) is created. Then, data (name, roll and marks) is taken from user and stored in data members of structure variable s. Finally, the data entered by user is displayed.

```
#include <iostream>
using namespace std;
struct student
{
   char name[50];
   int roll;
   float marks;
};
int main()
{
   student s;
   cout << "Enter information," << endl;
   cout << "Enter name: ";
   cin >> s.name;
   cout << "Enter roll number: ";
   cin >> s.roll;
   cout << "Enter marks: ";
   cin >> s.marks;
   cout << "\nDisplaying Information," << endl;
   cout << "Name: " << s.name << endl;
   cout << "Roll: " << s.roll << endl;
   cout << "Marks: " << s.marks << endl;
   return 0;
}
```

Output

Enter information,

Enter name: Bill

Enter roll number: 4

Enter marks: 55.6

Displaying Information,

Name: Bill

Roll: 4

Marks: 55.6

In this program, student (structure) is created.

This structure has three members: name (string), roll (integer) and marks (float).

Then, a structure variable s is created to store information and display it on the screen.

**Tasks**

1. Create an array of structures about students in a group. Write about each student: name, surname, year of birth, marks on five exams. Determine the average score and sort the list by total points.
2. Enter an array of structures. Sort the array in alphabetical order of the surnames included in the structure, moving the structures themselves.
3. Structure "Patient": last name, first name; home address; medical card number. Display patient information with a specific last name.
4. Create an array of structures about students in a group. Write about each student: name, surname, year of birth, grades on three exams. Display information about honors (in all subjects grade 5).
5. The program sets the month and year of two dates. The user enters another date (only month and year). Determine whether the third date belongs to the range from the first date to the second inclusive. The problem is solved using a data structure.
6. Using the data structure, write the program of addition of two complex numbers.
7. Program to Add Two Distances Using Structures

8. Structure "Patient": last name, first name; home address; medical card number. Display patient information with a specific medical card number.
9. Create an array of structures about students in a group. Write about each student: name, surname, marks on two exams. Determine the average score and sort the list by total points.
10. Using the data structure, write the program of multiplication of two complex numbers.

**Control questions**

1. What is a structure? What is the syntax for describing structures?
2. How can a static and dynamic array of structures be declared?
3. Give an example of declaring an array of variables of type structure.
4. How can I access the structure fields? Give examples.
5. How to assign structures and compare structures? What is special about these operations on structures?

# PRACTICAL WORK № 12

## Topic: Input/output with files

**Purpose:** learn how to create programs using input and output to a file.

**Theoretical information**

C++ provides the following classes to perform output and input of characters to/from files:

ofstream: Stream class to write on files

ifstream: Stream class to read from files

fstream: Stream class to both read and write from/to files.

The fstream library allows us to work with files.

To use the fstream library, include both the standard <iostream> AND the <fstream> header file:

Example

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

To write to the file, use the insertion operator (<<).

Example

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
  // Create and open a text file
  ofstream MyFile("filename.txt");
  // Write to the file
```

```
  MyFile << "Files can be tricky, but it is fun enough!";
  // Close the file
  MyFile.close();
}
```

Why do we close the file?

It is considered good practice, and it can clean up unnecessary memory space.

To read from a file, use either the ifstream or fstream object, and the name of the file.

Note that we also use a while loop together with the getline() function (which belongs to the ifstream object) to read the file line by line, and to print the content of the file:

**Example**

```
// Create a text string, which is used to output the text file
string myText;
// Read from the text file
ifstream MyReadFile("filename.txt");
// Use a while loop together with the getline() function to read the file line by line
while (getline (MyReadFile, myText)) {
  // Output the text from the file
  cout << myText;
}
// Close the file
MyReadFile.close();
```

These classes are derived directly or indirectly from the classes istream and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files.

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to open a file. An open file is represented within a program by a stream (i.e., an object of one of these classes; in the previous example, this was myfile) and any input or output operation performed on this stream object will be applied to the physical file associated to it. In order to open a file with a stream object we use its member function open:

*open (filename, mode);*

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open member function (the flags are combined).

For fstream, the default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as text files, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

Since the first task that is performed on a file stream is generally to open a file, these three classes include a constructor that automatically calls the open member function and has the exact same parameters as this member. Therefore, we could also have declared the previous myfile object and conduct the same opening operation in our previous example by writing:

*ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);*

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

To check if a file stream was successful opening a file, you can do it by calling to member is_open. This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise:

*if (myfile.is_open()) { /* ok, proceed with output */ }*

When we are finished with our input and output operations on a file we shall close it so that the operating system is notified and its resources become available again. For that, we call the stream's member function close. This member function takes flushes the associated buffers and closes the file:

```
myfile.close();
```

**Example:**

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ofstream fout("D:\\test.txt");
char fio[21];
int vozrast;
cout << "Enter Name: " << endl;
cin.getline(fio, 20);
cout << "Enter age: " << endl;
cin >> vozrast;
fout << "Name: " << fio << endl;
fout << "Age: " << vozrast << endl;
fout.close();
return 0;
}
```

## Tasks

Create a program in which you need to record data about yourself: last name, first name, age, phone, mail. And write this data to a file, then read this data from the file and display it on the console.

# PRACTICAL WORK №13

## Topic: Classes

**Purpose:** development of software for the implementation of algorithms for working with classes

### Theoretical information

Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C++ code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time.

Classes and objects are the two main aspects of object-oriented programming.

C++ is an object-oriented programming language.

Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

Attributes and methods are basically variables and functions that belongs to the class. These are often referred to as "class members".

A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

**Create a Class**

To create a class, use the `class` keyword:

Example

Create a class called "MyClass":

```
class MyClass {        // The class
  public:              // Access specifier
    int myNum;         // Attribute (int variable)
    string myString;   // Attribute (string variable)
};
```

Fig.13.1 Example

Example explained

The class keyword is used to create a class called MyClass.

The public keyword is an access specifier, which specifies that members (attributes and methods) of the class are accessible from outside the class. You will learn more about access specifiers later.

Inside the class, there is an integer variable myNum and a string variable myString. When variables are declared within a class, they are called attributes.

At last, end the class definition with a semicolon ;.

**Create an Object**

- In C++, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.
- To create an object of MyClass, specify the class name, followed by the object name.
- To access the class attributes (myNum and myString), use the dot syntax (.) on the object.

## Example

Create an object called " myObj " and access the attributes:

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};

int main() {
  MyClass myObj;  // Create an object of MyClass

  // Access attributes and set values
  myObj.myNum = 15;
  myObj.myString = "Some text";

  // Print attribute values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  return 0;
}
```

Fig.13.2 Example

**You can create multiple objects of one class:**

## Example

```
// Create a Car class with some attributes
class Car {
  public:
    string brand;
    string model;
    int year;
};

int main() {
  // Create an object of Car
  Car carObj1;
  carObj1.brand = "BMW";
  carObj1.model = "X5";
  carObj1.year = 1999;

  // Create another object of Car
  Car carObj2;
  carObj2.brand = "Ford";
  carObj2.model = "Mustang";
  carObj2.year = 1969;

  // Print attribute values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

Fig.13.3 Example

### C++ Access Specifiers

By now, you are quite familiar with the public keyword that appears in all of our class examples:

```
#include <iostream>
using namespace std;

class MyClass {   // The class
  public:         // Public access specifier
    int x;        // Public attribute (int variable)
};

int main() {
  MyClass myObj;  // Create an object of MyClass

  // Access attributes and set values
  myObj.x = 15;

  // Print values
  cout << myObj.x;
  return 0;
}
```

```
15
```

Fig.13.4 Example

The public keyword is an access specifier. Access specifiers define how the members (attributes and methods) of a class can be accessed.

In the example above, the members are public - which means that they can be accessed and modified from outside the code.

However, what if we want members to be private and hidden from the outside world?

**In C++, there are three access specifiers:**

- public - members are accessible from outside the class
- private - members cannot be accessed (or viewed) from outside the class
- protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about Inheritance later.

Note: It is possible to access private members of a class using a public method inside the same class.

Tip: It is considered good practice to declare your class attributes as private (as often as you can). This will reduce the possibility of yourself (or others) to mess up the code. This is also the main ingredient of the Encapsulation concept, which you will learn more about in the next chapter.

Note: By default, all members of a class are private if you don't specify an access specifier:

Example

```
class MyClass {
  int x;   // Private attribute
  int y;   // Private attribute
};
```

Fig.13.5 Example

**Tasks**

1. Create a class called MyClass.
2. Create an object of MyClass called myObj.
3. Use an access specifier to make members of MyClass accessible from outside the class.
4. Create an object of MyClass called myObj, and use it to set the value of myNum to 15.

**Control questions**

1. What is a class?
2. How is a class different from a structure?
3. How to define a new class?
4. How to define class objects?
5. Which operator is used to access class members through an object?

# PRACTICAL WORK № 14-15

## Topic: Polymorphism. Overload of functions, operators and methods of a class

**Purpose:** to learn the rules of creating overloaded functions; learn to overload unary and binary operators; learn in practice to overload functions.

### Theoretical information

### Function overload

When defining functions in their programs, you must specify the type of value returned by the function, as well as the number of parameters and the type of each of them. Suppose there is a function called *add_values* that works with two integer values, and we need to use a similar function to add three integer values. Then you should create a function with a different name, such as *add_two_values* and *add_three_values* . Similarly, if you wanted to use a similar function to add values of type *float* , you would need another function with another name.

To avoid duplication, the C ++ function allows you to define multiple functions with the same name. In the compilation process, C ++ takes into account the number of arguments used by each function, and then calls exactly the option you want. *Giving the compiler a choice among several functions is called* ***function overload***.

Restrictions on overloaded functions:

1. Any overloaded functions must have different parameter lists (with an argument of this type and a reference to this type being treated as one).
2. It is not allowed to overload functions with matching parameter lists only based on the type of return values.
3. Member functions cannot be overloaded just because one of them is static and the other is not.
4. All enum-types are considered different and can therefore be used to overload functions.

5. The types "array (of something)" and "pointer (of something)" are considered to be identical in terms of congestion. Example,

*void setval (char pz);*

*void setval (char * ptr);*

is an overload error.

But this only applies to one-dimensional arrays. For multidimensional arrays, the second, third, ..... dimensions are considered as part of the data type.

*For example, there will be no error:*

*void setval (char sz []);*

*void setval (char sz [] [4]);*

*Example 1.* The following program overloads a function named add_values. The first definition of a function is two values of type int. The second definition of a function is three values.

```
#include <iostream.h>
 int add_values(int a,int b){
return(a + b);
}
int add_values (int a, int b, int c){
return(a + b + c);
}
 void main(void){
cout<<"200 + 801 = "<<add_values(200,801)<<endl;
cout<<"10 + 21 + 70 = "<<add_values(10,21,70)<<endl;
}
```

*Example 2.* The following program overloads the show_message function . The first function displays a standard message, the parameters are not passed to it. The second displays the message sent to it, and the third displays two messages.

```
#include <iostream.h>
void show_message(void){  cout << "Standard message: "
  << "Learning to program in C "++<< endl; }
```

*void show_message(char *message){*

*cout << message << endl; }*

*void show_message(char *first, char *second){*

*cout << first << endl;*

*cout << second << endl; }*

*void main(void){*

*show_message();*

*show_message("Learning to program in C ++!");*

*show_message( "Overload is cool!!") ; }*

**Operator overload**

*Operator overload is to change the meaning of the operator when using it with a certain class* .

Overloading operators can demand - tyty most common class operations and improve the readability of the program

To overload the operators of the program use the keyword ***operator*** .

If the program overloads an operator for a certain class, the meaning of this operator changes only for the specified class, the rest of the program will use this operator to perform its standard operations.

In general, applications can overwhelm almost all opera - tori C ++, except for the following:

- member selection operator (operator “.”);
- pointer member selection operator (“. *” operator);
- vision expansion operator (“::” operator);
- condition operator (“? :“);
- preprocessor operator (“#”) and preprocessor symbol (“##”).

*Example 3.* The following is a definition of the strings class . This class contains one data element, which is actually a character string. In addition, this class contains several overridden operators:

*#include <iostream>*

*#include <string.h>*

*#include <locale.h>*

```
using namespace std;
class strings
{
  public:
    strings(char *);
    void operator +(char *);
    void operator -(char);
    void show_string(void);
  private:
    char data[256] ;
};
strings::strings(char *str){
   strcpy(data, str);
}
void strings::operator +(char *str){
   strcat(data, str);
}
void strings::operator -(char letter){
   char temp[256] ;
   int i, j;
   for (i = 0, j = 0; data[i]; i++)
         if (data[i]==letter) temp[j++] = data[i];
   temp[j] = NULL;
   strcpy(data, temp);
}
void strings::show_string(void){
   cout << "\n\t"<<data << endl;
}
void main(void){
   setlocale(0,"");
```

```
  strings title( "\n\t Learning to program in C ++ ");
  strings lesson("\n\ Operator overload");
  title.show_string();
  title + " and I'm learning too!";
  title.show_string() ;
  lesson.show_string();
  lesson – 'p';
  lesson.show_string();
  system("pause");
}
```

***Tasks for practical work №14.***

| | |
|---|---|
| 1 | Define the overloaded *square* functions to find the squares of integers, single and double precision floating point numbers. |
| 2 | Define overloaded *triple* functions to triple the values of integers, single and double precision floating point numbers. |
| 3 | Define the overloaded *minimum* functions to find the smallest of the three integers, single and double precision floating point numbers. |
| 4 | Determine the overloaded *maximum* functions to find the largest of the three integers, single and double precision floating point numbers. |

***Tasks for practical work №15.*** Write a class that implements the specified data type according task.

The class must contain:

- a set of constructors to create objects of a certain type (constructor by soaking, constructor with parameters, constructor copy);
- specified operations on class objects using the operation overload mechanism.

Write a program that demonstrates working with objects of the created class.

| № | Subject area | Overload operations |
|---|---|---|
| 1 | Matrix | Subtraction, multiplication by a matrix, addition of an integer |
| 2 | Complex numbers | Sum, product of complex number, product of real number increment |
| 3 | Vector in space | Adding vectors, vector product of two vectors |
| 4 | Plural | Extracting an element, combining sets, intersecting sets |
| 5 | Integers | Increment, decrement, addition, subtraction, logical operations |
| 6 | Vector on the plane | Scalar product, comparison of vectors, multiplication of a vector by a number |
| 7 | Matrix | Addition, multiplication by number, transposition |

## Control questions

1. What is function overload used for?
2. What are the limitations on overloaded functions?
3. How does the compiler work with overloaded functions?
4. Give an example of function overload.
5. What is the meaning of operator overload?
6. Give examples of overloading of unary and binary operators.
7. Which operators cannot be overloaded?

# LITERATURE


1. MSDN Library [Electronic resource]. - Access mode: URL http://msdn.microsoft.com/ru-ru/library/default.aspx - Name from the screen.
2. NET Framework [Electronic resource]. - Access mode: URL https://msdn.microsoft.com/ru-ru/library/w0x726c2(v=vs.110).aspx - Name from the screen.
3. Visual Studio 2015 [Electronic resource]. - Access mode: URL https://msdn.microsoft.com/library/dd831853(v=vs.140).aspx - Name from the screen.
4. B. Stroustrup: A Tour of C++ (Second Edition). July 2018. Addison-Wesley. ISBN 978-0-13-499783-4. 240 pages.
5. B. Stroustrup: Programming -- Principles and Practice Using C++ (Second Edition). May 2014. Addison-Wesley. ISBN 978-0321992789. 1312 pages.
6. B. Stroustrup: A Tour of C++. September 2013. Addison-Wesley. ISBN 978-0321958310. 190 pages.
7. B. Stroustrup: The C++ Programming Language (Fourth Edition). May 2013. Addison Wesley. Reading Mass. USA. May 2013. ISBN 0-321-56384-0. 1360 pages.

# APPENDIX

## Appendix 1. Basic C ++ Headers

C/C++ Header File

1. #include<stdio.h>  (Standard input-output header)

Used to perform input and output operations in C like scanf() and printf().

2. #include<string.h> (String header)

Perform string manipulation operations like strlen and strcpy.

3. #include<conio.h> (Console input-output header)

Perform console input and console output operations like clrscr() to clear the screen and getch() to get the character from the keyboard.

4. #include<stdlib.h> (Standard library header)

Perform standard utility functions like dynamic memory allocation, using functions such as malloc() and calloc().

5. #include<math.h> (Math header )

Perform mathematical operations like sqrt() and pow(). To obtain the square root and the power of a number respectively.

6. #include<ctype.h>(Character type header)

Perform character type functions like isaplha() and isdigit(). To find whether the given character is an alphabet or a digit respectively.

7. #include<time.h>(Time header)

Perform functions related to date and time like setdate() and getdate(). To modify the system date and get the CPU time respectively.

8. #include<assert.h> (Assertion header)

It is used in program assertion functions like assert(). To get an integer data type in C/C++ as a parameter which prints stderr only if the parameter passed is 0.

9. #include<locale.h> (Localization header)

Perform localization functions like setlocale() and localeconv(). To set locale and get locale conventions respectively.

10. #include<signal.h> (Signal header)

Perform signal handling functions like signal() and raise(). To install signal handler and to raise the signal in the program respectively

11. #include<setjmp.h> (Jump header)

Perform jump functions.

12. #include<stdarg.h> (Standard argument header)

Perform standard argument functions like va_start and va_arg(). To indicate start of the variable-length argument list and to fetch the arguments from the variable-length argument list in the program respectively.

13. #include<error.h> (Error handling header)

Used to perform error handling operations like errno(). To indicate errors in the program by initially assigning the value of this function to 0 and then later changing it to indicate errors.

## Appendix 2. Basic C ++ Data Types

| Type | Typical Bit Width | Typical Range |
|---|---|---|
| char | 1byte | -127 to 127 or 0 to 255 |
| unsigned char | 1byte | 0 to 255 |
| signed char | 1byte | -127 to 127 |
| int | 4bytes | -2147483648 to 2147483647 |
| unsigned int | 4bytes | 0 to 4294967295 |
| signed int | 4bytes | -2147483648 to 2147483647 |
| short int | 2bytes | -32768 to 32767 |
| unsigned short int | 2bytes | 0 to 65,535 |
| signed short int | 2bytes | -32768 to 32767 |
| long int | 8bytes | -2,147,483,648 to 2,147,483,647 |
| signed long int | 8bytes | same as long int |
| unsigned long int | 8bytes | 0 to 4,294,967,295 |
| long long int | 8bytes | -(2^63) to (2^63)-1 |
| unsigned long long int | 8bytes | 0 to 18,446,744,073,709,551,615 |
| float | 4bytes | |
| double | 8bytes | |
| long double | 12bytes | |
| wchar_t | 2 or 4 bytes | 1 wide character |

# Appendix 3. Basic format specifiers for the printf function

| SPECIFIER | USED FOR |
|---|---|
| %c | a single character |
| %s | a string |
| %hi | short (signed) |
| %hu | short (unsigned) |
| %Lf | long double |
| %n | prints nothing |
| %d | a decimal integer (assumes base 10) |
| %i | a decimal integer (detects the base automatically) |
| %o | an octal (base 8) integer |
| %x | a hexadecimal (base 16) integer |
| %p | an address (or pointer) |
| %f | a floating point number for floats |
| %u | int unsigned decimal |
| %e | a floating point number in scientific notation |
| %E | a floating point number in scientific notation |
| %% | the % symbol |