150 C++ Programming Assignments. Variants of tasks & Examples of Code

Tatyana Sopronyuk

150 C++ Programming Assignments. Variants of tasks & Examples of the code

TATYANA SOPRONYUK

Originally published in Ukraine, Chernivtsy, 2014

Translated by Nonna Shulga

ISBN-13: 978-1515254065

ISBN-10: 1515254062

The collection of tasks is an appendix to the teaching guidance “Object-oriented programming in C++” (http://amzn.com/B00NZ6M7L6). It contains problems and samples of their solutions using C++ programming language.

The textbook is intended to be used in the computer science course of “Object-oriented programming in C++”. The tasks, presented in the collection, covers fundamental concepts in object-oriented programming: objects, classes, and inheritance.

The purpose of the given collection is to teach how objects are instantiated from a class, and how new classes are inherited. It introduces constructors, public and private methods, abstract, base and inherited classes, hierarchy of classes, method overloading.

In order to help the students the textbook demonstrates codes of similar programs for all the types of assignments.

This textbook is intended for educational purposes.

150 C++ Programming Assignments. Variants of tasks & Examples of the code: Textbook / Authored by Tatyana Sopronyuk, Translated by Nonna Shulga: CreateSpace, 2015. – 73 p.

ISBN-13: 978-1515254065 (CreateSpace-Assigned) **ISBN-10: 1515254062**

TABLE OF CONTENTS

FOREWORD

ASSIGNMENT №1. OVERLOADING OF OPERATIONS AND FUNCTIONS. FUNCTIONS TEMPLATES Variants of the tasks

Examples of the code

ASSIGNMENT №2. STRUCTURES. OUTER FUNCTIONS AND MEMBER-FUNCTIONS OF THE STRUCTURE Variants of tasks

Examples of the code

ASSIGNMENT №3. STRUCTURES AND CLASSES

Variants of the tasks
Examples of the code

ASSIGNMENT №4. CREATION OF CLASSES

Variants of the tasks
Examples of the code

ASSIGNMENT №5. CREATION OF CLASSES, IMPLEMENTING THE EXISTING CLASSES. INHERITANCE. AGGREGATION Variants of the tasks

Examples of the code

ASSIGNMENT №6. ABSTRACT CLASSES

Variants of the tasks
Examples of the code

LIST OF LITERATURE

FOREWORD

Object-oriented programming is based on the concept that a program can be created as a collection of objects working together. This methodology focuses on the connections between objects rather than the details of their realization. All the OOP languages are based on three main ideas: encapsulation, polymorphism and inheritance [13, 18, 19, 22-30].

In C++ encapsulation is realized with the help of classes. Access to the code and data within the capsule is controlled by an interface which is given to the user.

The purpose of polymorphism, according to OOP, is the usage of one name for the task of the actions, general for the class. The exertion of every concrete action will be defined by the type of the data. In more general sense the concept of polymorphism means that it is possible to create general interface for the groups, common by the content and not by the realization of actions.

The main thing in polymorphism is the fact that it allows to manipulate with the objects of different complexity by means of creating for them general standard interface for the realization of identical actions.

The inheritance is a process with the help of which one object acquires the peculiarities of the other one. To be more exact, the object can inherit the peculiarities of the other object and add to them features, characteristic only for its own.

The collection of tasks is an appendix to the teaching guidance [22] (http://amzn.com/B00NZ6M7L6) “Object-oriented programming in C++”. It contains problems and samples of their solutions using C++ programming language.

The textbook consists of 6 parts. Every part contains variants of assignments, referring to the particular theme, and the examples of the code, meant to assist the student while writing the program. Codes contain the commentaries and the results of the work.

Themes, considered in the given collection, include overloading of functions and operators, template functions, work with the structural types, creation of classes (data-member, functions-member, constructions, destructor and itc.). We also suggest creating classes (aggregation and inheritance) for the extending possibilities of existing classes and the notion of the base and derivative classes, multiple inheritance, building of the

hierarchy of classes, usage of virtual functions, abstract classes, polymorphic functions, virtual inheritance.

In the textbook we have introduced the assignments for the course OOP in C++. Some assignments (№ 2, № 3 and № 6) are based on the ideas from [11, 27]. Other assignments belong to the author [22, 23].

Assignment №1. Overloading of operations and functions. Functions templates

Task

Overloading of the functions for the different types of input data or create functions templates, or overload the operations for the types, marked by the user (new types to assign in the way of structures) [23].

Variants of the tasks

1. To create the function, receiving the text (type **char***) at the output and transforming it by creating: rejects the repeated spaces. The function must return the number of rejected white spaces. To convey the text into the function by reference. To overload the operation "<<" the output of the changed string, in which to call the created function.

2. To create the structure, defining the triangle on the surface. To overload the operations:

- "++" – shift of the triangle by 10 pixels to the right;
- "--" – shift of the triangle by 10 pixels to the left;
- "<"– comparison of two triangles areas.

3. **To create the structure, defining the vector on the plane. To overload the operations:**
   - **"!" – calculation of the vector, received from the assigned by the turn to 90 degrees counter clockwise;**
   - **"-"– calculation of the vector, contrary to the assigned and having the same length with it;**
   - **"++"– normalization of vector.**

4. To create the functions, inverting the arrays, consisting of the elements of the type **int**, **double**, and as well – the function, inverting the words in

the sentence in the reversed order. To use the overloading of functions and create the template.

5. To create the functions, comparing integers, strings of symbols, vectors norms and return.

- **0**, if the data is equal;
- **1**, if the first is more/longer, than the **second**;
- **-1**, if the first is less/shorter, than the **second**.

To use the overloading of functions.

6. To create the functions, printing the square roots from the real and complex number, the norm of the vector. To use functions overriding.

7. To create the functions, finding the maximal array element, consisting of the elements of the type **int**, **double**, and as well – the function, finding the maximal word length in the string (type **char***). To use functions overloading and create the function template.

8. To define the operations "++", calculating the determinant of the matrix by Gauss method and vector norm.

9. Overloading the operations “<<”, “>>” for the matrixes input and the output of the matrixes, transposed to them. Matrixes, rectangular in size **n×m** with the integer and real elements. To use the templates.

10. To define the operation of the logical subtraction of two lines of symbols **S3=S1-S2**. Line **S3** consists of the symbols, belonging to string **S1** and not belonging to **S2**. To take into account the number of identical symbols in both strings. To use the operators overloading.

11. To define operations "+", "-", "*", "/" for a complex number. Test each operation separately. Calculate the expression **z**=(**x+y**)/(**x-y**)+**xy**/(**x+y**). To use the operators overloading.

12. To define operations "+", "-", "*" for the matrixes **(nxn).**Test each operation separately, compared with the results, obtained in the

environment **MathCad.** Calculate the expression **D=A*B+C*A-A*C**. To use the operators overloading.

13. To define operations "+", "-", "*" for the vectors of dimension **n.** Test each operation separately. Calculate the expression **d**=(**a+b**)(**a-b**). To use the operators overloading.

14. To define the logical addition operation of two character strings **S3=S1+S2**. Line **S3** consists of characters, which belong to at least one line **S1** or **S2.** Take into account the number of the same characters in both lines. To use the operators overloading.

15. To define the logical operation of multiplication of two character strings **S3=S1*S2**. Line **S3** consists of characters belonging lines **S1** and **S2** simultaneously. Take into account the number of the same characters in both lines. To use the operators overloading.

16. Three numbers (x, y, r) define a circle on the plane. To define operations "+", "-", "*" for two circles:

- "+"- the area of association;
- "*" - cross-sectional area;
- "-" - the area difference.

To use the operators overloading.

17. Four numbers (**x1**,**y1**,**x2**,**y2**) define a rectangle on the plane.

(**x1**,**y1**) (**x2**,**y2**)

To define the operation "+", "-", "*" for two rectangles:

- "+"- the union area;
- "*" - cross-sectional area;
- "-" - the area of difference.

To use the operators overloading.

18. The array of numbers **A**(n) contains the polynomial coefficients $a_0 + a_1 x + A_2 x^2 + ... + a_{n-1} x^{n-1.}$ To define operations "+", "-", "*" of

polynomials. To use the operators overloading.

19. The array of numbers **A**(n) defines a polynomial $a_0 + a_1 x + a_2 x^2 + ... a_{n-1} x^{n-1}$. To define the operation "++" - calculation of the polynomial value in the point. The coefficients are stored in memory as a list. To use the operators overloading.

20. To overload the operations "<<" and ">>" for input and output of a complex number, and operation "==" comparison of two modules of complex numbers.

21. To overload the operation "<<" and ">>" for the input and output structures. In structure the data about the number of pairs in each day of the week in the computer class are stored.

22. To overload the operation "*" for the multiplication of two complex numbers, of real and complex numbers and calculating the scalar product of two vectors.

23. For the symbols strings overload the operation

- "**-**"– unary minus (reverses the string in place),
- "*" – multiplication of the integer **k** by line **s.**

If **k**>0, then as a result, the string **k** repetitions of the string **s** is obtained**.** If **k**<0, then as a result the string **k** repetitions of inverted strings **s** is obtained.

24. To define the functions:

- **a** & **b** – vector multiplication of vectors;
- ! **a** – normalization of vector.

To calculate the expressions **a** & **a**, **a** & **b**, ! **a**.

25. To overload the operation"<<" and ">>" for the input and output of the polynomials of degree **n** with real coefficients.

26. To define operations "+", "-", "*", "/" for fractions **P /Q**, where **P** – integer, **Q** – natural numbers. Test all the operations separately. Calculate the expressions **a/d**-(**a+b**)/(**cd**), **a**, **b**, **c**, **d**- fractions.

27. To define the structure, denoting the rectangle on the plane. Define the operations "<<", ">>" for the rectangles:

- ">>" - input the information about the coordinates of the rectangle and the name;
- "<<" - display the rectangle (name, coordinates, area).

28. The array of numbers **A**(n) defines a polynomial with integer coefficients $a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1}$. To define the operators ! - calculation the derivative of a polynomial, & - finding the original polynomial.

29. To overload the operations for the sets, saved in the memory as the binary vectors of dimension **n**:

- "+" – union of the sets;
- "*" – crossing of the sets;
- ">>" – input of the sets.

The single vector of the binary vector denotes the presence of the element, corresponding to its index in the plural; zero one – its absence.

30. To overload the operations for the sets, saved in the memory as the binary vectors of dimension **n**: (see variant 29), overload the operation

- "-"- difference of the sets;
- "!" - the complement of the sets;
- "<<" - the output of the set.

31. Create functions, finding the length of the vector on the plane, the length of the vector in space, sentence length (the number of words). Use the overloading function

32. Create functions, determining the height of the trapezoid, the largest from the parallelogram heights, and the minimal from the triangle heights. Use the function of overloading.

33. To define the multiplication "*" of matrix with the dimension **n×n** by the vector with the dimension **n** and vice versa (vector by the

matrix), scalar multiplication of vectors with the dimension **n.**

34. The arrays of numbers define the polynomials with integer coefficients **A**(n) = $a_0 + a_1 x + A_2 x^2 + ... + a_{\mathbf{n-1}} x^{\mathbf{n-1}}$ and **B**(m) = $b_0 + b_1 x + b_2 x^2 + ... + b_{m-1} x^{\mathbf{m-1}}$. Define the operations "/", "%" - the division of these polynomials. The result - two polynomials **P**(r) and **Q**(l), such as **A**(**n**) = **P**(**r**)*B(**m**) +**Q**(**l**).

35. To overload such operations for the correct polygon on the plane (**n** – the number of tops, **a** – the length of the side) to overload such operations:

- "++" – defining the radius of the circle circumscribed round the polygon;
- "--" – defining the radius, inscribed into the polygon circle;
- "<" – comparison of two polygons areas.

Examples of the code

Example of operators' overloading for two structural types

The task: Redefine the operations for a vector of real numbers and a string of different symbols:

- "*" – scalar product of vectors;
- "*" – multiplying a vector by a scalar and the other way round;
- "*" – multiplying of the strings (in the string with the result only common symbols remain);
- "<<", ">>" – input and output of the array.

```
#include <stdio.h> #include <iostream.h> #include <string.h>
struct TStr {
   char s[80]; int lenr; };
struct TVec {
   float v[50]; int lenv; };
```

```
float operator*(TVec v1, TVec v2) {
if(v1.lenv == v2.lenv) {
    float s=0.; for(int j=0;j<v1.lenv;j++) s+=v1.v[j]*v2.v[j]; return s; }

    else {
    cerr<<"\ndimension is not coincided "<<endl; return 0; }
}

TStr operator*(TStr s1, TStr s2) {
    TStr s3; int k=0;
    for(int i=0;i<s1.lenr;i++) for(int j=0;j<s2.lenr;j++) if(s1.s[i]==s2.s[j]) s3.s[k++]=s1.s[i];
s3.s[k]='\x0'; s3.lenr=k; return s3; }

TVec operator*(int k, TVec v1) {
TVec v2;
for(int i=0;i<v1.lenv;i++) v2.v[i]=k*v1.v[i]; v2.lenv=v1.lenv; return v2; }

TVec operator*(TVec v, int k) {
return k*v; }

ostream& operator<<(ostream& os, TVec v){
os<<"\nVector output "<<endl <<"\nDimension n="<<v.lenv<<endl;
for(int i=0;i<v.lenv;i++) os<<v.v[i]<<" ";
return os;
}

istream& operator>>(istream& is,TVec& v) {
cout<<"\nVector input "<<endl<<"\nDimensioin n=";size is>>v.lenv;
cout<<"\nElements input "<<endl; for(int i=0;i<v.lenv;i++) is>>v.v[i]; return is;
}

int main()
{
TStr s1,s2,s3; TVec v1,v2,v3,v[5];
strcpy(s1.s,"abcd"); s1.lenr=strlen(s1.s);
strcpy(s2.s,"aefdl"); s2.lenr=strlen(s2.s);
s3=s1*s2;
```

```
cout<<s1.s<<endl; cout<<s2.s<<endl; cout<<s3.s<<endl;
cin>>v1>>v2>>v3;
float f=v1*v2;
cout<<f<<endl; cout<<3*v3<<endl;
int i;
for(i=0;i<5;i++) cin>>v[i];
for(i=0;i<5;i++) cout<<v[i]*i;
cin.get();
return 0;
}
```

As it is seen from the example, the work with the vectors is organized more convenient, than with the lines on account of the written operations of the overloaded stream vector input and output.
In order to have the possibility to use the overloading of the operators, it would be necessary to put the array and line into structures.

Remark, that the latter example illustrated the realization of 4 operators with the same name **operator*** which the compiler distinguishes by the input parameters.

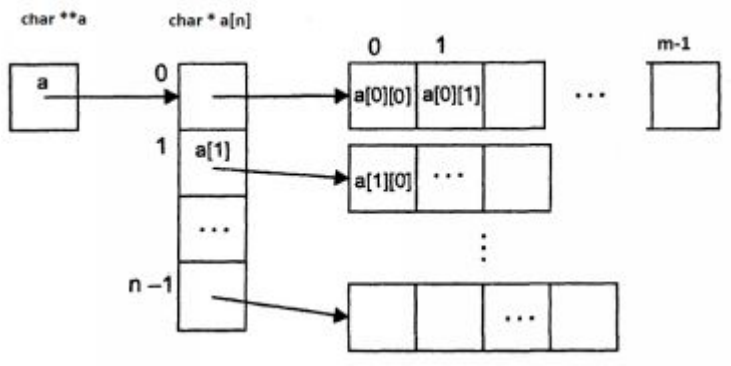
Example of overloading functions and usage of function templates

The task: create the functions sorting the dynamic arrays, made up of the elements of the types **int** , **float** or **char** *.

The length of the arrays is to be given at input.

Use the overloading of functions and the templates.

The algorithm for the input of dynamic array of variable length lines may be illustrated by the diagram, given below.




```
#include <stdio.h> #include <string.h> #include <iostream.h> typedef char* str;
// template for the input of dynamic number arrays // without denoting the type the “reference” won’t work template <class T>int input(T*& a) {
    cout<<"\nInput the array size "; int n; cin>>n; a=new T[n]; cout<<"\nInput the array elements "<<endl; for(int i=0;i<n;i++) cin>>a[i]; return n; }

//template for the output of // dynamic number and line arrays template <class T> void output(T* a, int n) {
    for(int i=0;i<n;i++) cout<<a[i]<<" "; cout<<endl; }

//template for the exchange of the values of two // variables template template <class T> void swap (T & a, T & b) {
    T c=b; b=a; a=c; }

//template for the sorting of two number arrays template <class T> void sort(T* a, int n) {
    int check; do {
    check=0; for(int i=0;i<n-1;i++) if(a[i+1]<a[i]) {
    swap(a[i+1],a[i]); check=1; }
    n--; }while(check); }

//function for the input of dynamic array of lines // of variable length
int input(str*& a) //without referring it won’t work {
cout<<"\n Input the number and maximum length of the lines ";
    int n,m; cin>>n>>m; a=new str[n]; str buffer=new char[m+1]; cout<<"\nInput lines "
<<endl;\ for(int i=0;i<n;i++) {
    cin>>buffer; a[i]=new char[strlen(buffer)+1]; strcpy(a[i],buffer); }
    delete[] buffer; return n; }

//function for sorting of dynamic array of lines of //variable length void sort(str* a,int n) {
```

```
int check; do {check=0; for(int i=0;i<n-1;i++) if(strcmp(a[i+1],a[i])<0) {
swap(a[i+1],a[i]); check=1; }
n--; }while(check); }

// template for the input, sorting and output of dynamic array numbers or the lines of variable
length template <class T> void InputSortOutput() {
T mas;
int size=input(mas); sort (mas,size); output(mas,size); }

int main(void) {

// specification of templates t InputSortOutput<str*>(); InputSortOutput<int*>();
InputSortOutput<float*>();
cin.get(); return 0;
}
```

In the latter program for the work with the arrays of integer and real numbers the templates have been created. Since dynamic array of lines of variable length is inputted and sorted not in the same way as number arrays, then for their input and sorting, the functions have been written in the way when the names of these functions coincide with the names of the corresponding templates for number arrays. But the output of all three types of arrays can be realized in the same way, therefore only one template has been created.

At last the template **InputSortOutput** has been created. It is thrice concretized for the work with all the necessary types of dynamic arrays.

Assignment №2. Structures. Outer functions and member-functions of the structure

Task

It is accepted to call the structure, consisting of two fields with the names **first** and **second,** as ***couple-structure***.

It is necessary to realize the own data-type by means of this structure [11].

To create some objects (**ob**, **ob1**, **ob2**, …) of such a structural type.

To realize in all the tasks:

- the input of structural variable (function **input**());
- the function **check**(), insuring checking the values, input by the function i**nput**() as correct;
- output on the screen the input information and the result of the work (function **output**()).

While inputting parameters error values one should output the corresponding announcement and again suggest inputting the necessary values.

To realize the program in two variants

- to describe by means of the **language C** all the necessary functions as ordinary outer functions;
- to describe the functions, operating by ***one object***, as member-functions of the structure **(language C++)**; for the input/output of the information, implementing the projects **cin**, **cout**.

Remark: for the realization of some own structural types there may arise the necessity in creating one more additional structural type, by means of which the fields **first** and **second** will be defined.

Variants of tasks

1. Square function **y=-x²+Ax+B**. Field **first** – real number, coefficient **A**; field **second** – real number, coefficient **B**. To realize the methods **argument**(**ob**) – calculation for the assigned in the corresponding values **x**, **isCrossing**(**ob**) – definition, or the parabola is crossed with the axis **OX**, **sim**(**ob**) – transformation of the parabola into the parabola, symmetric to it, relating to the axis **OY**.

2. Linear function **y=Ax+B**. Field **first** – real number coefficient **A**; field **second** – real number, coefficient **B**. To realize the methods **function**() – calculation for the assigned **x** value of the function **y**, **isCrossing** (**ob1**,**ob2**) – definition, if two direct lines are crossed.

3. Complex number. Field **first** – real number of the complex number, field **second** – imagery part. To realize the methods of finding the root of **n** power from the complex number, finding the argument of the complex number **compare**(**ob1**,**ob2**) – comparing two objects by the module.

4. Field **first** - fraction (numerator, denominator); field **second** – integer - the degree. To realize the method **power**() - raise the number in the **first** field to the power in the **second** field. The method should work right for all valid values **first** and **second**. Do not use function **pow**() from the library math.

5. Field **first** – positive integer, numerator; field **second** – positive integer, denominator. To realize the method **ipart**() – indicating the integer part of the fraction **first/second** and method **add**() – finding the fraction, when, by its addition, the integer, nearest to the values **first/second** is received. Function **check**() must check the equality of the denominator to the zero.

6. Field **first** – positive integer, banknote rating; the rating can acquire the values 1, 2, 5, 10, 20, 50, 100. Field **second** – positive integer, the quantity of the banknotes of this rating. To realize the methods **sum**(**ob**) – calculation of the money sum of

one object **summation**(**ob1**,**ob2**) – calculation of the total money sum of two objects.

7. Field **first** – positive real number, goods price; field **second** – positive integer number, quantity of the pieces of goods. To realize the method **cost**(**ob**) – calculation of the price of one type of the goods, **costTwo**(**ob1**,**ob2**) – calculation of the total price of two objects.

8. Field **first** – positive integer, caloric content 100 g. of the product; field **second** – positive fraction number, the product mass in kilograms. To realize the methods of calculation of one product total caloric content and **compareTwo**(**ob1**,**ob2**) – comparing two products total caloric content.

9. Field **first** – fraction number (numerator, denominator) the range border, field **second** – fraction number (numerator denominator) – the range border to the right to realize the method **rangeCheck**() – checking assigned fraction number as relating to the range **[first**, **second**].

10. Field **first** – integer, the left range border, belongs to the range; field **second** – integer, range border to the right, doesn't belong to the range. The couple of numbers assigns half-opened interval **[first**, **second**). To realize the methods **rangeCheck**() – checking of the assigned number as relating to the range, **rangePrintUnion**(**ob1**,**ob2**) – output all integer numbers of two ranges combination.

11. Time interval is specified fields: **first** - a positive integer, hours; Field **second** - a positive integer, minutes. To realize the methods: **minutes**() - transfer time into minutes, **shiftForward()** and **shiftBack()** - increase/reduce time period for 20 minutes.

12. The field **first** - a real number; field second – integer, degree. To realize the method **power**() - raise the number **first** to power **second**. The method should work right for all valid values **first** and **second**. Do not use function **pow**() from the library math.

13. Linear equation **Ax+B=0**. Field **first** – fraction number (numerator, denominator), coefficient **A**; field **second** – fraction number, coefficient **B** (numerator, denominator). To realize the method **root**() – calculation of the linear equation root. The method must check the identity of the coefficient **A** with zero.

14. The field **first** - a real number, x coordinate of a point on the plane; field **second** - real number, y coordinate of a point on the plane. To realize the methods **distance**() - calculate the distance from the point to the origin and **distance Two**(**ob1**,**ob2**) - calculate the distance between two points.

15. The field **first** - a positive real, cathetus of right-angled triangle; field **second** - a positive real number, height **h,** dropped on the hypotenuse of a right-angled triangle. To realize the methods **hypotenuse**(**ob1**) - calculate the hypotenuse, **compare(ob1,ob2)** - compare two areas of objects.

16. The field **first -** positive fractional number, salary; field **second** - a positive integer, count of working days in the month. To realize the method **sum**() - calculate accrued sum for the given number of days for the assigned number of the days in a month

salary / number days in month × *worked days*.

17. The field **first** - a positive number, the duration of a call in minutes; field **second** - positive fractional number, the price of one minute in UAH. To realize the methods **cost**() - calculate

the total price of the conversation, **compare**(**ob1**,**ob2**) - compare the cost of two conversations.

18. The field **first -** integer part of the decimal number; field **second** – the positive integer, fractional part of decimal number. To realize the method **multiply**() - multiply it by random real number with two digits after the decimal point. The result of the multiplication – variable of own structure type. The method should work correctly for any valid values **first** and **second**.

19. The field **first** - the positive integer, coordinates of the cursor horizontally; Field **second** - the positive integer, coordinates of the cursor vertically. To realize the methods: **changex**() - change of horizontal coordinates of the cursor; realize method **changey() -** change vertical coordinates of the cursor, **changexy**() - change both cursor coordinates. Methods should check the exit outside the screen.

20. The field **first** - a positive integer, integer part of the number; Field **second** - a positive integer, fractional part of number. To realize the method **multiply**() – multiplication on random integer with type **int.** The result of multiplication – the variable of the structural type. The method should work right for all valid values **first** and **second**.

21. A pair of distinct positive numbers. To realize the method **gcd(ob)** – the greatest common divisor of **first** and second**, lcm(ob) -** the least common multiple of **first** and **second, compareTwo(ob1,ob2)**- compare the coincidence of the greatest common divisors of two objects.

22. Item $\mathbf{a_j}$ decreasing geometric progression is calculated by formula

$\mathbf{a_j = a_0 r^{j,}\ j = 0, 1, 2,...}$

Field **first** - positive real number, the first element of progression $\mathbf{a_{0;}}$ field **second** - real number **r** between 0 and 1. To realize the method **element** () - calculate **j** element of progression, **sum() –**for the calculation of the total of the first **j** progression elements, **sumAll()-** for calculating the total of progression elements.

23. The field **first** -a positive integer, year; Field **second** - a positive integer from 1 to 12 - month number. To realize the methods **daysCount(ob)** – calculate the number of days of the assigned month (consider that the year can be leap), **monthsCount(ob1,ob2)** – calculate the number of months between two dates.

24. Fields **first** and **second** – real numbers, which determining the coordinates of a polar coordinate system. To realize the methods **getX**() and **getY**() – calculating the coordinates in a rectangular coordinate system by the relevant axes, **distanceTwo(ob1,ob2)** – calculate distance between two points.

25. The number with the neighborhood is assigned by the fields **first** and **second,** which determine the coordinates of the point in polar coordinates. To realize the methods **isRange(ob1, ob2)** –checking, if one number from neighborhood gets into the neighborhood of the other number with the neighborhood, **distanceTwo(ob1,ob2)-** calculate minimal distance between two objects (their boundaries).

Examples of the code

Example structure with member-data and outer functions (C code)

```
// C Code
// structure TDate with member-data //d-day, m-month, y-year
struct TDate {
    int d,m,y; };
//function of initialization of variable date with the //values d,m,y
void init (struct TDate* date, int d, int m, int y) {
    date->d =d; date->m =(m % 12)>0 ? m % 12 : 12; date->y =2000+y; }
```

```
void addYear(struct TDate* date, int n) { //addition n years to the field in the structure date
date->y +=n; }
void addMonth(struct TDate* date, int n) { //add n months to the field m in the structure date
date->m +=n; }
void addDay(struct TDate* date, int n) { // add n days to the field d of the structure date date-
>d +=n; }
#include <stdio.h> int main() {
// creation of the instances(objects) //of the structure TDate
struct TDate today, birthday,tomorrow; init(&today,5,9,13);
//initialization of variable birthday //with the values d,m,y+2000
init(&birthday,30,13,-8);
   //tomorrow,assignmened with the data of variable today tomorrow=today; //addition one
day to tomorrow addDay(&tomorrow,1); //print of the day of variable tomorrow
printf("%d\n", tomorrow.d); //print of variable birthday printf("%d %d %d ", birthday.y,
birthday.m, birthday.d); getch();
return 0; }
```

Remark that in the function initialization the date's year is given relating to the 2000th year, and the month may be in the range from one to 12. Other check for data correction either in the function initialization, or in the summation functions is not foreseen. Point that the analogue of the above mentioned example may be found [19, 28].

There is no evident connection between the type of the data and functions. Such connection is possible to state, defining the functions as member *struct*:

Example structure with member-data and member-functions (C++ code)

```
// C++ Code // structure TDate with member-data //d-day, m-month, y-year
struct TDate {
   int d,m,y; void void init (int dd,int mm, int yy); void addYear(int n); void addMonth(int n);
void addDay(int n); }; void TDate::init(int dd,int mm,int yy) { d=dd;
   m =(mm % 12)>0 ? mm % 12 : 12; y =2000+y; }
void TDate::addYear(int n) {
   y +=n; }
void TDate::addMonth(int n) {
   m +=n; }
void TDate::addDay(int n) {
   d +=n; }
```

```
#include <iostream.h>
int main() {
// creation instances (objects) of the structure TDate
TDate today, birthday; today.init(5,9,12);
//initialization of the variable birthday birthday.init(30,12,-8);
//creation of the instance of the structure TDate // with the name tomorrow,initialized by the
data // of the variable today
TDate tomorrow=today; //addition a day to tomorrow tomorrow.addDay(1); //print of the day
of the variable tomorrow cout << tomorrow.d << endl; // print of the variable tomorrow cout
<< tomorrow.y <<" " << tomorrow.m <<" "<< tomorrow.d; //print of the variable birthday
cout << birthday.y <<" " << birthday.m <<" "<< birthday.d; cin.get(); return 0; }
```

Here all variables of the structure type will contain both fields, **d** , **m** , **y** and the functions **init** (), **addYear** (), **addMonth** (), **addDay** ().

Since different structures may have function-members with identical names, one has, while defining the function-member, to point out the name of the *struct* together with the operator scope resolution – "::".

Inside of the function-member structure or class the names of the members of the same class may be used without any pointer of the object. The description of the structure *TDate* offers the choice of functions for the work with the date, but remark that not only they have an access to the class *TDate*. It is possible to limit the access using the key word *class* instead of *struct*.

Assignment №3. Structures and classes

Task

To write the class type in all the tasks (at first – the structure, then – the class with the structure implementing). Besides the functions, indicated in the task, the object of the assigned class type must also have the following realized functions [11]:

- initialization of object **init**();
- input from the keyboard **input**();
- output on the screen **output**();
- transforming the object into the string **toPChar**().

All the tasks must be realized in two ways:

- The data type is assigned by the structure with the necessary fields, and the operations are realized as outer non-operated functions, which the objects of the given type receive as arguments;
- The type of the data is assigned by the class. Class data-member is one private variable of the structural type, described in section 1, member-functions of the class are open non-operated functions **init**(), **input**(), **output**(), **toPChar**() and access functions **setStruct**(), **getStruct**() or **setVar**(), **getVar**().

Variants of the tasks

1. To define the type: quadratic function with the coefficients a, b, c ($y=ax^2+bx+c$). To define the functions:

- **shift**() – parabola shift for one unit up;
- **odd**() – even function definition;
- **roots**() – finding the real roots of the equation $\mathbf{ax^2+bx+c=0}$;
- print of the parabola coordinates top $\mathbf{y=ax^2+bx+c}$;
- finding the number of points, crossed with the axis **OX**.

2. To define the type: rhombus, assigned by the length of the side and the sharp angle. To define the functions:

- **compare**() – the operation of comparing two objects by the radius of inscribed circles;
- finding rhombus diagonals;
- finding the area.

3. To define the type: lineal function with the coefficients b, c (y=bx+c). To define the functions:

- **shiftRight**() – shift of the straight line for one unit to the right;
- **shiftUp**() –shift of the straight line for one unit up;
- defining of the angle straight line incline to the abscissas axis;
- defining the point of two straight lines crossing.

4. To define the type: TCircle – the circle on the area. The circle is defined by the center and a radius. To define the functions:

- **compare**() – comparison and coincidence of the circles by the transfer;
- **isIn**() – finding, whether one circle covers the other one;
- finding the length of the arc for the assigned central angle;
- calculation of the segment area for the assigned central angle.

5. The complex number is assigned by the pair of real numbers **(a, b),** where **a** - real part, **b** - the imaginary part. Realize type **TComplex** to work with complex numbers. Mandatory availability features:

- addition **add**(), (**a**, **b**) + (**c**, **d**);
- subtracting **sub**(),(**a**, **b**) – (**c**, **d**);
- multiplication **mul**(),(**a**, **b**) * (**c**, **d**);
- division **div**(), (**a**, **b**) / (**c**, **d**);
- compare **equ**(), (**a**, **b**) = (**c**, **d**);
- complex conjugate **conj**(a, **b).**

6. To define the type **TVector3D**, assigned by three coordinates. Addition and subtraction of vectors, the scalar's multiplication of vectors, multiplication by scalar, comparison of vectors, calculation of vector's length, comparison of the length of vectors must be by all means realized.

7. To define the type **TModelWindow** for the work with the screen windows models. The window headline, the coordinates of the left upper angle, the size along the horizontal, window color, the state “visible/non-visible”, the state "with/without the frame" are assigned as fields. The coordinates and sizes, indicated as the integers. To realize the operations: window transformation along the horizontal, along the vertical; the height change and/or window width, the change of color; the state change, the state questioning. The operations of transformation and the size change must implement the check of the screen frames crossing. The function of output on the screen must demonstrate the state of the object fields.

8. To define the type **TMoney** to work with the sums of money. The number must be submitted in two fields: type **long** for UAH and type **unsigned char** - for pennies. Fractional part (pennies) by the output must be separated from the integer with the comma. Realize addition, subtraction, division the amounts, division the amounts by the fraction, multiplying the amount by the fraction and comparison operations.

9. To define the type **TTriangle** for the introduction of the triangle. The data fields must include angles and sides. It is necessary to realize the operations: obtaining and changes of the data fields, calculation of area, calculation of perimeter, calculation of heights, and also defining the type (equilateral, isosceles or right-angled triangle).

10. To define the type **TAngle** for the work with the angles on the plane, assigned by the size in grades and minutes. One must realize by all means: transfer into the radians, adduction to the range 0-360, increase and decrease of the angle for the angle assigned size, obtaining the sine, comparison of the angles.

11. To define the type **TPoint** for the points on the plane. The coordinates of the point - Cartesian. Moving the point on the axis **X,** moving along the axis **Y,** defining the distance to the beginning of the coordinates, the distance between two points, converting to polar coordinates, comparing coincidence of two points and the other way round, must necessarily be realized.

12. Rational (irreducible) fractions is assigned by the pair of integers **(a, b),** where **a** - numerator, **b** - denominator. Realize type **TRational** to work with rational fractions. Such operations must be realized; · addition **add**(), (**a**, **b**) + (**c**, **d**);; · subtracting **sub**(),(**a**, **b**) – (**c**, **d**); · multiplication **mul**(),(**a**, **b**) * (**c**, **d**); · division **div**(),(**a**, **b**) / (**c**, **d**); · Compare **equ**(), **bigger**(), **smaller**().

The private function of reducing **reduce**(), fractions, which is necessarily called when performing arithmetic operations must be realized.

13. To create the type **TDate** for the work with the dates in the format "year**.**month**.**day". The date is assigned by the structure with three fields of the type **unsigned int**: for the year, month and day. The type must include not less than three initialization functions: by numbers, string, in the way of "year**.**month**.**day" (for example, "2015**.**08**.**28") and the date. The obligatory operations are: calculation of the date by means of the assigned number of days, subtraction of the assigned number of days from the date, defining the leap year, assignment and obtaining of separate parts (year, month, day), comparison of the dates (equal, to, past), calculation of the number of days between the dates.

14. To define the type **TTime** for the work with the time in the format of "hour**:**minute**:**second". The type must include not less than four initialization functions by the numbers, string, (for example, "23**:**59**:**59"), seconds and time. The obligatory operations are: the calculation of the difference between two moments of the time in the seconds, addition the time and the assigned number of the seconds, subtraction from the time the assigned number of seconds, comparison of the moments of the time, transfer into the seconds, transfer into the minutes (with the rounding to the integer minute).

15. To realize type **TFazzyNumber** to work with fuzzy numbers which are represented by three numbers(**x** – $\mathbf{e_1}$, **x**, **x** + $\mathbf{e_2}$). For the numbers **Ob1 =** (**A** – $\mathbf{a_l}$, **A**, **A** + $\mathbf{a_r}$) and **Ob2 =** (**B** – $\mathbf{b_l}$, **B**, **B** + $\mathbf{b_r}$) arithmetic operations are performed, using the following formulas: **Ob1 + Ob2 =** (**A + B** – $\mathbf{a_l}$ – $\mathbf{b_l}$, **A + B**, **A + B** + $\mathbf{a_r}$ + $\mathbf{b_r}$); **Ob1 - Ob2 =** (**A – B** – $\mathbf{a_l}$ – $\mathbf{b_l}$, **A – B**, **A – B** + $\mathbf{a_r}$ + $\mathbf{b_r}$); **Ob1 * Ob2** = (**A * B – B * a**$_l$ **– A * b**$_l$ **+ a**$_l$ *** b**$_l$, **A * B**, **A * B + B * a**$_r$ **+ A ***

$\mathbf{b_r + a_r * b_r}$); **1/Ob1** = (**1**/(**A** + $\mathbf{a_r}$), **1/A**, **1**/(**A** – $\mathbf{a_l}$)), **A > 0**; **Ob1/Ob2** = ((**A** – $\mathbf{a_l}$)/(**B** + $\mathbf{b_r}$), **A/B**,(**A** + $\mathbf{a_r}$)/(**B** – $\mathbf{b_l}$)), **B > 0**.

16. To realize type **TAccount,** which is a bank account. The type must have four fields: the name of the owner, the account number, interest charges and the amount in UAH. While opening a new account, the initialization operation is performed. You must do the following: change the owner of the account, withdraw some money from the account, deposit the money into the account, change interest transfer amount into dollars, transfer the amount in euros, obtain the amount in words.

17. The ratings of UAH may acquire the values 1, 2, 5, 10 20 50, 100, 200, 500. Penny set as 0.01 (1 penny), 0.02 (2 pennies), 0.05 (5 cents), 0.1 (10 cents), 0.25 (25 cents), 0.5 (50 cents). Create the type **TMoney** to work the ums of money. The amount should be represented by fields-ratings, the values of which should be the number of notes ratings. Implement the addition of the amounts, subtraction sums, division of two amounts, division the sum by the integer, multiplication by the integer and comparison operations. The fractional part (pennies), while outputting on the screen, should be separated from the integer with the comma.

18. To realize type **TBankomat,** which simulates the operation of the ATM. In type should contain fields for storing identification number of the ATM information about the current amount of money remaining in the ATM, minimum and maximum amounts that you can remove the client in one day. The amount of money fed fields denominations 1-500 (see variant 13). Implement init method ATM download method notes in ATMs and method of removing certain amounts of money. The method of withdrawal should be done to check the correctness of the amount is removed, it should not be less than the minimum value and greater than the maximum. Method **toString**() must be converted to string the amount of money remaining in the ATM.

19. To create type the **TFraction** to work with fractional numbers. The number should be assigned by two fields: the integer part - long integer with the sign, fractional part - unsigned short integer. To implement arithmetic operations of addition, subtraction, multiplication by the integer operations and comparisons.

20. To create type **TGoods** (goods). The type must contain the fields: the name of the product, the date of registration, the price of the product, the number of items, the invoice number, by which the goods arrived at the warehouse. Implement the methods of price changes of goods, changes in the number of goods (increase or decrease), the calculation of the value of goods. Method **toString**() should look like a value string of goods.

21. To define the type **TBitString** to work with 64-bit strings. the bit line should be assigned by two fields of the type **unsigned** long. Implement all traditional operations for working with bats: **and, or, xor, not.** Implement left shift **shiftLeft**() and right shift **shiftRight**() on the assigned number of bits.

22. To define the type **TLongLong** to work with integers of 64 bits. The number must be represented by two fields: **long** - the older part, **unsigned long** - the younger part. Arithmetic operations, available in **C**++ (without assignment), and comparisons should be realized.

23. To define the type: regular polygon on the plane. The polygon is defined by the number of sides, the center of the circumscribed circle and one vertex on the circumscribed circle. Identify the operations:

- comparison of two polygons by the perimeter;
- finding the radius of the circle, inscribed in the polygon;
- checking, if the assigned point belongs to the circle, inscribed into polygon;
- finding the area of the polygon.

24. To define the type: linear segment on the plane. The segment is assigned by two points - the linear segment ends. To define the operations:

- calculate the length of the segment;
- calculate the angle between two segments;
- check the intersection of two segments;
- check, whether two segments belong to the straight line.

25. To define the type: **TCursor**. The fields are the coordinates of the cursor horizontally and vertically - integer positive numbers, the cursor look

- horizontal or vertical, the cursor size – integer from 1 to 15. To realize the methods, changing the coordinates of the cursor and changes in the cursor size, the quenching method and updating of the cursor.

Examples of the code

Example structure with member-data and outer functions // C++ Code

```
// structure TDate with member-data //d-day, m-month, y-year struct TDate
   {
   int d,m,y; }; //function of initialization of variable date with the //values d,m,y void init
(TDate& date, int d, int m, int y) {
   date.d =d; date.m =(m % 12)>0 ? m % 12 : 12; date.y =2000+y; }
void addYear(TDate& date, int n) { //addition n years to the field in the structure date date.y
+=n; }
void addMonth(TDate& date, int n) { //add n months to the field m in the structure date
date.m +=n; }
void addDay(TDate& date, int n) { // add n days to the field d of the structure date date.d +=n;
}

#include <stdio.h> #include <iostream.h>
int main()
{
// creation of the instances(objects) //of the structure TDate
TDate today, birthday; init(today,5,9,13);
//initialization of variable birthday //with the values d,m,y+2000
init(birthday,30,13,-8);
// creation of the instance of the //structure TDate with the name //tomorrow,initialized with the
data of variable today
TDate tomorrow=today; //addition one day to до tomorrow addDay(tomorrow,1); //print of the
day of variable tomorrow cout << tomorrow.d << endl; //print of variable birthday cout <<
birthday.y << " "
            << birthday.m << " "
            << birthday.d <<endl; cin.get();
return 0;
}
```

Example class with one private variable of the structural type (data-member) and access functions

```
#include <iostream.h> // class with the data-members d-day, m-month,y-year class TDate
{
   int d,m,y; public:

//access functions to the variable m-month void setM(int m){this->m =(m % 12)>0 ? m % 12 :
12;}
   int getM(){return m;}
   char* getM(int); };

char* TDate::getM(int) {
char* season=new char[7]; switch(m)
            {
         case 12:; case 1:; case 2: strcpy(season,"winter"); break; case 3:; case 4:; case 5:
         strcpy(season,"spring"); break; case 6:; case 7:; case 8: strcpy(season,"summer");
         break; case 9:; case 10:; case 11: strcpy(season,"autumn"); break; }
return season; }

int main()
{
TDate day;
day.setM(12); cout<<day.getM()<<endl; cout<<day.getM(1)<<endl; day.setM(55);
cout<<day.getM()<<endl; cout<<day.getM(1)<<endl; cin.get();
return 0;
}
```

In the latter example there are 2 functions with the name **getM ()**. They differ by the input parameter and return the number of the month or the time of the year, depending on the parameter. The function of the access **setM ()** introduces the correct value of the month number.

Assignment №4. Creation of classes

Tasks To create the indicated class [17] in every variant and to test it, having in mind creation of the necessary number of objects in the main part. While using them, to demonstrate all the elaborated operators and functions. To make the part of operators and functions friendly, and the other part – members of the class. To use the static members of the class.

Variants of the tasks

1. To define the class: *triangle on the plane*. It is defined by three tops $\mathbf{A}(\mathbf{X_1},\mathbf{Y_1})$, $\mathbf{B}(\mathbf{X_2},\mathbf{Y_2})$, $\mathbf{C}(\mathbf{X_3},\mathbf{Y_3})$. To create constructors and a destructor.

To define the operations: "<<" – information print about the object; "==", "!=" – comparison of two objects.

To define the functions:

- finding the area and triangle perimeter;
- finding the radius of inscribed and described circles;
- finding the tops angles;
- finding the tops and medians;
- defining the type of the triangle (right-angled triangle, acute-angled, there is a blunt angle, isosceles, scalene);
- graph depiction of the object on the screen;
- finding the centers of the described and inscribed circles;
- return to the assigned angle around the center of the described circle;
- the object shift to the right, to the left.

2. To define the class of *polynomials of* **n** *power with the real coefficients* $\mathbf{P}(\mathbf{x})=a_0+a_1 x+a_2 x^2+\ldots+a_{\mathbf{n}\text{-}1} x^{\mathbf{n}-1}$, x **R**. In order to save the polynomials coefficients to implement the array.

To create constructors and the destructor.

To define the operations: "+", "-", "*" – addition, subtraction and the multiplication of polynomials.

To define the functions:

- the polynomial's print;
- creation of the polynomial's copy;
- finding the polynomial's value in the point;
- finding the part and the remainder from the division of polynomial by polygon;
- calculation of the polynomials **P'(x)**, **P(Q(x))**, **(x-b)P(x)**, **P(x+b)** (**b** is assigned).

3. To define the class*: long integer*. To save the number to implement the array (one element of the array – one number). To create constructors and the destructor.

To define the operations: "+", "-", "*" – addition, subtraction, multiplication of numbers; "/" – the integer division; "%" – the remainder from the division; "==", "!=", ">", "<", "<=", ">=" – comparison.

To define the functions:

- the number print;
- the logic function, denoting, whether the number is equal to zero.

To define, if number $2^{50}+1$ is simple. To print the numbers 2^{10}, 2^{100}, 100!.

4. To create the class*: use the single linked list to save the string of symbols*. To create constructors and the destructor.

To define the operations: "+" – concatenation of two strings; "==", "!=", "<", ">", "<=", ">=" – two strings comparison operations; "<<" – the string print operation.

To define the functions:

- the string copy;
- deleting **n** symbols from the string, starting from the position **Pos**;
- insertion of the string **st1** into string **st2**, starting from the position **Pos**;
- calculating the length of the string;
- revealing the first entry into string **st1,** of the substring **st2**. The result is index of the first entry

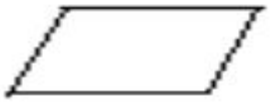
5. To create the class*: the set of points on the plane*. **P**(**X**,**Y**), **X** **Z**, **Y** **Z**. Use single linked list to implement the method, safe for the set. To create constructors and the destructor.

To define the operations: "*", "+", "-" – crossing, union, difference between two sets; "==", "!=", ">=", "<=" – operations of the sets relation; "<<" – the operation of the set print.

To define the functions:

- creation of the set copy;
- defining the number of the set elements;
- defining the point, belonging to the set;
- comparison with the empty set;
- ordering the set as to the distance from the points to the beginning of coordinates;
- ordering the set as to the angle of the turn regarding the axis OX.

6. To define class: *direct parallelogram in the plane*. Parallelogram defined numbers ($\mathbf{X_1}$, $\mathbf{Y_1}$, $\mathbf{X_2}$, $\mathbf{Y_2}$, $\mathbf{X_3}$, $\mathbf{Y_3}$), where $\mathbf{X_i}$ **N,** $\mathbf{Y_i}$ **N.** Create constructor and destructor.

($\mathbf{X_2}$, $\mathbf{Y_2}$) ($\mathbf{X_3}$, $\mathbf{Y_2}$) 

($\mathbf{X_1}$, $\mathbf{Y_1}$)

To define the operations: "+","*", "-" - area associations, intersection and the difference of parallelograms; "==" - comparing parallelograms coincidence when transferring.

To define the functions:

- print information about the parallelogram;
- finding the area and the perimeter of the object;
- graphic image of the object;
- finding the sharp angle at the top;
- the shift of object 10 pixels to the right.

7. To create a class*: the set of integers*. To describe the set, using the binary vector. Create constructors and destructors.

To define the operations: "*"- the intersection of two sets; "+" - the union of the sets; "-" - the difference of two sets; "~" – complement sets; "%" – symmetric difference sets; "= = ""! = ""> = "," <= "- sets relational operators; "<< "," >> "- operations of the input and output of the set.
To define the functions:

- create the copy of the set;
- define the number of the elements in **the set**;
- define, if the number belongs to the set;
- comparison with the empty set;
- finding the minimal element of the set.

8. To define the class of polynomials with complex coefficients $\mathbf{P(x)} = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$, $\mathbf{x}$ **C**, $\mathbf{a_k}$ **=u+iv**, **u Z**, **v Z**. To save the coefficients in memory use the unidirectional linear linked list. Create constructors and destructors.

To define the operations: "+","-", "*" - addition, subtraction, multiplication of polynomials; "==" - comparison operation polynomials.
To define the functions:

- the polynomial print;
- creation of the copy of the polynomial;
- calculation of the value the polynomial at the point **b;**
- the print of the polynomial **P**(x+b) for the given **b.**

9. To define the class of polynomials of arbitrary degree with integer coefficients $\mathbf{P(x)} = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$. To save the coefficients in linear memory use the unidirectional linked list. Create constructors and destructors.

To define the operations: "+","-", "*" - addition, subtraction, multiplication of polynomials; "<<", ">>" - input and output operations of polynomials.
To define the functions:

- checking the equality of two polynomials;
- creation of the polynomial copy;
- finding the value of the polynomial at a point;
- calculation of polynomials **P**(**x+b**), (**x - b)P**(**x**), **P'(x)**for a given **b;**
- graphic image of the function **y = P**(x).

10. To define the class *of polynomials with rational coefficients* $\mathbf{P(x)}=a_0+a_1 x+a_2 x^2+\ldots+a_{n-1} x^{n-1}$, $\mathbf{x}$ $\mathbf{Q}$, $\mathbf{a_k}=\mathbf{u_k}/\mathbf{v_k}$, $\mathbf{u_k}$ $\mathbf{Z}$, $\mathbf{v_k}$ $\mathbf{N}$. To save the coefficients used one-dimensional or two one-dimensional arrays. Create constructors and destructors.

To define the operations: "+"- addition two polynomials; "-" – subtraction of two polynomials; "*" – multiplying of two polynomials, "<<" - the polynomial print; "==" - comparison of two polynomials.

To define the functions:

- creation of the polynomial copy;
- rational point reduction;
- finding the value of the polynomial at a point;
- finding $\mathbf{P'(x)}$, $\mathbf{P(x+b)}$ ($\mathbf{b}$ –given number).

11. To define the class: the *complex number*. Create constructors and destructors.

To define the operations: "+","-", "*", "/" – arithmetic; "=" – assignment; "!" - obtaining conjugated numbers.

To create functions:

- printing complex number;
- copy creation;
- finding the root in the **n-th** degree from a complex number;
- finding of $|\mathbf{z}|$, $\mathbf{Arg\ z}$, $\mathbf{z^a}$ (a-real number);
- the calculation of the assigned accuracy, using the rows.

12. To create the class *matrix of the dimension* **n×n** *with real coefficients*. To define constructors and destructors.

To define the operations : "+","-", "*" – for the matrixes; "<<" - printing matrix; ">>" – input matrix; "==" – comparison of matrices.

To define the functions:

- find determinant of the matrix;
- finding the inverse of the matrix;
- finding the matrix trace;
- function, defining whether the matrix is symmetric.

13. To define the class *of vectors of dimension* **n.** Create constructors and destructors.

To define the operations: "+","-" - addition and subtraction of vectors; "*" - scalar product of vectors; ">>" - input vector; '^' - vector multiplication by the number.
To define the functions:

- print of the vector;
- finding the length of the vector;
- creation vector copy;
- sorting elements of the vector;
- vector normalization.

14. To create the class: *character string is stored as a character string in Pascal*. To save it, the array of characters is used. In the zero element the current string length is used. Create constructors and destructor.

To define the operations: "+"- concatenation of two strings; "%" - the logical addition (only those characters of the second string are added, that are absent in the first string); "==", "! =", "<", ">", "<=", "> =" - operations to compare two strings.

To define the functions:

- print string;
- string copying;
- destruction **N** character string starting from a position **Pos;**
- insertion a string **st1** into the string **st2,** starting from a position **Pos;**
- finding the length of the string;
- define the first input into the string **st1** of the substring **st2.** The result - the number of positions.

15. To define class: *character string*. To maintain the string to use the symbol array with the final symbol '\0'. Create constructors and destructors.

To define the operations: "*"- logical multiplication of two strings (return string, which is the only common symbols); "+" – concatenation of two strings, "==", "! =", "<", ">", "<=" "> =" - operations to compare two strings.

To define the functions:

- print string;
- string copying;

- destruction of **N** characters string, starting with a position **Pos;**
- insert string **st1** into a string **st2,** starting with a position **Pos;**
- determining the length of the string;
- withdrawing from the string the substring with the length of **N** characters, starting from a position **Pos;**
- revealing the first entry in the string line (result - item number);
- verification whether the line recording a positive number;
- converting character string to integer record he is.

16. To define the class: *the circle on the plane*. The circle is defined by three numbers (**x**, **y**, **r**). To create constructors and the destructor.

To define the operations: "+"– the area of two circles union; "–" – the area of two circles difference; "*" – the area of two circles crossing.

To define the functions:

- the print of the information about the circle;
- the areas of the comparison;
- finding the circle area;
- the arc length for the central circle, assigned in the degrees;
- the circle graph depiction on the screen.

17. To define the class: *rectangle on the plane*. It is defined by the coordinates of the point **A(x1**,**y1)** – the first left point – and the points **B(x2**,**y2)** – lower right point of the rectangle (**x1 N**, **y1 N**, **x2 N**, **y2 N)**. To create constructors and the destructor.

To define the operations: "+" – the area of two rectangle union; "–" – the area of two rectangle difference; "*" – the area of two rectangle crossing.

To define the functions:

- the print of the information about the rectangle;
- the comparison of two rectangle areas;
- the comparison of rectangle concatenation at transfer;
- finding the rectangle area and the perimeter;
- the rectangle graph depiction on the screen.

18. To define the class: *long integer*. To save the number use linear unidirectional list. To create constructors and the destructor.

To define the operations: "+","-", "*" - addition, subtraction, multiplication; "/" – integer division; "%" - the remainder of the division.
To define the functions:

- the print of the number;
- finding the amount digits in the number;
- the normalization of (discarding insignificant zeros);
- logical function determining, if the number is equal to zero;
- defining, if the $2^{100}+1$ is prime.

19. To create the class: *regular polygon*. The object is defined in 5 integers **X1**,**Y1**, **R**, **N**. Here **O**(**X1**,**Y1**) – circumscribed circle center, R - radius of the circumscribed circle, N - number of R, A(X1-R, Y1) coordinates of the first peak. To create constructors and the destructor.
To define the operations: "==", "! =" - comparing two objects; "<<" - print information about the object.
To define the functions:

- finding the area and perimeter of the object;
- finding the radius of the inscribed circle;
- finding the angle at the top;
- drawing the object on the screen;
- shift image 20 pixels to the left.

20. To define the class: *plane in space*. The plane is assigned by the general equation **A**, **B**, **C**, **D** coefficients. To create constructors and the destructor, the copy generator.
To define the operation: "^" – the angle between the planes; "||" – the distance between the planes; "==", "!=" – comparison of two planes.
To define the functions:

- information print about the plane;
- checking of the point belonging to the plane;
- defining the coefficients of the normal plane equation;
- checking, if the plane is parallel to the to the coordinates of the planes;
- checking, if two planes are parallel or perpendicular;
- finding the distance between the point and the plane.

21. To define the class*: the broken line on the plane*. It is assigned by the coordinates of the points, saved in the two-dimensional or one-dimensional array. To create constructors and the destructor To define the operation: "+" – concatenations of two broken lines; "<<", ">>" – the broken line output and input (the number of the tops and their coordinates).

To define the functions:

- defining, if the broken line is closed;
- defining, if there are the broken line selfcrossings;
- defining, if the broken line is convex;
- finding the area of the figure, limited by the convex broken line;
- drawing of the broken line.

To make the algorithm easier one can consider all the tops different, besides, probably, the first and the last one.

22. To define the class: *straight line on the plane*. It is assigned by the general equation, **A**, **B**, **C** – coefficients. To create constructors and the destructor, the copy generator.

To define the operations: "^" – the angle between the straight lines; "||" –the distance between the straight lines; To define the functions:

- the information print about the straight line;
- finding two straight lines crossing;
- checking the point assignment to the straight line;
- defining the coefficients of the normal straight line equation;
- check, if the straight line is parallel to the axis coordinates;
- check, if two straight lines are parallel and perpendicular;
- finding the distance between the point and the straight line.

23. To define the class: *point on the plane*. To create constructors and the destructor, the copy generator.

To define the operations: "++"g, "-" - the point shift right/left 10 pixels; "||" - the distance between the points; "<<", ">>" - output and input points; " == ","! = "- comparing two points.

To define the functions:

- finding polar coordinates of the point;

- graphic points on the coordinate plane;
- defining the quarter of the point location on the coordinate plane;
- defining, if 4 assigned points form a circle;
- defining, if 3 assigned points belong to one straight line.

24. To define the class: *broken line in space*. The broken line is assigned by the coordinates of points, stored in an array of **3×**n. To create constructors and the destructor, the copy generator.

To define the operations: "+"- gluing two broken lines; "=" - assignment.

To define the functions:

- print information about the broken line (the number of vertexes and their coordinates);
- defining, if broken line is closed;
- defining, whether it belongs to one plane;
- finding the projections of the broken line on the coordinate plane;
- graphic projections of the broken line at **XOY.**

25. To define the class: *point in space*. To create constructors and the destructor, the copy generator.

To define the operations: "==", "! =" – comparison of the points; "||" - the distance between the points; "=" – assignment; "+", "-" - addition, subtracting of the points.

To define the functions:

- print information about the point,
- finding spherical coordinates;
- defining the point belonging to the coordinate planes;
- defining the equation of the line, passing through two assigned points;
- defining the equation of the plane, passing through three assigned points.

26. To define the class: *integer*. To create constructors and the destructor.

To define the operations: "++"- the amount of the characters digits; "-" - the destruction of the last digits, if they are equal to zero; "!" – rotation of the number (digits – in the inverted order); "&" - division numbers (result - the real number).

To define the functions:

- check, if he number is prime;
- finding the prime divisors of the number;
- expansion of the number into its prime factors,
- the least common multiple of two numbers;
- printing numbers in different number systems (from binary to decimal).

27. To define the class: the *rational number* x = p /q, where p - integer, q - a positive integer number. To create constructors and the destructor.

To define the operations: "+","-", "*", "/" - addition, subtraction, multiplication, division of fractions; "==", "! =" ">", "<", "<=" "> =" - comparison; "<<" - output and ">>" - input.

To define the functions:

- exponentiation: $\mathbf{x^a}$, **a** – integer;
- reduction of rational fraction;
- the object copying.

28. To create the class: *sphere in space*. The sphere is set by the coordinates of the center and radius. To create constructors and the destructor, the copy generator.

To define the operations: "==" - comparing radii; "<<"- print information about the sphere; ,"<=" - determining, if one sphere belongs to the other; "> =" - defining, if one sphere is included into the other; "*" - finding the volume of the intersecting spheres.

To define the functions:

- the shift of spheres on the assigned vector;
- compression of spheres into the assigned number of times;
- the print information about the intersection with the other sphere (one point and its coordinates, many points of intersection).

29. To create the class: *the vector on the plane*. To create constructors and the destructor, the copy generator.

To define the operations: "+"- the sum of vectors; "-" - the difference of vectors; "*" – scalar product; "=" – assignment; "<" - the angle between the vectors.

To define the functions:

- defining whether the vectors are collinear or orthogonal vectors;
- vector printing;
- graphic vector image;
- defining the length of the vector.

30. To create the class: *vector in space*. To create constructors and the destructor, the copy generator.

To define the operations: "&" - the angle between the vectors; "&&" - vector product of vectors; "*" - scalar product; "=" - assigning; "<<" - printing vector.

To define the functions:

- defining if two vectors are collinear;
- defining, if three vectors are coplanar;
- the definition of mixed product of vectors;
- graphic image of vector projections on each coordinate plane;
- defining the length of the vector.

31. To create the class: *ellipse*. Ellipse is assigned by the coefficients **a, b** of the canonical equation. To create constructors and the destructor, the copy generator.

To define the operations: "="- assignment; "==" – comparison; "<<" - print information about the ellipse; "<=" "> =" – inclusion of one ellipse into the other one and vice versa.

To define the functions:

- defining coordinate focuses;
- defining eccentricity;
- focal parameter;

- defining the equations of directrixes;
- graphic ellipse.

32. To create the class: *cylinder*, the lower base of which is parallel to the coordinate plane **XOY.** The cylinder is assigned by the coordinates of the lower base center, by the base radius and height. To create constructors and the destructor, the copy generator.

To define the operations: "=" – assignment; "==" – comparison of cylinder volumes; "<<" – print information about the cylinder, "<=" – defining, whether one cylinder belongs to the other; "> =" – defining the inclusion of one cylinder into the other.

To define the functions:

- cylinder shift for an assigned vector;
- compression cylinder size in the assigned number of times;
- printing information about crossing with the other cylinder (yes, no).

33. To create the class*: the car*. The car is assigned by, the following data: number, make, price, color, name and surname of the owner, year of production, the note of the customs control. To create constructors and the destructor.

To define the operations: "="- assignment; "<", ">" - comparing the price of cars; "<=" "> =" - comparing cars by the year of production; " <<" - print information about the car; "+" - the total price of the cars.

To define the functions:

- customs regulations, depending on the production year;
- the change of color and the owner;
- defining, whether the car had passed the customs control;
- defining, whether the car number is lucky (symmetrical);
- defining the age of the car.

34. To create the class*: the timer*. The timer is assigned by the period of the time, through which the function will be called, and the beginning and the end of work. To create constructors and the destructor. .

To define the operations: "="- assignment; "==" - comparing two timers; "<=" "> =" - comparing two timers periods of time; "<", ">" - comparing two timers beginning of work; ,"<<" - prints information about the timer; "-" - defining the time difference between timers.

To define the functions:

- graphic changes in the time interval, the length of which increases with the time (the function must be called through the time interval, assigned by the timer);
- changes in the time interval;
- defining the number of calls by the timer of the assigned function;
- creation of the timer copy;
- changes of the timer beginning of work.

Examples of the code

Example of the elaboration of the class “Set of the Integer Numbers”

Task : To define the class: set of the integer numbers. To create constructors and the destructor.

To define the operations:

- **'*'** – crossing of two sets;
- **'+'** – unity of two sets;
- **'-'** – difference between two sets;
- **'='** – assignment;
- **'<=', '>=', '!=', '=='** – ratio operations.

To define the functions:

- print of the set;
- the number of elements calculation in the set;
- comparison with the empty set;
- ordering the set;
- the number belongs to the set check;
- finding the maximum element of the set.

Create the necessary number of object sets for the illustration of all the operations and functions. The part of functions and operators are to be

friendly, and the other part – members of the class.

```
#include <vcl.h> #include <windows.h> #pragma hdrstop #pragma argsused #include
<tchar.h> #include <stdio.h>
// the library, containing classes of input-output, // cin, cout, cerr and others.
// enables implementing overloading // input and output operators #include <iostream.h>
#include <conio.h>
typedef int Bool; const Bool NO=0; const Bool YES=1;
class TSetInt {
// closed data-members // access allowed only to the members and friends of the class int* V; //
set element int Sz; // the number of elements V[]
   int Num; // object number Bool Copy; // copy object, or not
//number of object sets, one static variable
//for all the objects static int CountObjects;
// opened data-members public:
   TSetInt(int); // constructor with an argument TSetInt(int,int*); // constructor with
arguments TSetInt(); // default constructor TSetInt(const TSetInt&); // copy constructor
~TSetInt(); // destructor
// binary operators-= class-members // object *this – the first parameter //(conveyed implicitly)
// the first and the second parameter is not changed //(const)
            TSetInt operator*(const TSetInt&) const; TSetInt operator-(const TSetInt&) const;
Bool operator<=(const TSetInt&) const; Bool operator==(const TSetInt&) const; Bool
operator>=(const TSetInt&) const; Bool operator!=(const TSetInt&) const;
// assigned operator // (only the second parameter is not changed) TSetInt& operator=(const
TSetInt&);
//friendly operator friend TSetInt operator+(const TSetInt&, const TSetInt&);
// functions – class-members void printSet()const; int kilElem(){return Sz;}

// inline function Bool isEmpty(); void order(); TSetInt& inSet(int); // possible like this: void
inSet(int); Bool ifInSet(int) const;
// function-friend of the class friend int maxElem(const TSetInt&); };

// initialization of static variable int TSetInt::CountObjects=0;
TSetInt::TSetInt(int sz) {
                        V=new int[sz]; Copy=NO; Sz=sz; Num=++CountObjects; int i=0,el;
// record of the accidental elements into the array set // only one time while(i<Sz) {
   el=random(50); if (!ifInSet(el))V[i++]=el; }
```

```
cout<<"\n Created object (int) N"<< Num <<endl; }

TSetInt::~TSetInt() {
   cout<<"\nDeleting object N"<< Num << " Copy "<<Copy<< " number of objects "<<--
CountObjects <<endl; delete[] V; }

TSetInt::TSetInt() {
   Sz=0; V=NULL; Copy=NO; Num=++CountObjects; cout <<"\nCreated object() N"<<
Num <<endl; }

//Constructor writes down array elements //only once
TSetInt::TSetInt(int sz,int* vek) {
   int* temp=new int[sz]; Copy=NO; Sz=1; Num=++CountObjects; temp[0]=vek[0]; for(int
i=1;i<sz;i++) {
   Bool flag=YES; for(int j=0;j<Sz;j++) if (vek[i]==temp[j]) flag=NO;
if(flag)temp[Sz++]=vek[i]; }
   V=new int[Sz]; for(int i=0;i<Sz;V[i]=temp[i],i++); delete[] temp; cout<<"\nCreated object
(int,int*) N" <<Num<<endl; }

TSetInt::TSetInt(const TSetInt& Obj):
         //the list of initializers Num(Obj.Num), Copy(YES), Sz(Obj.Sz) {
   V=new int[Obj.Sz]; for(int i=0;i<Obj.Sz;i++)V[i]=Obj.V[i]; CountObjects++;
cout<<"\nCreated the copy of the object N"<<Num<<endl; }

TSetInt TSetInt::operator*(const TSetInt& Ob) const {
         //The call of copy constructor TSetInt temp(*this); int k=0;
   for(int i=0;i<Ob.Sz;i++) for(int j=0;j<Sz;j++) if(Ob.V[i]==V[j]) temp.V[k++]=Ob.V[i];
temp.Sz=k; return temp; }

TSetInt TSetInt::operator-(const TSetInt& Ob) const {
   TSetInt temp(*this); int k=0; for(int j=0;j<Sz;j++) if(!Ob.ifInSet(V[j])) temp.V[k++]=V[j];
temp.Sz=k; return temp; }

TSetInt& TSetInt::operator=(const TSetInt& Ob) {
   if (this==&Ob) {
                  cerr<<"\nAssignment mistake"<<endl; return *this; }
   delete[] V; V=new int[Sz=Ob.Sz]; for(int i=0;i<Sz;i++) V[i]=Ob.V[i]; return *this; }
```

```
// there is no field of visibility, // since this operator is not – a member // of the class, but a
friend TSetInt operator+(const TSetInt& Ob1,const TSetInt& Ob2) {
   int* v=new int[Ob1.Sz+Ob2.Sz]; for(int i=0;i<Ob1.Sz;i++)v[i]=Ob1.V[i]; int sz=Ob1.Sz-1;
for(int i=0;i<Ob2.Sz;i++)
// no need to check, // since the constructor has done it //if (!(Obb1.ifInSet(Ob2.V[i])))
v[sz++]=Ob2.V[i]; return TSetInt(sz,v); }

Bool TSetInt::operator<=(const TSetInt& Ob) const {
   if(Ob.Sz < Sz ) return NO; if(Sz==0)return YES; for(int i=0;i<Sz;i++) if (!
(Ob.ifInSet(V[i])))return NO; return YES; }

Bool TSetInt::operator>=(const TSetInt& Ob) const {
   return Ob<=(*this); }

Bool TSetInt::operator==(const TSetInt& Ob) const {
   if(Ob.Sz != Sz) return NO; return(operator<=(Ob)); // better this way: *this<=Ob; }

Bool TSetInt::operator!=(const TSetInt& Ob) const {
   return(!(operator==(Ob))); }

TSetInt& TSetInt::inSet(int a) {
   int i; if(ifInSet(a))return *this; TSetInt Ob(*this); delete[] V; V=new int[++Sz];
for(i=0;i<Ob.Sz;i++) V[i]=Ob.V[i]; V[i]=a; return *this; }

void TSetInt::printSet() const {
          cout<<"\Object N"<<Num<< " Copy "<<Copy<<endl<< "Elements: "; for(int
i=0;i<Sz;i++) cout<<V[i]<<" "; cout<<"\nThe number of objects "<<CountObjects<<endl;
return; }

Bool TSetInt::isEmpty() {
   if(Sz==0)return YES; return NO; }

void TSetInt::order() {
   for(int i=1;i<Sz;i++) {
   for(int j=Sz-1;j>=i;j--) if(V[j-1]>V[j]) {
   int a=V[j-1]; V[j-1]=V[j]; V[j]=a; }
```

```
    }
    return ; }

Bool TSetInt::ifInSet(int a) const {
    for(int i=0;i<Sz;i++) if(a==V[i])return YES; return NO; }

int maxElem(const TSetInt& Ob) // friendly function {
    int a,i; for(a=Ob.V[0],i=1; i<Ob.Sz; i++) if(Ob.V[i]>a)a= Ob.V[i]; return a; }

int main()
{

TSetInt Ob1(4),Ob2(5); Ob1.printSet(); cout<<"\nThe number of elements "<<Ob1.kilElem()
<<" in the object N1"<<endl; cout<<"\nifInSet 4 - "<<Ob1.ifInSet(4)<<" ifInSet 1-"
                                <<Ob1.ifInSet(1)<<endl; cout<<"\nisEmpty "<<Ob1.isEmpty()
                    <<endl;
Ob2.printSet();
TSetInt Ob3(Ob2); Ob3.printSet(); cout<<"\nmax Ob3 "<<maxElem(Ob3)<<endl;
TSetInt Ob4(0); Ob4.printSet(); cout<<"\nisEmpty Ob4 "<<Ob4.isEmpty()<<endl;
int vek[5];
for(int i=0;i<5;vek[i]=6+i,i++); vek[1]=8;
TSetInt Ob5(5,vek); Ob5.printSet(); TSetInt Ob6;
{
// The block of existence for the temporary objects, // Created by the copy constructor
Ob6=Ob5+Ob1;
}

Ob6.inSet(3); Ob6.order();
Ob6.printSet();
{
// The block of the existence of temporary objects, // Created by the copy constructor
Ob2=Ob6-Ob1;
}
Ob2.printSet();
Ob2.inSet(27);
{
```

```
// The block of the existence of the temporary objects, // Created by the copy constructor
Ob6=Ob1*Ob2;
}
Ob6.printSet(); Ob2.printSet();
cout<<" Ob6<=Ob2 ? "<< (Ob6<=Ob2) <<endl; cout<<" Ob2<=Ob6 ? "<< (Ob2<=Ob6)
<<endl; cout<<" Ob2==Ob1 ? "<< (Ob2==Ob1) <<endl; cout<<" Ob2>=Ob6 ? "<<
(Ob2>=Ob6) <<endl; Ob6.printSet(); Ob6.inSet(9);
TSetInt Ob7;
Ob7=Ob2;
Ob7.printSet(); cin.get();
return 0;
}
```

Result of the program work:

```
Created object (int) N1

Created object (int) N2
Object N1 Copy 0
Elements: 49 4 36 44
The number of objects 2

The number of elements 4 in the object N1

ifInSet 4 - 1 ifInSet 1-0

isEmpty 0
Object N2 Copy 0
Elements: 12 33 6 34 36
The number of objects 2

Created the copy of the object N2
Object N2 Copy 1
Elements: 12 33 6 34 36
The number of objects 3

max Ob3 36
```

Created object (int) N4
Object N4 Copy 0
Elements:
The number of objects 4

isEmpty Ob4 1

Created object (int,int*) N5
Object N5 Copy 0
Elements: 6 8 9 10
The number of objects 5

Created object() N6

Created object (int,int*) N7

Deleting object N7 Copy 0 number of objects 6

Created the copy of the object N6

Deleting object N6 Copy 1 number of objects 6
Object N6 Copy 0
Elements: 3 4 6 8 9 36 44 49
The number of objects 6

Created the copy of the object N6

Created the copy of the object N6

Deleting object N6 Copy 1 number of objects 7

Deleting object N6 Copy 1 number of objects 6
Object N2 Copy 0
Elements: 3 6 8 9
The number of objects 6

Created the copy of the object N2

Deleting object N2 Copy 1 number of objects 6

Created the copy of the object N1

Created the copy of the object N1

Deleting object N1 Copy 1 number of objects 7

Deleting object N1 Copy 1 number of objects 6
Object N6 Copy 0
Elements:
The number of objects 6
Object N2 Copy 0
Elements: 3 6 8 9 27
The number of objects 6
Ob6<=Ob2 ? 1
Ob2<=Ob6 ? 0
Ob2==Ob1 ? 0
Ob2>=Ob6 ? 1
Object N6 Copy 0
Elements:
The number of objects 6

Created the copy of the object N6

Deleting object N6 Copy 1 number of objects 6

Created object() N7
Object N7 Copy 0
Elements: 3 6 8 9 27
The number of objects 7

Assignment №5. Creation of classes, implementing the existing classes.

Inheritance. Aggregation

Tasks Implementing the class, elaborated in assignment № 4; to create the new class [17] with the help of one of two ways:

- to create the new class, in which to **inherit the old class**, making its possibilities wider;
- to create the new class, implementing in it data-members – **pointers to the type of the old class (aggregation)**.

By all means to add in the new class the new data-member, for example, the color of the object and its name. To define the constructors, realize them through the call of the class constructors, elaborated in assignment № 4. Other new functions must use to the maximum the possibilities of the code, written before.

The new operators are possible – unary, postfixed and prefixed decrements and increments, operators indexes, call, transformation of the type.

New function-members are possible – the objects graph depiction.

Overloaded functions are possible – input/output (considering the new data-members).

To divide the project into several files, to switch them on by means of the preprocessor directives.

We give below the examples of the tasks conditions for the variant 1 (inheritance) and variant 2 (aggregation), corresponding to the first two conditions of assignment № 4.

Variants of the tasks

1. To create the class: right prism, which foundations are parallel to the coordinate plane **XOY**. For this to inherit the class "triangle on the plane". To add the data-members of the inherited class with the height and the

distance from the lower foundation up to the plane **XOY**. To create constructors and the destructor.

To define the operations:

"=" – assignment;

"==" – the comparison of prisms by the volume; "!=" – the comparison of prisms for the concatenation when the parallel transfer takes place; "<<" – the information print about the prism (coordinates of all the tops).

To define the functions:

Prism shift to the assigned vector;

- finding the volume of the cylinder, encircled around the prism;
- finding the sum of the length of all prism sides;
- finding the area of the prism surface.

2. To define the class: rational function. The data-members – numerator and denominator, – these are the pointers to the objects of the class "polynomial of **n** power with the real coefficients". To create constructors and the destructor**.**

To define the operations:

"+", "-", "*" – addition, subtraction and multiplication of the rational functions; "<<" – the print of the rational function.

To define the functions:

- raising to the integer power of the rational function;
- creation of the rational function copy;
- finding the rational function value in the point;
- finding the rational function derivative.

Examples of the code

Example of aggregation for the realization of the stack

Aggregation insures the relation the whole – the part. The part may be the object of the local class (such a type of aggregation is called the composition) or it may be indicated by a pointer.

Consider the example of aggregation for the realization of the stack [17], which will save any types of the data.

```
// File with an interfaced part of the class Stack.h #ifndef STACKH //if the directive STACKH
is not defined #define STACKH // define it
// the announcement of the class TStack
class TStack
{
        struct TLink
        { // structure, announced in the class
                void* data;
                TLink* next;
                TLink(void* dat, TLink* nxt);
                ~TTLink();
    }* head;//aggregation, using the pointer public:
        TStack();
        ~TStack();
        void push(void* dat);
        void* peek();
        void* pop();
};
#endif // the end of defining the directive STACKH

// File of realization Stack.cpp
#include "Stack.h"
#include <iostream.h>
TStack::TLink::TLink(void* d, TLink* n)
{
data = d;
next = n;
}

TStack::TLink::~TLink() { }
TStack::TStack() { head = NULL; }

void TStack::push(void* d)
{
```

```
        head = new TLink(d,head);
}

void* TStack::peek()
{
        return head->data;
}

void* TStack::pop()
{
        if(head == NULL) return NULL;
        void* rez = head->data;
        TLink* oldHead = head;
        head = head->next;
        delete oldHead;
        return rez;
}

TStack::~TStack()
{
        while(head)pop();
        cout<<"\nTStack empty";
}

// File: Main.cpp
#include <iostream.h>
#include "Stack.h"

int main(int argc, char* argv[])
{
{
TStack my;
int el1,el2,el3;
cout<<"Input elements\n";
cin>>el1>>el2>>el3;

my.push(&el1);
```

```
my.push(&el2);
my.push(&el3);

// delete el3
my.pop();

// print el2 (transformation of the type and // location of the pointer)
cout<<*(int*)my.peek();
}// here the destructor of the class Tstack will work
cin.get(); // delay of the window with the result return 0;
}
```

The program is made up of 3 files: header file **Stack . h** , in which the description of the class **TStack** is contained, the file **Stack . cpp ,** containing the realization of this class, and the file **Main . cpp** with the main function main.

In the class **TStack** the structure **TLink** , is announced in the private section. It saves the object pointer (variable data) and the pointer to the next element (variable next); the pointer head, indicating **TLink** ; constructor **TStack** () and destructor **~TStack** ().

The functions of member-class **TStack** are accessible:

- **push** for addition the object to the stack;
- **peek** for returning the object;
- **pop** for the output of the object from the stack.

Example of inheritance As a base class implement the class **TArray** – dynamic array with the limits of the change of elements' index from 0 till **size-1** . The derivative class – **TRangeArray** with the limits of the change of elements' index from **low** to **high-1** . In both classes overload the index operator .

```
#include <string.h>
#include <conio.h>
#include <iostream.h>

//the name's own space with the types
// TArray, TName, TNameArray
namespace My
```

```
{
// the base class – dynamic vector
// (index - з 0 до size-1)

class TArray {
    int* v;
    int size;

protected:
// access functions for the derivative classes int getSize(){return size;}
    int& getV(int i)const { return v[i]; }

public:
                TArray(int); // constructor TArray(const TArray&); // copy constructor
                ~TArray() {delete[] v;}; // destructor TArray operator+(const TArray&) ;
                TArray& operator=(const TArray&); int& operator[](int); };

TArray::TArray(int n)
{
if (n<=0)
    {
    cerr<<"\nWrong vector's size"; return; }
size=n;
v=new int[n];
}

TArray::TArray(const TArray& a)
{
v=new int[size=a.size];
for(int i=0;i<size;i++)
    v[i]=a.v[i]; }

int& TArray::operator[](int i)
{
if(i<0 || size<=i)
    {
    cerr<<"index goes outside the limits"; return v[0]; }
```

```
return v[i];
}


TArray TArray::operator+(const TArray& two) { TArray sum=*this;
    if (size != two.size)cerr<<"\nSizes don't coincide"; else for(int i=0;i<size;i++)
    sum[i]+=two.getV(i); return sum;
}

TArray& TArray::operator=(const TArray& a) {
if(&a == this)return *this;
delete[] v;
v=new int[size=a.size];
for(int i=0;i<size;i++)
    v[i]=a.v[i]; return *this;
}

// inheritor class with the limits of the index // change low .. high-1
class TRangeArray : public TArray
{
    int low, high; public:
    TRangeArray (int,int);
// two copy constructors
    TRangeArray (const TRangeArray &); TRangeArray (const TArray& a):TArray(a) {
                        low=0;
                        high=getSize();
                        }
/* TRangeArray& operator=(const TRangeArray &); -possible not to write */

            int& getV(int i)const
                        {
                        return TArray::getV(i-low);
                        }
    int& operator[](int i) {
                        return TArray::operator[](i-low);
                        }
```

```
/* can work for the class TArray too
at the expense of the copies' generator TRangeArray (TArray&) */

friend ostream& operator<<(ostream& os, TRangeArray a);
    /* operator+ will work for the class TRangeArray at the expense of the copies' generator
    TArray(TRangeArray &)and is created automatically */
};

TRangeArray ::TRangeArray (int left, int right): TArray(right-left)
{
    low = left; high = right; if (high < low) high = low;

}

TRangeArray ::TRangeArray (const TRangeArray & a): TArray(a.high-a.low)

{
            low=a.low;
    high=a.high; for(int i=low;i<high;i++) //impossible to address vector v directly (*this)
[i]=a.getV(i);
}

ostream& operator<<(ostream& os, TRangeArray a) {
for(int i=a.low;i<a.high;i++)s<<a[i]<<" "; return os;
}


int main(int argc, _TCHAR* argv[])
{
TArray a(3),b(3),c(3);
TRangeArray d(5,8),e(0,3),s(2,5) ;
for (int i=0;i<3;i++)
{
a[i]=i+1; // index operator from TArray class b[i]=i+4; // index operator from TArray class
//index operator from TRangeArray class
d[5+i]=(i+1)*2;
```

```
//called index operator from TRangeArray class e[i]=i*3;
}

TRangeArray f(a); // TRangeArray (TArray&)
//called the operator << from TRangeArray class cout<<"\n f(a): "<<f;

//called TRangeArray(TArray&),the operator << // from class TRangeArray
cout<<"\n a: "<<a;

//called TRangeArray(TArray&),
//the operator << from TRangeArray class cout<<"\n b: "<<b; cout<<"\n d[5-8]: "<<d;
// called the operator << from TrangeArray class cout<<"\n e[0-3]: "<<e;

// called TRangeArray(TRangeArray &)
TRangeArray j(d);
// called the operator << from TRangeArray class cout<<"\n j(d): "<<j;

/* called operator operator+(a,b), TRangeArray (TArray&), operator= from TrangeArray class
*/

s=c=a+b;

// called the operator << from TRangeArray class cout<<"\n (a+b)[2-5]: "<<s;

TRangeArray g(s); // called TRangeArray(TArray&)
//called the operator << from TRangeArray class cout<<"\n g()[0-3]: "<<g;

/* copies' generator TArray(TRangeArray &) not written by us, it works automatically */

TRangeArray k(a);
cout<<"\n k(a): "<<k;

d=a;
cout<<"\n d=a d[5-8]: "<<d;
/* there are no operators operator+(TRangeArray, TArray) and operator+
(TRangeArray,TRangeArray) (not written), but at the expense of copies' generator the next
code works: */
```

```
d=d+a;
cout<<"\n d=a[3]+d[5-8]: "<<d; a=e+d;
cout<<"\n a=e[0-3]+d[5-8]: "<<a; getch();
return 0;
}
}
```

Result of the program work:

```
a: 1 2 3
b: 4 5 6
d[5-8]: 2 4 6
e[0-3]: 0 3 6
f(a): 1 2 3
j(d): 2 4 6
(a+b)[2-5]: 5 7 9
g()[0-3]: 5 7 9
k(a): 1 2 3
d=a d[5-8]: 1 2 3
d=a[3]+d[5-8]: 2 4 6
a=e[0-3]+d[5-8]: 2 7 12
```

Assignment №6. Abstract classes

Tasks

To build the hierarchy of classes [11]. The hierarchy root – the abstract class with the virtual functions, indicated in the program. To create the derivative real classes in which to override all purely predecessor functions. To write the input/output functions. To define by oneself, which data-members are necessary, which of them must be defined in the base class, and which – in the derivative one. To define by oneself any polymorphic function in order to manipulate by the objects of different types in the created hierarchy.

Variants of the tasks

1. To create the abstract base class **TPair** with the virtual arithmetic operations +=, -=, *=. To create the derivative classes **TComplex** (complex number) and **TRational** (rational fraction) with the own realization of arithmetic operations.
2. To create the base class – geometric figure with the assigned base (the radius circle **r**) and height **h** and vertical function-members of the area calculation of the figure and volume surface. To create the derivative classes – **TCylinder, TCone, TTruncatedCone** (cylinder, cone, conoid) with the own realization of virtual functions.

3. To create the base class **TFigure** with virtual function-members calculate square shape and length of the contour. To create the derivative classes: **TRectangle** (rectangle), **TCircle** (circle), **TTrapezium** (trapeze) with its functions to calculate the square shape and length of the contour.
4. To create the base class - an employee with a given rate and payroll function. To create the derivative classes - employee hourly rate officer in the state fixed-rate and interest rate of employee self-realization payroll function by entering the number of eight-hour days.
5. To create the base class **TBody** with virtual functions of calculating surface area and volume. To create the derivative classes **TParalelepiped** (a box) and **TPyramid** (regular **n**-angled pyramid with its functions of surface area and volume.) 6. To create the base class **TCurrency** to work with sums

of money. Define virtual functions to transfer the amount in UAH and output to the screen. To create the derivative classes **TDollar** (USD) and **TEuro** (EUR) with its functions in UAH transfer and printing.

7. To create the base class **TTriangle** to describe a triangle with virtual functions of calculating the area and perimeter. As the parameters of the class to use two sides of the triangle and the angle between them. To create the derivative classes - right-angled triangle, isosceles triangle and an equilateral triangle with their functions of calculating the area and perimeter.

8. To create the abstract base class **TRoot** with the virtual calculation functions of the roots and pint results. To define the derivative class **TLinear** (linear equation) and **TSquare** (square equation) with the own calculation functions of the roots and print equations.

9. To create the abstract base class **TFunction** with the virtual function-members of the calculation of the function value in the assigned point **x** and the result output on the screen. To define the derivative classes **TEllipse** and **THyperbola** with the own calculation functions and the print of the curve equations.

10. To create the abstract base class **TTriad** with the virtual increase functions for one point of each of the three data-members and the value print. To create the derivative classes **TTime** (current time) and **TDate** (current date) with own realization of the above-mentioned functions.

11. To create the abstract base class **TSorting** with ID sequence, virtual member functions, sorting, obtaining the amount of elements and their output on the screen. To create the derivative classes **TChoice** (method of choice) and **TQuick** (quick sort).

12. To create the abstract base class **TList -** list with the virtual functions of addition **push**() into the list and delete from the list **pop**(). To realize on the base of the list stack and the turn with the own realization of the above-mentioned functions.

13. To create the abstract base class **TInteger** (the integer, represented by the array of its numbers) with the virtual arithmetic operations **+=**, **-=**, ***=** and the function of the output on the screen. To create the derivative classes **THeximal** (the integer number in the hexadecimal form), **TBinary** (the integer number in the octal form) and to override for them the above-mentioned functions.

14. To create the abstract class **TLine** (line segment). On the base of the class **TLine** to create the class **TColoredLine,** the classes **TPolyLine** and **TColoredPolyLine** (right polygons with the assigned number of the vertexes). The classes must have virtual methods of the coordinates extract set and obtaining the values of all figures coordinates as well as the color change and the running color.

15. To create the abstract base class **TSeries** (progression) with virtual functions of calculating the progression of the specified member and its amount. To define derived classes **TLinear** (arithmetic progression) and **TExponential** (geometric progression) with their own implementation of the above functions.

16. To create the abstract base class **TNorm** with the virtual functions of the normalization and calculation of the vector's length. To define the derivative classes **TVector2D** (vector on the plane) and **TVector3D** (vector in space) with the own realization of the above-mentioned functions. For **TVector2D** to calculate the norm as the root from the sum of the squares, for **TVector3D –** as maximum from the modules of the vector components.

17. To create the abstract base class **TContainer** with virtual member functions, **sort**() and **norm**(). To create derived classes **TBubble** and **TChoice.** For **TBubble** method **sort**() must implement bubble sort, and **norm**() - to calculate the square root of the sum of the elements. For **TChoice** method **sort**() must override the sort by means of the choice of maximal element and **norm**() - calculate the arithmetic mean of all the elements.

18. To create the abstract base class **TArray** with virtual member-functions of finding the amount of unit processing elements of the array **foreach**(). To create derived classes **TUnionArray** and **XorArray.** For **TUnionArray** finding the summation of elements is realized as a bitwise union, and **foreach**() - the inversion of array elements (changing the sign of each element to the opposite). For **XorArray** finding the summation of the items is realized as the bitwise exclusion **OR** for array elements, and **foreach**() performs the rationing of the array.

19. To create the abstract base class **TArray** (the array of different digits) with virtual member functions, finding the sum of elements and item-processing of the array **foreach**(). To create derived classes **TAndArray** and **TOrArray.** For **TAndArray** finding the summation of digits is realized as the intersection of sets, and **foreach**() - bubble sort method. For

TOrArray finding the summation of the digits is realized as a union of sets, and **foreach**() performs the quick sorting.

20. To create the abstract class **TFunction** (the curve, defined by the coefficients **a, b, c**) with virtual functions-members of calculating the function **y**(x) in the point x=**x0,** check, whether the assigned point **(x0, y0)** belongs to the **c**urve and the curve drawing. To create derived classes **TSquareFunction** (function $\mathbf{y = ax^2 + bx} + c$) and **TLinearFunction** (function **ax + by + c** =0) with their own implementations of virtual functions.

21. To create the abstract class **TMatrix** (fourth-order square matrix) with virtual elements – sorting functions **sort**() rows column matrix and **norm**() – finding matrix norm. To create derived classes **TRowMatrix** (sorting function rows **sort**() should be realized with the growth of the minimal row element, the function **norm**() – using the matrix norm, defined by the square root of the sum of the squares of its elements) and **TColumnMatrix** (column sorting function **sort**() to realize with the growth of the maximal column element, the function **norm**() – using the matrix norm, defined by the maximal module of its elements).

22. To create the abstract class **TShape** with the data-members real number **r** and the array of the points. To create the derivative classes **TCircle** (with the functions, defining the number of the array points **count**(), which belong to the circle of radius **r** and the center at the coordinates beginning and **sort**() sorting array by the distance to the coordinates beginning) and **TSquare** (with the functions **count**(), defining the number of the array points, belonging to the square with the side **r** with the center at the beginning and the sides, parallel to the axis coordinates and **sort**() – sorting the array points by the turn angle, relating to the added direction of the abscissa axis) with the own realization of virtual functions.

23. To create the abstract class **TMatrix** (fifth the other square matrix) with virtual members-functions **count**() - calculating the number of rows/columns of the matrix and **remake**() - the transformation matrix. To create derived classes **TRowMatrix** (function **count**() returns the number of rows, containing the increasing sequence of elements, the function **remake**() assigns zero to all the negative elements of the matrix) and **TColumnMatrix (**function **count**() returns the number of columns,

containing only even elements, the function **remake**() changes the signs of matrix elements to the contrary) with their own realizations of virtual functions.

24. To create the abstract base class **Row** (character string) with virtual arithmetic operation += and function **change**(). To create derived classes **TNameRow** and **TColorRow** with their own implementations of virtual operations and functions. In the class **TNameRow** add the data-member, which is the name of the row (operation += creates the string concatenation with the name of the first line, and the function **change**() changes the name, given to the line). In the class **TColorRow** add the data-member of the color line (operation += finds the concatenation of inverted lines with the color of the first line, and the function **change**() - changes the color of the line).

25. To create the abstract base class **TFunction** with virtual functions **equation**() and **draw**(), the class-descendant **TPolinom** (polynomial of degree **n** with coefficients, assigned in the array) and its derivative classes **TDerivative** and **TAntiderivative** (derivative and antiderivative) with their own implementations of virtual functions. In the classes **TFunction, TDerivative** and **TAntiderivative** the function **equation**() correspondingly tabulates polynomial, its derivative and antiderivative, on the assiged uniform grid. The function **draw**() in each class draws the graph of the corresponding object on this grid.

Examples of the code

Example creating the abstract base class with two derivative real classes *Condition of the task*: To create the abstract base class **TNumber** (the number, written in the string of symbols) with the pure virtual arithmetic operation "+=" and the function **sum** .

To create the derivative classes **TIntNumber** and **TCharNumber** with the own realizations of virtual operations and functions. In the class **TIntNumber** operation "+=" finds the arithmetic sum of two numbers, and the function **sum** – the sum of digits of one number.

In the class **TCharNumber** the operation "+=" finds the concatenation of two number-strings, and the function **sum** – the quantity of digits of one number string.

```
#include <vcl.h>
#pragma hdrstop
```

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <iostream.h>


class TNumber {
protected :
char* a;
public:
TNumber(char* A){a=strdup(A);}
virtual TNumber& operator+=(TNumber&)=0 ;
virtual void print()=0;
virtual int sum()=0;
virtual ~TNumber(){delete[] a;}
};

class TIntNumber :public TNumber
{
public:
TIntNumber(char* A):TNumber(A) {}
TIntNumber(TNumber& ob):TNumber(ob){};
TNumber& operator+=(TNumber&) ;
virtual void print();
virtual int sum();
~TIntNumber(){}
};

int TIntNumber::sum()
{
int i=0,s=0;
while(i<strlen(a))s+=a[i++]-'0';
return s;
}

TNumber& TIntNumber::operator+=(TNumber& ob)
{
```

```
TIntNumber* p=(TIntNumber*)&ob;
int s=atoi(a)+atoi(p->a); // atoi(ob.a)- doesn't work
itoa(s,a,10);
return *this;
}

void TIntNumber::print()
{cout<<"\nTIntNumber:"<<a;}

class TCharNumber: public TNumber
{
public:
TCharNumber (char* A):(A) {}
virtual TNumber& operator+=(TNumber&) ;
virtual void print();
int sum(){return strlen(a);}
~TCharNumber(){}
};

TNumber& TCharNumber::operator+=(TNumber& ob)
{ TCharNumber* p=(TCharNumber*)&ob;
strcat(a,p->a);
return *this;
}
void TCharNumber::print()

{cout<<"\nTCharNumber:"<<a;}

void printAll( TNumber* mas[],int size)
{
for(int i=0;i<size;i++)
{
mas[i]->print();
cout<<" sum="<<mas[i]->sum();
}
}
```

```
int main()
{
TIntNumber a("12"),b("345"),c("35");
TCharNumber A("12"),B("345"),C("35");
TNumber* mas[10]={&a,&b,&c,&A,&B,&C};

b+=a;
b.print();

C+=A;
C.print();


B+=a;
B.print();

(a+=A).print();

mas[6]=new TIntNumber("1333");
mas[7]=new TCharNumber("1555");

printAll(mas,8);

delete mas[6];
delete mas[7];
getch();
return 0;
}
```

Result of the program work:

```
TIntNumber:357
TCharNumber:3512
TCharNumber:34512
TIntNumber:24
TIntNumber:24 sum=6
TIntNumber:357 sum=15
TIntNumber:35 sum=8
TCharNumber:12 sum=2
```

TCharNumber:34512 sum=5
TCharNumber:3512 sum=4
TIntNumber:1333 sum=10
TCharNumber:1555 sum=4

LIST OF LITERATURE

1. Айра П. Объектно-ориентированное программирование на C++, 2-е издание / П. Айра. – М. : Бином, 2001. – 462 с.

3. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / Г. Буч. – М. : Бином, 2001. – 560 с.

4. Грегори К. Использование Visual C++ 6. Специальное издание / К. Грегори. – М.; СПб. К. : Вильямс, 1999. – 864 с.

5. Дерк Л. С и C++ [справочник] / Л. Дерк. – М. : Бином, 1997. – 588 с.

6. Джефф Э. C++ : библиотека программиста / Э. Джефф. – М.; СПб : Питер, 2002. – 320 с.

7. Дьюхарст С. Программирование на C++ / С. Дьюхарст, К. Старк. – К. : ДиаСофт, 1993. – 271 с.

8. Вандевурд Д. Шаблоны C++: справочник разработчика = C++ Templates: The Complete Guide / Д. Вандевурд, М. Николай. – М. : Вильямс, 2003. – 544 с.

9. Лаптев В. В. Объектно-ориентированое программирование. Задачи и упражнения / В. В. Лаптев, А. В. Морозов, А. В. Бокова. – СПб. : Питер, 2007. – 288 с.

10. Мейерс С. Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению наших программ и проектов / С. Мейерс. – М. : ДМК Пресс, 2000. – 304 с.

11. Павловская Т. А. Програмирование на языке высокого уровня / Т.А. Павловская. – СПб. : Питер, 2003. – 461 с.

12. Павловская Т. А. C++. Объектно-ориентированное программирование : практикум / Т. А. Павловская, Ю. А. Щупак. – СПб. : Питер, 2005. – 265 с.

13. Рассохин Д. От Си к Си++ / Д. Рассохин. – М. : Эдель, 1993. – 128 с.

14. Ритчи Д. Язык программирования СИ / Д. Ритчи, Б. Керниган. – М. : Финансы и статистика, 1992. – 294 с.

15. Саттер Г. Решение сложных задач на C++. Серия C++ In-Depth М.; СПб. / Г. Саттер. – К. : Вильямс, 2002. – 400 с.

16. Сопронюк Т. М. Технології візуального й узагальненого програмування в C++Builder : навчальний посібник / Т. М. Сопронюк. – Чернівці : ЧНУ, 2009. – 80 с.

17. Сопронюк Т. М. Об’єкто-зорієнтоване програмування на C++ : Методичні рекомендації та завдання для лабораторних робіт / Т. М. Сопронюк, С. М. Тимку. – Чернівці : ЧНУ, 2004. – 51 с.

18. Страуструп Б. Дизайн и эволюция C++/ Б. Страуструп. пер. с англ. – М. : ДМК Пресс; СПб. : Питер, 2006. – 448 с.

19. Страуструп Б. Язык программирования C++ / Б. Страуструп. – К. : ДиаСофт, 1993. – 256 с.

20. Уолтер С. С++. Курс объектно-ориентированного программирования / С. Уолтер. – М.; СПб. ; К. : Вильямс, 2001. – 704 с.
21. Фейсон Т. Объектно-ориентированное программирование на Borland C++ 4.5. / Т. Фейсон. – К. : Диалектика, 1996. – 544 с.
22. Object-oriented programming in C++: Textbook / Authored by Tatyana Sopronyuk, Translated by Nonna Shulga: CreateSpace, 2014. – 130 p.
23. Об'єктно-орієнтоване програмування на С++ [Текст] : навч. посіб. / Т. М. Сопронюк ; Чернів. нац. ун-т ім. Юрія Федьковича. - Чернівці : Рута, 2014. - 175 с.
24. http://tanet.tneu.org/phocadownloadpap/osnovy_progrsmuvannja_C_C++.pdf
25. http://bookwebmaster.narod.ru/cplusplus.html
26. http://www.intuit.ru/department/pl/ccpp/print.lit.html
27. http://bulletinsite.net/index.php?id1=6&category=programmer&author=pavlovskaya-ta&book=2003
28. http://khpi-iip.mipk.kharkiv.edu/library/pgm/cpp/index.html
29. http://articles.org.ru/docum/builder/3.php
30. http://www.codenet.ru/progr/cpp/ipn.php