OBJECT-ORIENTED PROGRAMMING IN C++

TATYANA SOPRONYUK

Originally published in Ukraine, Chernivtsy, 2014

Translated by Nonna Shulga

ISBN-10: 1502520907
ISBN-13: 978-1502520906

The textbook is intended to be used in the computer science course of "Object-oriented programming in C++". It contains all the information needed for completing assignments and labs.

This textbook covers fundamental concepts in object-oriented programming: objects, classes, and inheritance. This book shows 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. Detailed examples with comments are provided.

This textbook is intended for educational purposes.

Object-oriented programming in C++: Textbook / Authored by Tatyana Sopronyuk, Translated by Nonna Shulga: CreateSpace, 2014. – 131 p.

ISBN-13: 978-1502520906 (CreateSpace-Assigned)

ISBN-10: 1502520907

TABLE OF CONTENTS

FOREWORD **6**

1. ELEMENTS OF OBJECT-ORIENTED PROGRAMMING **9**

1.1. OBJECT. CLASS. STRUCTURE 9

1.2. STANDARD OBJECTS OF THE STREAM INPUT/OUTPUT 13

1.3. OVERLOADING OPERATIONS FOR THE TYPES STRUCT, UNION AND ENUM 15

1.4. OVERLOADING OF FUNCTIONS 21

1.5. EXAMPLE OF OPERATORS' OVERLOADING FOR TWO STRUCTURAL TYPES 22

1.6. TEMPLATES OF FUNCTIONS 25

1.7. EXAMPLE OF OVERLOADING FUNCTIONS AND USAGE OF FUNCTION TEMPLATES 26

QUESTIONS FOR SELF-CONTROL 29

TASKS FOR INDEPENDENT WORK 30

2. WORK WITH THE OBJECTS WITHIN ONE CLASS **31**

2.1. OPERATING ACCESS TO THE MEMBERS OF THE CLASS 31

2.2. CONSTRUCTORS 34

2.3. STATIC MEMBERS OF THE CLASS 39

2.4. DESTRUCTORS 41

2.5. DYNAMIC CREATION AND DELETING OF OBJECTS 43

2.6. FRIENDS OF THE CLASS 47

2.7. OVERLOADING OPERATIONS FOR CLASSES 50

2.8. BINARY AND UNARY OPERATIONS 53

2.9. ASSIGNMENT OPERATOR 54

2.10. OPERATORS OF THE CALL, INDEXING AND TYPE CONVERSION 56

2.11. OVERLOADING OF THE OPERATOR NEW AND DELETE 58

2.12. EXAMPLE OF THE ELABORATION OF THE CLASS "SET OF THE INTEGER NUMBERS" 61

QUESTIONS FOR SELF-CONTROL 71

TASKS FOR THE INDEPENDENT WORK 71

3. CREATION OF NEW CLASSES USING THE EXISTING ONES.....72

3.1. BUILDING CLASSES WITH MEMBER-OBJECTS.....72
3.2. THE TEMPLATES OF THE CLASSES.....74
3.3. INHERITANCE.....77
3.4. HIERARCHY OF CLASSES INHERITANCE UNARY INHERITANCE.....79
3.5. TRANSFORMATION OF THE POINTERS ON THE OBJECTS FROM ONE HIERARCHY OF CLASSES.....83
3.6. TRANSFORMATION OF THE TYPES FOR THE OBJECTS FROM ONE HIERARCHY OF CLASSES.....84
3.7. MULTIPLE INHERITANCE.....90
QUESTIONS FOR SELF CONTROL.....95
TASKS FOR INDEPENDENT WORK.....96

4. ELABORATION OF FAMILY CLASSES. ABSTRACTION.....97

4.1. STATIC AND DYNAMIC POLYMORPHISM.....97
4.2. VIRTUAL FUNCTIONS.....97
4.3. ABSTRACT CLASSES.....98
4.4. PROJECTING AND BUILDING THE CLASS LIBRARY WITH VIRTUAL FUNCTIONS.....102
4.5. POLYMORPHIC FUNCTIONS.....107
4.6. VIRTUAL INHERITANCE.....109
QUESTIONS FOR SELF-CONTROL.....112
TASKS FOR THE INDEPENDENT WORK.....113

5. ASSIGNMENTS.....114

ASSIGNMENT № 1.....114

Theme: Structures. Outer functions and member-functions of the structure.....114

ASSIGNMENT № 2.....116

Theme: Overloading of operations and functions. Functions templates..116

ASSIGNMENT № 3 ..... **118**

Theme: Structures and classes *118*

ASSIGNMENT № 4 ..... **121**

Theme: Creation of classes *121*

ASSIGNMENT № 5 ..... **125**

Theme: Creation of classes, implementing the existing classes. Inheritance. Aggregation *125*

ASSIGNMENT № 6 ..... **127**

Theme: Abstract classes *127*

LIST OF LITERATURE **130**

FOREWORD

Object-oriented programming or OOP is methodology of programming, 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 conditions of OOP, especially C++, are based on three main ideas: encapsulation, polymorphism and inheritance [13, 18, 19, 22-26, 28].

Encapsulation is a mechanism linking in one construction the data and the code, which manipulates with these data. Encapsulation also defends the part of the functions and data against the outer intrusion or wrong usage. In object-oriented programming, functions and the data with which they operate, can be linked into one whole. Such a union creates a class type.

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

Thus, *class* is a capsulated abstraction with the strictly defined rules of the access to the separate sections.

Object is a variable of the class type (instance of the class). Object supports encapsulation.

Within the object both codes and data can be *private*. Private codes or data are accessible only for other parts of these objects. Thus, private codes and data are not accessible for those functions existing outside the object. If codes and data are open, then even if they exist within the object, they are accessible for other parts of the program. As a rule, the open part of the program is used in order to insure the controlled access to the private elements of the object.

The object unites the codes and the data and is a variable of the class type, defined by the user.

Polymorphism (from the Greek polymorphos) is a peculiarity which allows using the same words to denote several tasks, identical by the content but different as to their realization.

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. The types of the data, used as arguments at the calling of *functions*, denote how the concrete version of the function is really called. Thus, in C++ it is possible to use one name of the function for the realization of different

actions. Such a possibility is called *function overloading*. The overloading of functions is one of the forms of polymorphism

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 advantage of polymorphism is the fact that it allows using one interface for the task of the only type of actions and makes it impossible lowering the complexity of programs. The choice of the concrete action, depending on the situation, is exerted by the compiler. It is enough to use only the general interface.

Polymorphism is also used for the operators. In fact polymorphism in the language C is used limited, in particular, in arithmetic operators. For example, the symbol "+" is used for the adding of integer, long integer and real numbers. In this case the compiler automatically defines which type of arithmetic's is necessary.

It is also possible in the language C++ to use this conception for other types of the data, put forward by the user. So, the symbol "+" may be used for the exertion of the operation by adding complex numbers. For this it is necessary to define the corresponding operator's function. This type of polymorphism is called *overloading of operators*.

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.

Inheritance allows supporting a concept of the *hierarchical classification*. Without its usage for every object it would have been necessary to introduce all characteristics which would have defined it completely. By using inheritance it is possible to describe the object, indicating the *general* class (the *family base or ancestor* class) to which it belongs and those specific features that makes the object unique. The generated class inherits all the features, linked with the parent base and adds to them its own specifics characteristics. Inheritance plays the determining role in OOP.

The textbook supplies the students with all the information needed in the course "Object-oriented programming".

The textbook consists of 5 parts. At first we consider the possibilities of C++, which did not exist in the language C and which don't refer directly to the notion of classes, in particular, stream input/output, operators of dynamic allocation of memory and setting it free, overloading of functions, template functions. Further, we passed

over from the notion of structure in the language C to the notion of the class. We described the operations of the construction of the class and its elimination, static members of the class, friendly functions, interface and realization of the class, overloading of operators for the objects of the given class, templates of classes and friendly ones.

We have also considered means of using created classes (aggregation and inheritance) for the extending possibilities of existing classes and the notion of the base and derived classes, multiple inheritance, building of the hierarchy of classes, usage of virtual functions, abstract classes, polymorphic functions, virtual inheritance.

In every part of the textbook we have given examples of programs with comments, demonstrating suggested theoretical material. Every part ends with the questions for self-control and tasks for independent work with the demonstrated programs.

For the most part the textbook contains the console programs, checked in the field of the development of **`Microsoft Visual C++.Net`** and/or **`Borland C++Builder`** [4, 9, 16, 27].

Work with the libraries **STL** [8, 9, 11, 12] and **VCL** is not considered in the given textbook. These instruments OOP in C++ and some other aspects, not mentioned in the textbook, have been described by the author in [16].

At the end of the textbook we have introduced the mandatory assignments for the course OOP in C++. Some assignments (laboratory works № 1, № 3 and № 6) are based on the ideas from [11, 25]. Other assignments belong to the author, [17], they were tested and improved by the department.

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

The author is grateful for many-year cooperation while conducting classes to Yakymov Ihor, Dudnitskiy Pavel, Danilyuk Ivan.

1. Elements of object-oriented programming

1.1. Object. Class. Structure

Object in the general understanding denotes arbitrary real or imaginary reality, that is, abstract essence endowed with the characteristics of the real world.

From the point of view of programming the creation of the objects and manipulating with them is a result of the methodology of programming which is implemented into the code construction of the description of objects and operations with them. Every object of the program, as well as any other real object, differs with its own attributes and characteristic behavior. So, any class denotes some category of objects, and any object is the instance of some class.

Class generalizes the notion of the data type. Object is variable of the class type. The goal of introducing classes in C++ is to enable the programmer to create new data types, both convenient and ordinary. Thus, class is the type, defined by the user. Every class has the data and the choice of methods which manipulates with this data. In the same way as the instance of the standard type, such as **int,** is called variable, the instance of the type, chosen by the user (that is class), is called the object. The center of attention in C++ is not methods but objects. New types are necessary for the definition of the conception which is not possible to define by standard types.

Class has no physical essence, its closest analogy is a structure. The memory is pointed out only in the case if the class is used for the creation of object. This process is also called *class instance*.

Any object of the language C++ has the same attributes and procedures as the other objects of the same class.

Working in some environment the programmer obtains the access to the libraries of classes (for example for the library of visual components of **C++Builder** [16] or to the standard library of the templates **STL** [8,10]).

As a rule the object finds itself in some unique state which is described by the stream value of its attributes. The functioning of the object class is defined by the possible actions with this class instance.

The definition of the class in the language C++ contains the encapsulation of the members of the data and methods which operate with the *data members* of the object and define the behavior of the object. In OOP terminology an object's data are called *data members* of the object

and codes – *member functions*.

Method is a function into which object, i.e. class instance, is conveyed additionally by implicit parameter. Method describes the action with this very object for which it is called. Just this hidden link of the method with the class to which it belongs isolates it from the notion of the ordinary function. During its exertion the method has an access to all the data of its class, though it doesn't demand to convey the object of this class as a parameter. The object is conveyed to the method by means of a hidden parameter– the pointer 'this" to the class instance. At any address of the method to the members of the data class the compiler generates a special code which the pointer "**this**" uses.

Encapsulation allows isolating the object from the outer medium. It enhances the reliability of the developed programs, since the localized in the object functions exchange with the program comparatively small portions of the data. And the number and the type of this data are constantly checked. As a result the change or modification of the functions and data, encapsulated into the object doesn't, as a rule, cause any consequence for the program as a whole. And this is difficult to check. Moreover, the global variables in OOP are almost not used. It is done with the aim of enhancing the program defense.

Classes in C++ are the natural continuation of the structures in C. That is why before considering the specifics of the development of classes in C ++ we shall build the type, defined by the user, created by using *struct*.

Example 1.1.1:

```
// 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)
{  //adding 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>
 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;
  //adding one day to до 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);
  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, 26].

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 1.1.2:

```
// 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;
  }
```

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 – "::".

If functions are member-structures, then the object, for which these functions are called, is not already conveyed as a parameter. Therefore bodies of these functions in the example 1.1.2 are simpler, than in the example 1.1.1.

Functions, pointed out inside the structure (the class) are called function-members and one can call them out only for changing the variable of the created class, using the standard syntax of the access to the members of the structure. In order to obtain the elements of the structure (or the class) one can use the operation point (.) and the operation arrow (–>). The operation point addresses the element of the structure (or class) by the name of the variable (of the object). The operation arrow ensures the access to the element of the structure (or class) through the pointer to the object.

Propose the example of the main program by using the created *struct TDate* from the example 1.1.2.

Example 1.1.3:

```
#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;
  //adding 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;

    }
```

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*.

1.2. Standard objects of the stream input/output

Note that in the examples 1.1.1 and 1.1.2 the operations input/output are organized differently. In the example 1.1.1 we use functions from the library of the language C **`stdio`**. To be able to work with them one has to include the head title file **`<stdio.h>`**.

In the language C++ there is a library of the stream input/output **`iostream`**. In order to work with it we need to connect the file of the title **`<iostream.h>`**.

Two important classes from the library **`iostream`** [13]:

- **`istream`** – input from the standard device input, contains the overloaded operator input **`>>`** and the functions for the

noninformative input;

- **ostream** – output on the standard device output, contains the overloaded operator input **<<** and the functions for the not informative output.

Standard objects of the stream input/output C++ are identical by their purpose to the standard files of input/output of the language C:

- **cin** – is a standard input (by the suppression keyboard), analogue of the file of standard input **stdin**;
- **cout** – is a standard output (by the suppression display), analogue of the file of the standard output **stdout**;
- **cerr** – is a standard not buferized output of information about errors (a screen by default), analogue of the file of the standard output of the information about errors **stderr**;
- **clog** – is a standard buferized output of information about errors (a screen by default), analogue of the file of the standard output of the information about errors **stderr**.

Thus, as it is evident, the streams of the output are possible to classify by the purpose, redirect to the different devices or make the program more visual. For example, if the output is pointed into the stream **cerr**, then it immediately signals about the error in the program.

In the output data it is possible to guide the formatting of the information using manipulators. The part of manipulators is defined in the library **iostream**, and the part – in **iomanip**.

Here is the content of some manipulators:

- **endl** – transition to the next line;
- **dec** – output of the integer in the decimal form;
- **oct** – output of the integer number in the octal form;
- **hex** – output of the integer number in the hexadecimal form;
- **setprecision**(**3**) – output of the real number rounded to three decimal places.

As is evident from the example 1.1.3 the stream of the output is possible to continue and redirect into it the variables of different standard types.

In the same way it is possible to act with the stream of the output.

For example, the data in the next fragment will be analyzed and put down into the variables according to their type.

```
int i, j;
double a;
char s[80];
cin>>i>>a>>j>>s;
```

Note that the first word will be put down only to the first space symbol. In order to put down the whole line it is necessary to write down such a code: `

```
cin.get(s, 80);
```

The overloaded operator **<<** (**>>**) must return reference to the standard class of the stream output **ostream** (of the stream input **istream**) which is defined in the file **iostream.h** (in the language C operators **<<** and **>>** are used for shift).

Besides, the operational function **operator<<** (**operator>>**) must have two arguments:

- the firstst parameter – reference to **ostream** (**istream**);
- the second parameter for the **operator<<** – the value of the variable class type which is necessary to output or the reference to this variable;
- the second parameter for the **operator>>** – reference to the variable of the class type, necessary to input.

1.3. Overloading operations for the types struct, union and enum

Let's consider the fragment of the main program from the example 1.1.3.

```
TDate tomorrow=today; //adding one day to tomorrow
tomorrow.addDay(1); //print of variable tomorrow

cout<<tomorrow.y <<" "<< tomorrow.m <<" "<<tomorrow.d;
```

The applied programmer, using the structure **TDate**, would find the fragment of the code, given below, more convenient and compact:

```
TDate tomorrow=today+1;
cout << tomorrow;
```

In order to give an applied programmer such a possibility one has to use the mechanism of overloading the operations, possible to be realized in C++ for variables of class type (**enum**, **union**, **struct**, **class**).

Operational functions have a reserved name **operator@**, where **@** –the overloaded operation. Two kinds of operations are distinguished – unary (one argument) and binary (two arguments).

There are rules of operations' overloading. We shall consider them later while learning classes. And now we'll give an example of overloading operators by adding the number of days "**+**" and the stream output "**<<**" for the structure type **TDate**.

Example 1.3.1:

```
#include <iostream.h>

struct TDate
        {
        int d,m,y;
        };

TDate operator+(TDate date, int dd)
{
    TDate rez=date;
    rez.d+=dd;
    return rez;
}

ostream& operator<<(ostream& os,  TDate date)
{
   os<<"\n year - "<<date.y<<endl<<
    "\n month - "<<date.m<<endl<<
    "\n day - "<<date.d<<endl;
    return os;

 int main()
 {
  // creation of structure instances TDate

 TDate today ={5,9,2013};
 TDate tomorrow=today+1; // tomorrow=operator+(today,1);
  cout << tomorrow;      // operator<<(cout,tomorrow);
  cin.get();
  return 0;
 }
```

The result of the work of the program:

```
Year- 2013
Month- 9
Day- 6
```

Remark that while adding the number of days, the correctness control has not been made, therefore some cases, when the number of days is more than 31, are quite possible.

One can't overload the operations for nonclass types. Hence it follows that at least one of the operands of operator functions has to be of class type.

Further we'll show the application of the operator overloading of the stream output **operator<<** for the **union** type.

In the program below there are two class types – bit field and union. Let us impose the input symbol on the bit field (using only one field of memory). For every input symbol we shall output every bit of its bit image.

Example 1.3.2:

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

// bit field with the named bit
struct Tbyte
  {
  unsigned a:1, b:1, c:1, d:1, e:1, f:1, g:1, h:1;
  };

// union
//(the symbol is imposed on its bit image)

union Tsymbol
  {
  unsigned char ch;
  struct TByte bit;
  };

// overloading the operation of steam output
// for union
ostream operator<<(ostream& os, TSymbol p)
{
return os<<p.bit.h<<p.bit.g<<p.bit.f<<p.bit.e
        <<p.bit.d<<p.bit.c<<p.bit.b<<p.bit.a<<endl;
}

void main()
{
TSymbol symbol;

// while ESC is not pressed
while (symbol.ch != 27)
          {
              // to read symbol
              symbol.ch=getche();
              putch(':');

              // to output its bit image
```

```
            cout<<symbol;
         }
return;
}
```

Results of the program work:

```
g:01100111
f:01100110
k:11011010
```

In the following example we shall show the overloading of operators increment for the type enumeration **enum**. The interface of these operations for a type **T** must look like this:

```
T& operator++(T&);       // prefix increment
T operator++(int, T&);   // postfix increment
T operator++(T&, int);   // postfix increment
```

Remark that the Interface of the postfix increment depends on the version of the compiler. Therefore we shall introduce two of its possible forms.

By the overloading of the operations **++** and **--** for a type **T** it is sometimes necessary to define their prefix and postfix forms. It is evident that the name of the operator's function, e. g. **operator++**, is identical for both prefix and postfix forms. For the compiler to be able to distinguish between these two forms one additional parameter of the whole type is used in the postfix form. It is not used in any other way but only denotes the necessary choice of the operator's function.

Example 1.3.3:

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

// the type of enumeration (file attributes)
enum  Tmode
  {
  mRead, mEdit, mWrite, mCreate
  };

// overloading of the stream output operation

// for the overloading type enum
ostream& operator<<(ostream& os, TMode mode)
{
os<< "Output of the regime of work\n";

switch (mode)
```

```
{
         case mRead:
              os<<"Reading regime - 0 \n";
                 break;
         case mEdit:
              os<<"Editing regime - 1 \n";
                 break;
         case mWrite:
              os<<"Outputting regime - 2 \n";
                 break;
         case mCreate:
              os<<"Creation regime - 3 \n";
                 break;

}
return os;
}

//overloading of the stream operation
//output for the type enum
istream& operator>>(istream& is, TMode& mode)
{
  cout<< "Choose the work regime with the file \n"<<
              "Reading regime - 0\n"<<
              "Editing regime - 1\n"<<
              "Outputting regime - 2\n"<<
              "Creation regime - 3\n";
  int m;

  is>> m;
  mode=m%4;
return is;
}

// prefix increment
TMode& operator++(TMode& mode)
{
switch (mode)
{
         case mRead:
              mode=mEdit;
                 break;
         case mEdit:
              mode=mWrite;
                break;
         case mWrite:
              mode=mCreate;
                 break;
         case mCreate:
              mode=mRead;
                break;
}
```

```
return mode;
}

//postfix increment
TMode  operator++(TMode& mode, int)
 {
 TMode rez=mode;
switch (mode)
{
         case mRead:
              mode=mEdit;
                 break;
         case mEdit:
              mode=mWrite;
                 break;
         case mWrite:
              mode=mCreate;
                 break;

  case mCreate:
              mode=mRead;
}
return rez;
}

void  main()
{
TMode mode1,mode2;
cin>>mode1;
mode2=++mode1;
cout<<mode1;
mode2=mode1++;
cout<<mode2;
cout<<mode1;

cin.get();
return;
}
```

Result of the work program:

```
Choose the work regime with a file
Reading regime – o
Editing regime – 1
Outputting regime – 2
Creation regime – 3
0
Output of the work regime
Editing regime – 1
Output of the work regime
Editing regime – 1
Output of the work regime
```

```
Outputting regime – 2
```

1.4. Overloading of functions

As a rule, different functions are given different names. However, when functions solve conceptually identical problems for the objects of different types, it is more convenient to use the same name.

For example, the language C demands using three different functions in order to find the absolute value of a number, that is, **abs**, **labs** and **fabs**. These functions return the absolute value of integer, long integer and real numbers correspondingly. In the language C++ all these different functions can have one name, e. g. **abs**. The compiler will distinguish them by the type of the input argument.

Using one name for the operation, dealing with different types of the data, is called *overloading*. This idea is realized in different libraries C++, it is easy to broaden it to the functions, defined by the user.

For example, if it is needed to write create the function that prints the values of **TDate** type variables and the values of variables of the structure type array (size 10 type **float**), then these functions can have the same name, e.g. **abs**. The compiler will distinguish them by the type and number of input parameters.

Example 1.4.1:

```
#include <iostream.h>

struct TDate
        {
        int d,m,y;
        };

void print(TDate date)
{
cout<< "struct: ";
cout<<"\n year - "<<date.y<<endl<<
    "\n month - "<<date.m<<endl<<
    "\n day - "<<date.d<<endl;
}

void print(float* a)
{
cout<< "Array: ";
for(int i=0;i<10;i++)
    cout<<a[i]<<" ";
cout<<endl;
}

int main()
```

```
{
TDate today={5, 9, 2013};
float mas[10]={1.2, 44., 2.33, 6.9, -77.5};
print(today);
print(mas);
cin.get();
return 0;
}
```

Both functions have the same name and purpose, but their bodies are completely different. Overloaded functions (redefined for different types of input parameters) are meant first of all for the convenience of the calling. When the overloaded function **`print`** is called, the compiler defines which of the functions with name **`print`** to use. And that function is used which has the arguments, closest to the types. And if such a function is not found, then the message is sent.

The process of determining the necessary overloaded functions consists in finding the best suited types of formal and actual arguments. It is done by checking the criteria [11].

- precise correspondence of the types;
- correspondence, achieved by non-obvious transformation of the types **`char`** in **`int`**, **`int`** in **`unsigned int`**, **`bool`** in **`char`** etc.) and by standard transformation of the types (**`int`** in **`double`**, **`double`** in **`int`**);
- correspondence, achieved by the transformation of the pointers in arbitrary types in **`void*`**, pointers in inherited classes – into pointers in base ones.

The result of the overloading doesn't depend on the order of functions' definition. Types returned are not the criteria of the choice of overloaded functions. Functions, pronounced in different fields of visibility, are not overloaded.

1.5. Example of operators' overloading for two structural types

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

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

Example 1.5.1:

```
#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.

1.6. Templates of functions

If there is a necessity to work out several identical functions, different only by the type of input parameters, the possibility to create the **template** for some generalized types of parameters is foreseen in C++. With help of the template the compiler generalizes the functions with the necessary types of input parameters, and the type of parameters is defined at the corresponding call of the function. By the description of the template of the function the writing **<class T>** or **<typename T>** denotes the announcing the unknown type **T** but when a function with the obvious argument is called the compiler itself automatically will introduce the corresponding type.

Example 1.6.1:

```
#include <iostream.h>

// announcing the template of the function for the
// exchange of values between the first and the last
// elements of the array
template <typename T, int size> void firstSwapLast(T* x)
```

```
{
T temp = x[0];
x[0]= x[size-1];
x[size-1] = temp;
}

int main()
{
char* s="abcdefghkl";
float* mas={1.2,44.,2.33, 6.9, -77.5};
firstSwapLast<float,5>(mas);
cout<< "Transformed array: ";
for(int i=0;i<5;i++)
       cout<<mas[i]<<" ";
cout<<endl;

firstSwapLast<char,10>(s);
cout<<" Transformed line: "<<s;
cin.get();
return 0;
}
```

As seen from the example it is possible to use in the parameters of settling the template not only the generalized types but existing ones as well. Moreover, the parameters of settling the template may be several (e g. **<class T1**, **class T2**, **class T3>**). When used the template (concrete definitions of the template), they may be omitted (if in the angle brackets of the template all the types are introduced as generalized with the help of the key word **class** or **typename)**.

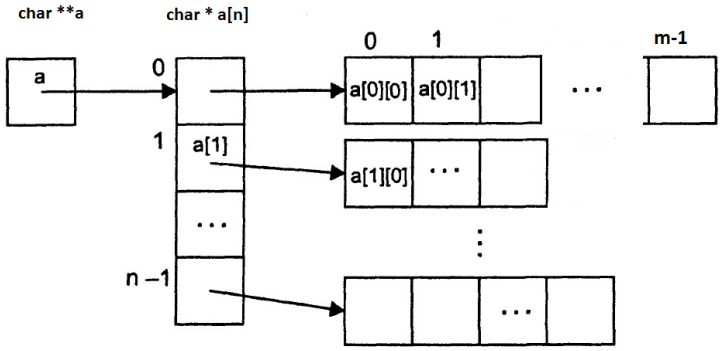
1.7. Example of overloading functions and usage of function templates

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.



Example 1.7.1:

```
#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          tSortOutput<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.

Questions for self-control

1. What is object? How can it be described?
2. By what is structure differed from the class?
3. Is it possible to overload the operations for the types “array of the integer numbers” and “array of real numbers”? And functions?
4. Is it possible to create the template for the sorting of the array of words and the array of numbers? And for the print of these arrays?

5. What is the template of the function?
6. What key words can be used for defining some generalized type in the template of the function?
7. How many parameters can there be in the template?
8. What is the specification of the template at the calling of the function?
9. When is it possible to omit the real values of parameters at the generalization of the concrete function?
10. Is it possible to define some operator functions with the same name for one structure?
11. What types of the data are considered class?
12. For what types of the data can the overloading of functions be used? And as to operators?
13. Is it possible to change the priority of the operations exertion at their overloading?
14. How many parameters can there be in the overloaded operator function **`operator+`**?
15. Is it possible to use the name **`operator**`** as the name of the operator function?

Tasks for independent work

1. Write the function **`addWeek()`** adding one week to the date for the structure from section 1.1. Make it at first as the outer function and then as a function-member of the structure.
2. Add the examples 1.3.1 and 1.5.1. for the structures **`TDate`** and **`TStr`** by the operator **`operator>>`** of the overloaded stream input of the date and the line correspondently.
3. Realize the overloading of the stream output operator **`operator>>`** for the type of the union, as it was done in the section 1.3. For every input integer number (the type char) output its bit image.
4. Write the template for the sorting and print of the arrays from section 1.6 and demonstrate their specification for them.
5. Clean the memory set for the dynamic data from section 1.7. Is it possible to do in the template?

2. Work with the objects within one class

2.1. Operating access to the members of the class

Created classes may be used further for the elaborating of complex programs and creation new classes from already existing ones (**inheritance**). That is why the question of the access to the members of the class has an essential importance.

Every declaration inside the class denotes the privilege of the access to the names of the class, depending on the section the name appears. Every section begins with one of the key words: **private**, **protected** and **public**.

Construction outlook

```
class <className>
{
[private:]
       i><private data-members>
        < private constructors>
        <private members-functions>
protected:
       <protected data-members>
       <protected constructors>
       <protected members-functions>
public:
       <generally accessible data-members>
       <generally accessible constructors>
       <generally accessible destructor>
       <generally accessible members-functions>
};
```

is called the definition (declaration, announcement, interface) of the class because it denotes a new type.

The fields of the class may have any type, except the type of the same class (but they may be pointers or referring to this class). The initialization of the fields during the declaration is not allowed.

Any section may be absent or occur not once. The order of the sections location is not essential.

The section **protected** will be used further by inheritance. If the class is not inherited, then the section **protected** acts analogous to the section **private**.

In order to explain the action of the specifications of the access **public** and **private**, we'll consider the type of the data, given in the example 1.1.2, but change the key word **struct** for the class and the members-data put in the closed (private) section of the class and the members-functions – in the open (accessible) section.

Example 2.1.1:

```
#include <iostream.h>
// class with data-members d-day, m-month, y-year
class Tdate
{
     int d,m,y;
public:
     void init (int dd,int mm, int yy);
     void addYear(int n){y+=n;}
     void addMonth(int n){m+=n;}
     void addDay(int n){d+=n;}

 };
void TDate::init(int dd,int mm,int yy)
 { d=dd;
   m=mm;
   y=yy;
 }
.  .  .
 int main()
 {
//error(data-members are not accessible)
 TDate today={5,9,2013};
 TDate today;
 today.d=5; //error(variable d is not accessible)
 today.init(5, 9, 2013);  //correct
 cin.get();
 return 0;
}
```

Example 2.1.2:

```
#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.

Thus, the announcement of the base class in C++ gives such rights of the access and corresponding fields of visibility:

- **private** names have more limited access, allowed only to the function-members and friends of the given class. The access of derived classes to the private methods of the base classes is forbidden;
- **protected** names have an access, allowed to the members-functions of the given class and its derivatives (and friends);
- **public** names have an unlimited access, allowed to the methods of all classes and their objects.

The following rules are used at the creation of different sections in the announced class:

- sections may appear in any order, and their names may occur more than once;
- if the section is not named, then the compiler considers the next announcements of the names of the class **private.** Here takes place the difference of class and structure announcement – the latter is viewed by default as **public**;
- if the members of the data class are not used in the program, they are announced protected in order to let the access only for the methods of derived classes.

Using sections has a number of advantages:

First, if it is necessary to change the realization of the class (for example, to realize sorting differently), then it will be enough only to change the members-functions. As to the code of the user of the class, it won't change because it depends only on the open interface, and there won't be any need to copy it anew (though its recompilation might be necessary).

Secondly, to be able to use the class, its possible user will have to get acquainted only with the prototypes of open functions-members.

By default all the elements of the class are **private** and structure elements are open. Therefore constructions, given below, are equivalent.

```
class My {public: …};     ~     struct My { … };
```

Hence one can make a conclusion that structure is a class, the members of which are announced as **public**.

Object is an instance of the class. The definition of the class does not create the objects of the given class. They are created during the description (pointing out of the memory is taking place). Every object has its own choice of elements.

2.2. Constructors

Using functions of the type **`init( )`** from the example 2.1.1 for the initialization of the objects of the class often results in errors. The syntax rules don't oversee the object being initialized, therefore the programmer may accidentally work with the non -initialized object or initialize it twice. In C++ it is taken into account, and one can declare a special function, having a necessary goal – creation and initialization of objects. Since such a function creates (constructs) the object of the given type, it is called a constructor. The constructor is recognized by the name which coincides with the name of the class itself. If the class has a constructor, all the objects of this class will be initialized. The class may contain any number of constructors or not a single one. Constructors can't be announced virtual.

There are 3 types of constructors:

- default constructor has no parameters. If the class has not a single constructor (as in the example 2.1.1.), then the complier automatically will create one default constructor, which simply assigns the memory at the creation of the object of its class;

- constructor with arguments (with parameters) allows to initialize the object at the moment of its creation, using the given parameters – to call different functions, to assign the dynamic memory, to give variables the initial values, etc;

- copy constructor is meant for the creation of objects of the given class

by way of copying the data from already existing object of this class. Such constructors are especially suitable, even necessary for the creation of objects copies, which model dynamic structures of the data.

Constructors obey the same rules of overloading, as the other functions. Since they differ by the types of their arguments, the compiler chooses the necessary constructor in every concrete case.

Local variables are sometimes called automatic objects, and objects, created in the field of free memory – dynamic and are not initialized by default. Constructors for automatic objects don't return the value, answer for the creation of the instance of the class, and the memory is assigned at the input into the block from the stack.

Let us include constructors and the printing function into the example 2.1.1 and take away the function of the initialization:

Example 2.2.1:

```
#include <iostream.h>

//structure strDate is with data-members
// d-day, m-month, y-year

struct TStrDate
{
int d,m,y;
}today={5,9,2013};

//class with the data-members d-day, m-month, y-year
class TDate
{
     int d,m,y;

public:

//constructor, to which three arguments of the whole
// type are transmitted
// arguments of the integer type
     TDate(int d,int m,int y);

//constructor to which one argument of the
//type TStrDate is transmitted
     TDate(TStrDate);

//default constructor
     TDate(){d=today.d; y=today.y; m=today.m;};

//constructor with the data parameter
// in the form of const char*
     TDate(const char*); // till now – only prototype
     TDate(const TDate& obj); //copy constructor
     void print();
```

```
     void addYear(int n){y+=n;}
     void addMonth(int n){m+=n;}
     void addDay(int n){d+=n;}
 };

TDate::TDate(int d,int m,int y)
 {
   this->d = d;
   this->m = m;
   this->y = y;
 }

TDate::TDate(TStrDate strD)
 {
   d = strD.d;
   m =strD.m;
   y =strD.y;
 }

TDate::TDate(const TDate& obj)
 {
   d = obj.d;
   m =obj.m;
   y =obj.y;
 }

void TDate::print()
{
cout<<"\n year - "<< y << endl <<
      "\n month - "<< m << endl <<
      "\n day - "<< d << endl;
}
int main()
 {
// The call of the constructor with three parameters
 TDate day1(30,12,2013);

// the call of the constructor with the parameter of the
type TStrData
TDate day2(today);

// The call of the copy constructor
 TDate day3(day2);
 day3.addMonth(2);

// also the call of the copy constructor
 TDate day3=day2;
 day3.addMonth(2);

//the call of the default constructor (5 times)

 // print results
```

```
 day1.print();
 day2.print();
 day3.print();
 for(int i=0; i<5; i++)
       days[i].print();
 cin.get();
 return 0;
}
```

Note that there are cases when the copy constructor is called implicitly. It takes place by the transfer of the object into the function by value and by the return of the object from the function by the return operator.

There are rules, by which an argument can get a value by default. If the default parameters are used at the call of the constructor, it must not cause the conflict with the call of the constructor without parameters or with the call of the other constructor with the parameters where default parameters are also used.

Let us rewrite the class from the example 2.2.1, using for some arguments the value by default, and let change data-members d, m, y to one variable date of the structure type **`TStrDate.`**

Example 2.2.2:

```
#include <iostream.h>
//structure TStrDate with data-members
// d-day, m-month, y-year
struct TStrDate
{
int d,m,y;
}today={5,9,2013};

// class TDate with data-members date
class TDate
{
     TStrDate date;
public:
//constructor, to which 3 arguments are transmitted
// of the integer type;
// value by omission
// (the name of the argument may not be mentioned)
     TDate(int d,int m=9,int = 2000);

// constructor, to which one argument is transmitted
// of the type TStrDate
// in the conflict with the omission constructor
// TDate(TStrDate=today);
     TDate(TStrDate);

//default constructor
       TDate()
      {
       date.d=date.m=1;
       date.y=2000;
```

```
        }

// constructor with the data-parameter
// in the form const char*

     TDate(const char*); // till now only prototype
     TDate(const TDate& obj); //copy constructor
     void print();
     void addYear(int n=1){date.y+=n;}
     void addMonth(int n){date.m+=n;}
     void addDay(int n){date.d+=n;}
 };

TDate::TDate(int d,int m,int y)
 { date.d = d;
   date.m = m;
   date.y = y;
 }

TDate::TDate(TStrDate strD)
 {
 date=strD;
 }

TDate::TDate(const TDate& obj)
 {
 date = obj.date;
 }

void TDate::print()
{
cout<<"\n year - "<< date.y << endl <<
      "\n month - "<< date.m << endl <<
      "\n day - "<< date.d << endl;
}

int main()
 {
// three calls of the constructor with three parameters
// e parameters by omission
TDate day1(30,12,2013);
 TDate day11(30,12);
 TDate day12(25);
// TDate day12(5,,2005);  error
 day1.print();
 day11.print();
 day12.print();

// the call of the constructor with the parameter
// of the type TStrData
 TDate day2(today);
```

```
//the call of the copy constructor
 TDate day3(day2);
 day3.addMonth(2);
// omission parameter
 day3.addYear();
 day3.print();

 cin.get();
 return 0;
}
```

In two last examples the object variables of the type **TDate** depend on the global variable **today**. Now class **`TDate`** may be used only in the places where the variable **today** is defined. And this makes class **`TDate`** not suitable for usage. In order to solve this problem static members of the class are used.

2.3. Static members of the class

A variable which is the part of the class, but not the part of the object of this class, is called the static data-member of the class. There is only one copy of the static member, different from the ordinary members, when every object of the class has its independent members. Similarly, the function which needs the access to the members of the class on the condition that it should not be called to the concrete object of the class, is called the static function-member.

In order to describe the static elements the key word **`static`** is used.

The action of the qualifier of the access is spread on the static members of the class, therefore the static data members, described as **`private`**, can be changed only by means of the static function members

Memory, dealing with the static data-member, is not taken into consideration, when the size of the object is defined by the operation **`sizeof`**.

Now we shall add to the class **`TDate`** the static variable **`today`** and the static functions **`setToday`**() and **`getToday`**().

Example 2.3.1**:**

```
#include <iostream.h>
// structure TStrDate with the variables
// d-day, m-month, y-year
struct TStrDate
{
int d,m,y;
};

// class TDate with the variable date
class TDate
{
     TStrDate date;
```

```
//the static variable
     static TStrDate today;
public:
//constructor, receiving, only one argument
// of the type типу TStrDate
     TDate(TStrDate strD) {date=strD;}
// default constructor
     TDate(){date=today;};
     void print();
//static functions
     static void setToday(int d,int m,int y);
     static TStrDate getToday(){return today; }

 };

//initialization of the static variable
 TStrDate TDate::today={5,9,2013};

//description of the static function
 void TDate:: setToday(int d,int m,int y)
 {
 today.d=d;
 today.m=m;
 today.y=y;
 }

void TDate::print()
{
cout<<"\n year- "<< date.y << endl <<
      "\n month - "<< date.m << endl <<
      "\n day - "<< date.d << endl;
}

int main()
 {
//the call of default constructor
 TDate day1;
 day1.print();
 TDate day2;

//call of the static function through
// the name of the class
TDate::setToday(8,3,2013);
day2.print();

//call of the static function through
// the name of the object
 day2.setToday(23,2,2013);
 TDate day3(TDate::getToday());
 day3.print();
 cin.get();
 return 0;
```

```
}
```

It is evident from the example that one may address the static functions in the same way as any other members. Besides, it is possible to address the static member without pointing the name of the object. Instead, the name of the class itself with the operator is used as a qualifier of its name.

2.4. Destructors

Constructors initialize the object, that is, they create the environment in which functions -members work. Sometimes creation of such an environment foresees the allotment of recourses (files, memory) which must be set free after their usage. Therefore some classes need the function, the call of which is required for the object to be destroyed. Such functions are called destructors. They clean the functions and set free the resources.

Destructor is a member function of the class, having no parameter and not returning the value. The name of the destructor is made up of the symbol of "~" and the name of the class. The class must have no more than one destructor.

If the destructor is not defined, the complier automatically creates the empty destructor. Destructors are called automatically (implicitly), when the object leaves the sphere of visibility:

- for global objects – at the output function **main**();
- for local objects – at the output from the block in which they have been announced;
- for the objects, given through the pointers (created dynamically), destructors are called at the operator's exertion **delete**.

Example 2.4.1:

```
#include <iostream.h>

//structure TStrDate with variables
// d-day, m-month,y-year
struct TStrDate
{
int d,m,y;
};

//клас TGroupDate with the array
// of the students' birthdays

class TGroupDate
{
  //dynamic birthdays array of the students of the group
     TStrDate* dates;
     // the number of students in the group
     int count;

public:
```

```
     //default constructor
      TGroupDate(){count=0; dates=NULL;};

     // constructor with parameters
     TGroupDate(int size, TStrDate* mas);
     TGroupDate(int size)
             {dates=new TStrDate[count=size];}

     //destructor
     ~TGroupDate(){delete[] dates;};
     void input();
     void print();
 };

//constructor, creating dynamic array
//of the students birthdays using the array mas
TGroupDate::TGroupDate(int size, TStrDate* mas)
{
count=size;
dates=new TStrDate[count];
for (int i=0;i<count;i++)
        dates[i]=mas[i];
}

void TGroupDate::print()
{
for (int i=0;i<count;i++)
cout<<"\n" << i+1 <<") Рік - "<< dates[i].y <<
      " month - "<< dates[i].m <<
      " day - "<< dates[i].d << endl;
}

void TGroupDate::input()
{
delete[] dates;
cout<< "\n input the number of students in the group: ";
cin>>count;
dates=new TStrDate[count];
cout<< "\n input the list of the students' birthdays (day,
        month, year) in the group:";
for (int i=0;i<count;i++)
cin >> dates[i].d >> dates[i].m >> dates[i].y;
}

int main()
 {
 cout<< "\n Group 201\n";
 TGroupDate days;
 days.input();
 days.print();

 cout<< "\n Group 202\n";
```

```
 TStrDate mas[]={{14,4,1993},{3,11,1994},{23,10,1994}};
 TGroupDate days202(3,mas);
 days202.print();
 cin.get();
 return 0;
}
```

In the given example we have described the class that works with the dynamic array of the students' birthdays in some group. The constructors of the class point out the place for the dynamic array **dates**, and destructor **~TGroupDate**() sets free the memory which shows the pointer **dates.** Note, that the input of data about the object and its construction – are two different operations. The input function may be called only for the existing object, for which the memory has been assigned. The memory might not be sufficient for the other object, therefore in the function **input**() the old memory has been set free and a new one for the necessary object assigned.

2.5. Dynamic creation and deleting of objects

In the language C++ there are operators **new** and **delete**, which could be used instead of the library functions of language C **malloc**(), **calloc**(), **free**().

The operator **new** assigns the memory, necessary for the location of the variable or the array of the corresponding type (dimension). It can also assume the initial value.

```
// point out the memory for the variable
// of the type int
  int *p1=new int;

// point out the memory for the variable
// of the type double and initialize it
  double *p2=new double(25.7415);

// point out the memory for the string
  char *p3=new char[80];

// point out the memory for the array of real numbers
// of dimension 50
  float *p3=new float[50];
```

Here the pointers **p1**, **p2**, **p3**, **p4** indicate the addresses of the pointed out memory blocks.

Memory can be set free by means of the operator **delete**.

```
    delete p1;
    delete p2;
    delete[] p3;
    delete[] p4;
```

Remark that the operators **new** and **delete** work much more effective than the library functions **malloc**(), **calloc**(), **free**(). Besides they, as well as other operators, can be overloaded for the own class types (section 2.11).

Using the operators **new** and **delete** one can call out the necessary constructors and destructor and, in case of necessity, to create objects dynamically.

We can watch the whole circle of the existence of class instance in case if we add into the bodies of constructors and print destructor the information about the beginning of the existence of the object (into the constructors) and the end of its existence (into the destructor).

Example 2.5.1:

```
#include <iostream.h>
#include <stdlib.h>
class TDemo{
  private:
     char *name;
     static int number;
  public:
     TDemo();
     ~TDemo();
 };
// constructor realization
  TDemo::TDemo()
   {
      name=new char[10];
      char s[4];
      itoa(++number,s,10);
      strcpy(name,"Object");
      name=strcat(name,s);
      cout << "Creating " << name << endl;
   }

TDemo::~TDemo() // destructor realization
   {
      cout << "Deleting " << name << endl;
      delete[] name;
   }

int TDemo::number=0;

int main()
{
TDemo *p1,*p2;
   {// the block of the existence of objects and *p1
     //creation of objects №№1-3

TDemo objects[3];

// creation of object №4
```

```
p1=new TDemo;
...
// delete of object №4

delete p1;
}

//dynamic creation i deleting of object *p2
p2=new TDemo;         // creation of object №5
...
delete p2;            // deleting of object №5

cin.get();   // delaying of the window with the result
return 0;
}
```

Result of the program work

```
Creating Object1
Creating Object2
Creating Object3
Creating Object4
Deleting Object4
Deleting Object3
Deleting Object2
Deleting Object1
Creating Object5
Deleting Object5
```

Sometimes while working in the regime Console in the environment **C++Builder**, the necessity to draw the given object appears. In order to realize this task the functions **Windows GDI** are as a rule implemented.

The other way is using the property **Canvas**, existing in many components of the library **VCL** (**Visual Component Library**).

To make the access to the property **Canvas** possible one can dynamically create with the help of the operator new the form, which is a class component and further through the pointer address the needed properties and methods of the class **Canvas**.

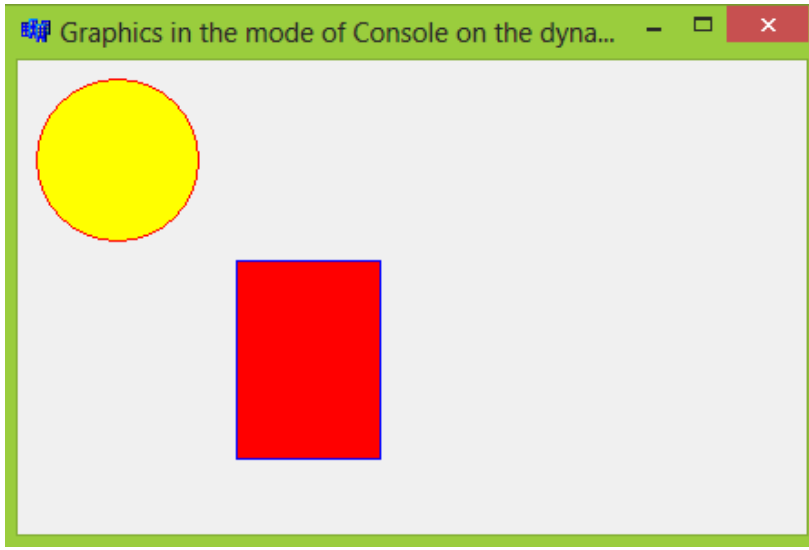
Example 2.5.2:

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

int main(int argc, char* argv[])
{

// dynamic creation of the form
```

```
TForm* pForm;
pForm = new TForm (Application);
pForm->Caption =
"Graphics in the mode of Console on the dynamically created
form";
pForm->Width=450;
pForm->Height= 300;
pForm->Top=400;
pForm->Left=400;
pForm->Visible=true;
// color of the contour (pen)
pForm->Canvas->Pen->Color = clRed;

// color of the brush
pForm->Canvas->Brush->Color = clYellow;

// yellow ellipse, outlined with the red contour
pForm->Canvas->Ellipse(10, 10, 100, 100);

// c (перо)
pForm->Canvas->Pen->Color = clBlue;

//pouring color (brush)
pForm->Canvas->Brush->Color = clRed;

// Red rectangle, outlined with the dark blue contour
pForm->Canvas->Rectangle(200, 110, 120, 220);

// colors in the window Console
// white color on the black background
cout<<"without color"<<endl;
SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),
FOREGROUND_GREEN | BACKGROUND_RED);

// green text on the red color<<"with color"<<endl;

SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),
FOREGROUND_GREEN |FOREGROUND_RED | BACKGROUND_BLUE);

//yellow text on the blue background
cout<<"with color"<<endl;

// white text on the black
backgroundetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HA
NDLE), FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE
);
cout<<"without color"<<endl;

// delay of the time
getch();

// deleting of the form
```

```
pForm->Hide();
pForm->Release()
return 0;
}
```

Result of graphic output on the dynamically created form:



2.6. Friends of the class

One often needs the necessity in writing functions, having the access to the private sections of several classes. But function cannot be a member of two classes. Therefore in C++ such a method of the language is foreseen, which conveys to the other class or function the right of the access to the closed part of the class by means of the reserved word **`friend`**.

Non-member function, that received the right to the access to the closed part of the class, is called the friend of the class (friendly function).

Friendly function is an ordinary function, it has no other peculiarities, except the right of the access to the closed part of the class. Particularly, friend-function doesn't have the pointer **`this`**, since it doesn't have the full rights of the function-member. The key word **`friend`** is introduced in the prototype function in front of its name and type that returns. The description of the friendly function can be placed in the closed or open part of the class description, in what place in particular, it doesn't matter. By the description of the body of the friendly function the scope operator is not mentioned because the function is not a member of the class.

Let us for example overload the operator which finds a product of a matrix

(**TMatrix**) by a vector (**TVector**). Since classes **TMatrix** and **TVector** hide their elements, announced in the section private, the access to their elements can be opened for other functions, announcing them as **friend**.

Example 2.6.1:

```
#include <iostream.h>

// the above mentioned announcement of the class TMatrix
//(in the class TVector class Tmatrix is used)
class TMatrix;

//class vector
class TVector
{
  float v[4]; //array having, coordinates of the vector
public:
  //constructor
TVector(float x1=1, float  x2=1,
       float  x3=1, float  x4=1)
      {
         v[0] = x1;
         v[1] = x2;
         v[2] = x3;
         v[3] = x4;
       }
 // vector print();
 //matrix vector multiply(friendly function)

 friend  TVector multiply(const TMatrix&,
                          const TVector&);
};

void TVector::print()
{ // vector print  cout <<v[0]<<" "<<v[1]<<" "<<v[2]<<"
"<<v[3]<<endl;
}

// class TMatrix
class TMatrix
{
       TVector v[4];//array of vectors
public:
       TMatrix();
       TMatrix(TVector, TVector, TVector, TVector);
       void print ();
//matrix vector maltiplay (friendly function)
friend  TVector multiply (const TMatrix&,
                          const TVector&);
};

TMatrix::TMatrix()//default constructor
```

```
{
// initialization of vectors' array
for (int i = 0; i<4;i++)
        v[i] = TVector(1,1,1,1);
}

// constructor, receiving arguments -
//vectors for matrix creation
TMatrix::TMatrix(TVector x1, TVector x2,
                 TVector x3, TVector x4)
{
       v[0] = x1;
        v[1] = x2;
        v[2] = x3;
        v[3] = x4;
}

void TMatrix::print()
{ //matrix print
for (int i = 0; i<4;i++)
        v[i].print();
}

//matrix vector multiply (friendly function)

TVector multiply(const TMatrix& m, const TVector& v)
{
       TVector rez;
           for(int i=0;i<4;i++)
                 {
                  rez.v[i] = 0;
                   for(int j=0;j<4;j++)
                     rez.v[i] += m.v[i].v[j]*v.v[j];
                  }
    return rez;
}

int main()
{
    TVector v(1,2,3,4), rezv;
    TMatrix m;
    cout<<"Matrix m:\n";
    m.print();

    cout<<"Vector v:\n";
    v.print();

// the call of friendly function multiply(),
// having an access
// for the hidden data-members TVector and TMatrix
    rezv = multiply (m,v);
```

```
    cout<<"Vector r:\n";
    rezv.print();
    cin.get();
    return 0;
}
```

If **fun**() – is some member function **TMyClass1** and all member functions of the class **TMyClass2** must have an access to the closed members of the given class **TMyClass**, then in the announcement of the class it is shown like this:

```
class TMyClass2; // above-mentioned announcement of the
class class TMyClass
{. . .
       friend void TMyClass1::fun();
       friend class TMyClass2;
. . .
};
```

Class **TMyClass2** is now becomes friendly for the class **TMyClass.**

2.7. Overloading operations for classes

In **C**++ there is a possibility of standard operations overloading:

+	-	/	*	%	^	&	|	~	!	=	<	>
+=	-=	/=	*=	%=	^=	&=	|=	<<	>>	>>=	<<=	==
!=	>=	<=	&&	||	++	--	->*	,	->	[]	()	

Besides, syntax allows to overload the operations **new**, **delete.**
Operations which are not allowed to overload:

- . (member class choice);
- .* (member class choice through the pointer);
- :: (the operator, allowing the visibility field or the scope operator);
- **?:** (condition operator).

The name of the operator function begins with the key word **operator** and the symbol @, where @ – is one of the operations, given in the table. Operator function **operator@** is used to denote a new overloaded action of the concrete language operator. Identically, as in the case with the overloaded functions, the compiler distinguishes different functions by the number and type of input parameters. It enables implementing the objects in the expressions, instead of conveying them as the parameters in the function. The language C++ appoints to every operator the corresponding name of the function, consisting of the key word **operator**, the sign of the operation and the argument of corresponding types:

```
<the type of the result> operator <operator symbol>
```

```
(<input arguments);
```

If in the example 2.6.1 we rename the friendly function using **operator***, then instead of calling the function

```
rezv = multiply (m,v);
```

it is possible to use the expression with multiplication operation of two objects, namely:

```
// the shortened call of operator function
rezv = m*v;
// obvious call of operator function
rezv = operator*(m,v);
```

It is evident, that operator function can be called as any other function. The implementation of operator function as operator is simply a shortened form of its obvious call.

As well, as in the case with structures, there are unary (one argument) or binary (two arguments) operators. If for the structures it is reasonable to denote the operator functions only as usual outsider functions, then for the classes, as a rule, three ways of operator functions overloading are implemented. [11, 25]:

- as a member-function of the class;
- as a friendly function;
- as a usual function

Consider the example of the class **TComplex** (complex integer), for which we shall denote three operator functions by three different ways:

- **operator+** – member-function of the class;
- **operator-** – friendly function;
- **operator*** – usual function.

Example 2.7.1:

```
#include <iostream.h>
class TComplex
{
// real and imaginary parts correspondently
      float re,im;
public:
       TComplex(float r=0., i=0.) //constructor
              {
                re=r;
                im=i ;
              }
// access functions
       oat getRe(){return re;}
       float getIm(){return im;}
```

```
//operator overloading + (member of the class)
       TComplex operator+(const TComplex&)const;

//operator overloading - (friend of the class)
friend TComplex operator-(const TComplex&,
                          const TComplex&);
};

//operator overloading * (usual function)
TComplex operator*(TComplex& one, TComplex& two)
{
float re1=one.getRe();
float im1=one.getIm();
float re2=two.getRe();
float im2=two.getIm();
return TComplex(re1*re2-im1*im2, re1*im2+im1*re2);
}

TComplex operator-(const TComplex& one, const TComplex&
two)
{
       return TComplex(one.re-two.re,one.im-two.im);
}

TComplex TComplex::operator+(const TComplex& two)const
{
       return TComplex(re+two.re,im+two.im);
}

int main(int argc, char* argv[])
{
   TComplex op1(1.5,-3.32),op2(5.2,7.02);
   TComplex sum, sub, rez;

   sum=op1+op2;
   sub=op2-op1;
   rez=op1*op2-sum*sub;
   cout<<rez.getRe()<<"+i*("<<rez.getIm()<<')'<<endl;
   cin.get();
   return 0;
}
```

Operations overloading are submitted to such rules:

- the number of arguments is saved: their number is strictly defined (except the call operators), as well as operations priorities and the associations rules, which are used in the standard data types;
- for standard data types operations are not submitted to overloading;
- the overloaded operator-function can't have default parameters, is not inherited and can't be announced static;
- operators **+, -, *, &** can be both unary and binary;
- operators **[], (), =, ->** can be only members of the class;

- it is impossible to define new lexical symbols of the operations (for example, one can't redefine the operation ** for its rising to the degree);
- it is impossible to overload the operations for the non-class types (there are two possibilities for the operator function become a member of some class, created by the user, or at least one of the operands of this operation was of the class type);
- there are no limitations for the type which returns the overloaded operator.

2.8. Binary and unary operations

Binary operation can be defined as a function- member receiving one parameter, or as a friend function, or as usual function, receiving two parameters [13]. So, for any other binary operation @ the expression **one@two** can be interpreted as **one.operator@(two)** or as **operator@(one, two)**. If both operator functions are defined, the expression **one@two** is error. Unary operation, prefixed or postfixed, can be defined as a function-member, receiving no parameters or as a friend-function or usual function, receiving one parameter. Thus, for any unary operation @ expressions **obj@** or **@obj** can be interpreted as **obj.operator@()** or **operator@(obj)**. If we define both, then the expressions **obj@**, and **@obj** will be errors.

Consider the examples of announcing the operations for some class T and denote error expressions.

Example 2.8.1:

```
  class T
{. . .
 // friendly operators

// unary minus
       friend T  operator-(T);
// binary minus     friend T  operator-(T,T);
 // error no operands
       friend T  operator-();
 // error: ternary operation
    friend T  operator-(T,T,T);
 // unary prefixed decrement
    friend T& operator--(T&);
 //unary postfixed decrement
       friend T operator--(int,T&);
 //unary postfixed decrement
       friend T operator--(T&,int);
       . . .
public:
//class member operators
//(with non-obvious first parameter this):

// unary & (receiving the address)
      T* operator&();
```

```
//binary &(logical bitwise multiplication)
      T  operator&(T);
// error ternary operation
      T  operator&(T,T);
//unary prefixed decrement
      T& operator--();
//unary postfixed decrement
      T  operator--(int);
  . . .
};
```

When operations ++ i – are overloaded, it is sometimes necessary to denote both of their forms (prefixed and postfixed).The name of the operator function, for example, operator--, is the same for prefixed and postfixed forms. That was why in order to distinguish two forms one more (for friendly operator– the first or the second, depending on the compiler's version) parameter of the whole type, not used but only denoting the necessary choice of the operator function, is implemented.

2.9. Assignment operator

Assignment operator function, as a rule, returns the reference to the object, for which it was called, and receives as a parameter one more argument – reference to the object which is assigned. It is also allowed to use **void** as the type of the result.

Syntax of two forms of assignment operator:

```
<class type &> operator=(<[const]class type &>);
<void> operator=(<[const]class type &>);
```

The case, in which the type of the result – **void**, is simpler in writing, but it works only at one-time usage and doesn't work (points to the compilation error) at the chain objects assignment (Ob1=Ob2=Ob3;).

In the body of the operator assignment function it is necessary to set free the memory meant for the old object and assign a new one for the assigned object. While deleting the old object one needs to control the operation of self-assignment. After this control it is necessary to copy the object content into assigned memory.

Example 2.9.1:

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

class TString
{
      char* s;
public:
      TString(char* t)
```

```
            {
            s=strdup(t);
            }
      TString(const TString&);
      ~TString()
            {
            delete[] s;
            }
      TString& operator=(const TString&);
      void print()const;
};

void TString::print()const
{
cout<<s<<endl;
}

TString::TString(const TString& a)
  {
   s=new char[strlen(a.s)+1];
   strcpy(s,a.s);
   }

TString& TString::operator=(const TString& a)
  {
        if (this != &a) // self-assignment control
        {
        delete[] s;
        s=new char[strlen(a.s)+1];
        strcpy(s,a.s);
        }
        return *this;
  }


void main()
{
// the call of the constructors with the parameter
TString s1("abcabc6"),s2("123");

// the call of the copy constructor
TString s3(s2);
s3.print();

// again the call of the copy constructor
// (not assignment)
TString s4=s2;
s4.print();

// the call of the assignment constructor
// (one-time assignment)
s2=s1;
```

```
// the call of the assignment constructor
//(chain assignment)
s2=s4=s1;

// print of the results
s1.print();
s2.print();
s3.print();
s4.print();

cin.get();
}
```

Remark: If among the data-members of the class there are pointers, to which memory is dynamically assigned in the constructor, then to make class to work correctly is necessary to define, besides constructors, a destructor, assignment operator and copy constructor.

Example 2.9.2:

```
class TDemo
{
        . . .
public:
// creates the object
        TDemo([<parameters>]);

// creates the copy of the object
        TDemo(const TDemo &);

// deletes the object (sets the memory free)
        ~TDemo();

// sets the memory free, points out anew one for the
// necessary object, copies the content of the object
// into the pointed out field of the memory
        TDemo& operator=(const TDemo&);
        . . .
};
```

2.10. Operators of the call, indexing and type conversion

Operators of the call, indexing and transformation of the type can be denoted anew only as class members. They can't be friends of the class. Besides, one can't use parameters with default values.

- Type conversion operator can be announced for conversion into any data type. It is called without the arguments, and the type of the returning value is denoted by the name of the operator. The returned type is not indicated explicitly.

Syntax:

```
operator <transformation type>();
```

Type conversion operators are usually not called explicitly, since the compiler can often denote itself when to call them.

- Indexing operator **operator[]** can be overloaded only with one argument of the arbitrary type and can return also arbitrary type as its value. In case it is foreseen to use **operator[]** from the left side of the assignment sign, then it is necessary to return the assignment sign.

Syntax:

```
<the type of the result> operator[](<The type of the input parameter>);
```

- Call operator **operator**(). If call operator is overloaded, then the brackets on the left of the name of the object are interpreted as the return function to return the assignment sign. Operator arguments are formal arguments conveyed to the object-function by the call. The call operator can return any type and receive the arguments of any other types of arbitrary number. And it obeys the same rules as any other function.

Syntax:

```
<type of the result> operator()(<input parameters>);
```

Example 2.10.1:

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

class TString
{
       char* s;
public:
       TString(char* t){s=strdup(t);}
       ~TString(){delete[] s;}

// call operators
       int operator()(int pos, char c);

//int operator()(int pos=0, char c); one can't write in
//this way

       int operator()(char c)  {return operator()(0,c); }

// index operators
       char& operator[](int i){return s[i];}
       int operator[](char c)  {return operator()(0,c); }
```

```
// transformation type operators
       operator int() {return atoi(s);}
       operator char*(){return s;}
};

int TString::operator()(int pos, char c)
       {
       for(int i=pos+1;i<strlen(s);i++)
       if(s[i]==c)return i;
       return -1;
       }

void main()
{
TString s1("abcabc6"),s2("123");

// operator char* (char*)s- type conversion operator
// doesn't run into conflict with (int)s
cout<<" s1="<<(char*)s1;

// operator call operator()(int pos, char c)
cout<<"\n s1[3]="<<s1(2,'a')

//operator call operator() (char c)
<<"\n s1[1]="<<s1('b')

//operator call[](char c)
<<"\n s1[1]="<<s1['b'];

// operator call operator[](int i) twice
 s2[2]=s1[6];

// implicit operator call operator int
int i=s2;
cout<< "\n i="<<i;

cin.get();
}
```

Result of the program work:

```
S1=abcabc6
 S1[3]=3
 S1[1]=1
 S1[1]=1
 i=126
```

2.11. Overloading of the operator new and delete

In order to implement the non-standard operations of memory management the overloading of operators **new** and **delete** is used.

The operator **new** can be overloaded both in the form of outer function, and in the form of function-member of the class. The default operator **new** has such interface:

```
void* operator new(size_t <the number of bites>);
```

The argument assigns the number of bites. The address of the assigned memory is returned.

After the mandatory parameter non mandatory additional parameters can be placed. At the call of the operator new the additional arguments are indicated between the key word **new** and in front of the name of the class, they are conveyed to the overloaded operator after the size of the block, assigned by the complier.

For example,

```
MyClass* p = new (<the list of additional arguments>)
                    MyClass;
```

For such a call the announcement of the function-member of the operator **new** looks like

```
void* operator new(size_t <the number of bites>,
            <list of additional arguments>);
```

In this case the mandatory first parameter is assigned by the compiler.

In order to implement the non-standard operations of memory management together with the operator **new** we, as a rule, overload the operator **delete**. There are two interfaces which are called implicitly by the compiler while deleting the object:

```
void operator delete(void* < address>);

void operator delete(void* < address>,              size_t
<  the number of bites>);
```

Example 2.11.1:

Task: Input the data into the associated array of pairs (pair is a word and the number of its appearances). Use the overloaded operator **new**. Create the function of the input into the array of one function and the function of the output of all array pairs.

```
#include <iostream.h>

struct TPair {
         char* name;
         int val;
         void* operator new(size_t bytes,char* word)
                              {
                              TPair* p= ::new TPair;
                              p->name=strdup(word);
                              p->val=1;return p;
                              };
   };

class TAssoc {
         TPair vec[100];  // pairs vector
         int max;         // size
// index of the first non-implemented element
         int ifree;
 public:
         TAssoc(){
                     ifree=0;
                     max=100;
                  };
         ~TAssoc();
         void input(char*);
         void print_all();
  };

TAssoc::~TAssoc()
{
for (int i = 0; i<ifree;i++)
delete[] vec[i].name;
ifree=0;
}

void TAssoc::input(char* s)
  /* work with the set of the pairs TPair:search of the
word s between the pairs and the increase of the whole part
TPair (the number of words);
Creation of the new pair TPair, if s is not met in the
array.
  */
  {
          if (ifree==max)
```

```
                {
                     cerr<< " overflow\n"
                     exit(1);
                }
          if (ifree)
               for (TPair* p =&vec[ifree-1]; vec<=p; p--)
                              if (strcmp(s,p->name)==0)
                                {
                                 p->val++;
                                 return;
                                 }
          TPair* temp=new (s)TPair;
          vec[ifree++] = *temp;
          delete temp;
   }

//print function of  all the pairs
  void TAssoc::print_all()
  {
  for (int i = 0; i<ifree;i++)
  cout << vec[i].name << ": " << vec[i].val << "\n";
  }

 int main()
  {
       {
       TAssoc vec;
       char buf[80];

// the end of the input Ctrl-z
       while (cin>>buf) vec.input(buf);
       vec.print_all();
       }  //  destructor is called

       cin.get();
       return 0;
  }
```

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

Task: Create the class **TSetInt (**set of the integer numbers) using. Constructors, destructor, copy constructor.

To define the operations:

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

To define the functions:

- print sets;
- the number of elements calculation in the sets;
- comparison with the empty set;
- ordering the sets;
- determining if the number belongs to the set;
- determining 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.

Example 2.12.1:

```
#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
```

Questions for self-control

1. What names of the sections are used in the description of the class?
2. By what are the advantages of the access to the names in the class and structure differed?
3. What is the encapsulation?
4. What are the ways of addressing the open members of the class?
5. Is it possible to address the closed members of the class?
6. Why are the constructors necessary and how many of them can there be in the class?
7. What is the difference between the function of initialization and constructor?
8. In what cases is the constructor called not obviously?
9. When is it allowed not to write constructor and destructor?
10. How can one address the static members of the class?
11. How is the static member of the class initialized?
12. Is it necessary to call the destructor while deleting the dynamically created object?
13. Why are friendly functions necessary?
14. When is the need to appoint the friendly class?
15. By what means can one overload the binary operator for the classes?
16. How is the interface of post-prefixed and prefixed unary operations assigned?
17. In what cases is it necessary to overload the assigned operator?
18. On what does the necessity of returning the assigned type in the indexing operator depend?
19. Is it allowed to overload several indexing operators for one class?
20. In what form is the operator of the transformed type most often used?

Tasks for the independent work

1. Write the body of the constructor **`TDate(const char*)`** in the class from section 2.2 and implement it in the function **`main`**(), creating the objects as automatic and dynamic variables.
2. Add the class **`TDate`** from section 2.2 conversion operator of the type into **`char*`** and implement it for the implicit call at the data print.
3. Add the class **`TDemo`** from section 2.5 the call operator which will return the value of one of the data-members.
4. Copy the functions **`addYear`**(), **`addMonth`**(), **`addDay`**() from the example 2.1.1 and the function **`print_all`**() from the example 2.11.1, announcing them friendly.
5. Copy the operations of the relation of the class **`TSetInt`** from the example 2.12.1, announcing them friendly, and the operation of the unity of two sets change into the operator member of the class.

3. Creation of new classes using the existing ones

3.1. Building classes with member-objects

Using class or its variety – structure within the other class is called the aggregation.

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.

Example 3.1.1:

// 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";
}
```

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.

The structure **TLink** is announced in the private section of the class **TStack. TLink** 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 accessible functions of member-class **TStack** are:

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

The constructor **TLink**::**TLink**() assigns the pointer to the object variable **data** and the pointer to the next element **next**. The line **head = new TLink(dat**, **head**) in the function **TStack**::**push** creates a new element **TLink** and adds it to the stack.

Remark: The destructor **~TLink**() does not set free the place in the memory, because:

- the operator **delete** cannot set free correctly the pointer to **void**;
- if to set free the place in the destructor, then the function-member pop would turn the pointer to the deleted object, and it would be the error.

Let us test the class **TStack** for the elements of the whole type.

Example 3.1.2:

```
// 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;
}
```

Classes can be global and local. If one class is put into the other class (aggregation), then both classes have no particular rights of the access to the elements of each other.

In the local class it is forbidden to use the automatic variables from the field, in which it is described.

The local class cannot have the static elements, and the methods of this class can be described only within the class.

3.2. The templates of the classes

The templates of the classes are often called the generators of the classes or generalized (abstract), or generic classes, or parameterized

types. They make it possible to define the structure of the whole family of classes, due to which the compiler creates the concrete classes, depending on indicated parameters.

The templates of the classes are used only in the cases, when there is a necessity to create several quite identical classes but still different. There is a library of the templates of the classes **STL** – **Standart Template Library** [8, 10, 16, 25], in which many useful containers have been worked out.

While describing both the template of the class (the key word **template**), and the template of the function (section 1.6), the notation **<class T>** or **<typename T**>, is used which means the announcement of the unknown type **T**, but by the generation of the concrete class it is an obvious argument, and the compiler will automatically substitute the corresponding type.

Besides the generalized type, the inbuilt type or again the template can be used. There can be several parameters of the template, and they are pointed out by comas. For any parameters of the values can be placed by default.

In a general case the template of the class looks like:

```
template <опис параметрів шаблону> <опис класу>
```

Methods of the template of the class automatically become the templates for the functions. If the method from the template is described outside the template, its opening line must look like this:

```
template <the description of the template parameters>
<the type, returned as a function> <the name of the class>
<parameters of the template setting>::<function family>
(<the list of function parameters>)
```

Rewrite the example 3.1.1 in the way one could work with any data types without their obvious transformation. For this we shall create two templates: for the structure **TLink** and for the class **TStack**. With the help of these templates we shall generalize the objects-stacks with the whole and character elements. Remind that in section 1.6 we have already considered the construction template for the creation of the templates functions.

Example 3.2.1:

```
#include <iostream.h>
template <class T>
struct TLink
{
```

```
        T* data;
        TLink* next;
        TLink(T* d, TLink* n){data = d; next = n;}
        ~TLink(){};
};

template <class T >
class TStack
{
// aggregation, using the pointer
        TLink<T>* head;
public:
        TStack(){ head = NULL; }
        ~TStack();
        void push(T* d){head = new TLink<T>(d,head);}
        T* peek(){return head->data;}
        T* pop();
};


// the description of the template function
// outside the class
template <class T>
T* TStack<T>::pop()
{
        if(head == NULL) return NULL;
        T* rez = head->data;
        TLink<T>* oldHead = head;
        head = head->next;
        delete oldHead;
        return rez;
}

//the description of the template function
// outside the class
template <class T>
TStack<T>::~TStack()
{
        while(head)pop();
        cout<<"\nTStack empty";
}

int main(int argc, char* argv[])
{
TStack <int> my1; //template concretizing
TStack <char> my2;//template concretizing
int el1,el2,el3;
cout<<"Input elements\n";
cin>>el1>>el2>>el3;

my1.push(&el1);
my1.push(&el2);
```

```
my1.push(&el3);
// remove el3
my1.pop();
char c1='a',c2='b';
my2.push(&c1);
my2.push(&c2);
// print el2 i c2(dereference of the pointers)
cout<<*my1.peek()<<endl<< *my2.peek();

cin>>c1;   // window delay with the result
return 0;
}
```

Remark [25]:

- the templates of the classes may contain static elements, friendly functions and classes;
- it is impossible to define friendly templates within the template;
- local classes can't contain the templates as their elements

3.3. Inheritance

Besides aggregation there is one more method of implementing the existing classes while building new ones – it is inheritance.

Inheritance enables us practically without limitations to build successively and extend the classes. Starting with the simplest classes, it is possible to create the derived classes, complicating, extending or limiting their functions.

The implementing of inheritance, especially while elaborating big program projects is well coined with the technique of structural programming (from general to partial) and in many ways stimulates such an approach. At the same time the complexity of the program code is in general considerably shortened. The derived class acquires all function-members and data-members of their base class and all its ancestors from the class hierarchy.

Using inheritance the base class acquires new attributes and functions. The derived class acquires new data-members. While working with the objects, the programmer, as a rule, chooses the most suitable existing class for solving a particular task and creates one or several its inheritors, extending the possibilities of base classes. Friendly functions make it possible for the derived class to receive the access to all data-members of the outer classes.

Besides, the derived class can overload the inherited function-members when their realization in the base class doesn't suit.

C++ makes it possible to announce the derived class which inherits

the data and functions of all its ancestors in the hierarchy of classes and can as well announce new characteristics and overload some of the inherited functions. Inheriting characteristics of the base class, one can make the derived class expand, narrow, change or delete them, or leave unchanged.

Inheritance allows to use repeatedly the code of the base class in the instances of the derived class.

It is recommended to assign the order of the enumeration of sections in such a way, that it should answer the growing of privileges of the access of visibility fields of the elements, located in them: from the most limited to – to the most accessible.

Syntax of the derived class announcement (inheritance):

```
class Name : [Specifier of the access]  Parent
 {
<friendly classes announcement>
<Friendly functions announcement>

[private:]
       <private data-members>
       <private constructors>
       <private functions-members >
 protected:
       <protected data-members>
       <protected constructors>
       <protected functions-members>
 public:
       <generally accessed data-members>
       <generally accessed constructors>
       <generally accessed destructor>
       <generally accessed functions-members>
};
```

`Parent` – base class, **`Name`** – class-inheritor of the base class (derived class).

While creating the instances of the derived classes at first the constructors of the base classes are called, and then–constructors of the derived classes (in the succession, in which classes are located in the hierarchy of inheritance).

While pointing out the memory for the instance **`Name`** the field of the memory for the instance **`Parent`** is added.

If the class is derived from the base one, and nonobligatory access-specifier is omitted, then all its names in the derived class automatically become **`private`** by **`default`**. But it is possible to change by means of access-specifier of the base class.

Access-specifier **`private`** denotes, that derived (that is protected and accessed) names of the base class become inaccessible in the instances of the derived class

Access-specifier **`public`** confirms that accessed names of the base class and its predecessors will be accessed in the instances of the derived class too, and all protected names will stay protected.

And, finally, access-specifier protected makes all derived (protected and accessed) names of the base class protected in the instances of the derived class.

It is possible to derive classes, which expand the possibilities of the base class. For this one can redefine in the derived class that function of the base class which is to be changed. In the same way one can derive classes, limiting the possibilities of the base class.

So, `i`nheritance makes it possible to exclude from the program the repeated fragments of the code and simplify the modification of the program.

Besides, inheritance is the only possibility of implementing classes for the further elaboration, the outlet code of which is inaccessible, and only the the special syntax for the access to the protected and generally accessed sections are allowed.

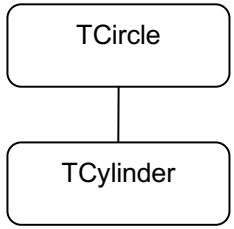
3.4. Hierarchy of classes inheritance Unary inheritance

While building the new class there is a possibility to inherit one or several base classes. Inheritance can be unary and multiple; direct (direction from the indicated classes) and through the other classes. Each from the indicated classes, in its turn, can be derived from the other classes, base for it. In such a way a new complex hierarchical structure can be built. It will reflect the ways of classes inheritance. Such a structure is called *hierarchy of classes inheritance*.

Consider at first the example of base and derived classes elaboration in the simplest case, when class hierarchy of only two simple geometrical objects: circle and cylinder (unary inheritance) has been announced.

In the classes the interior values of the variables **`r`** (radius of the circle) and **`h`** (height of the cylinder) assign the parameters of the created objects. The base class **`TCircle`** (cylinder with the height **`0`**) denotes the circle, and the derived **`TCylinder`** – cylinder.

Consider the classes' hierarchy.



Ancestor class
(parent class)

Inheritor class
(derived class)

The program illustrates the direct unary open inheritance (class **TCylinder** is inherited from the class **TCircle**).

Example 3.4.1:

```
#include <math.h>
#include <iostream.h>
#include <windows.h>

// class predecessor
class TCircle The
{ protected:
       double r ;
  public:
       TCircle (double R = 0) {r = R;}
       void setRadius(double R) {r = R;}
       double getRadius() {return r; }
       double area() { return M_PI*r*r; }
       void show();
};

// inheritor class
class TCylinder : public TCircle
 {
  private:
        double h;
  public:
// constructor

    TCylinder(double H = 0, double R = 0):
                                  TCircle(R)
                             {h = H;}
//to establish the height of the cylinder
    void setHeight(double H) { h = H;}
//to get the height of the cylinder
    double getHeight() { return h;}
//to get the area of cylinder surface
    double area(){return 2*TCircle::area()+2*M_PI*r*h;}
//to get the volume of the cylinder surface
     double volume() { return TCircle::area()*h; }
```

```
//to put the information about the cylinder on the screen
     void show();
  };

void TCircle::show()
{
cout << "\nRadius of the circle = " << getRadius() << endl
<< "The area of the circle = " << area() << endl << endl;
}

void TCylinder::show()
{
cout << "\nRadius of the base= " << getRadius()
<< "\t\tHeight of the cylinder = " << getHeight() << endl<<
"\nArea of the surface = " << area()
<< "\nVolume of the surface = " << volume() << endl;
}

void main()
{
// creation of objects by the corresponding constructors
TCircle circle(3) ;
TCylinder cylinder(5, 1);

// outlet of the information about the circle
circle.show();

// outlet of the information about the cylinder
cylinder.show();

// outlet of the information about the base of the cylinder
cylinder.TCircle::show();

// delay of the time
cin.get();
}
```

Announcement of the class **TCircle** contains the only data member **r**, constructor and some of function members of the class. While creating the object constructor initializes the data member **r** by the initial value of the circle radius.

The access functions are set by **setRadius**(), and **getRadius**() returns the value of the data-member **r**. The method (function-member) **area**() returns the area of the circle. The method **show**() lets out to the screen the value of the radius and the area of the circle.

The class **TCylinder**, announced as derived from **TCircle**, contains the only data member **h**, constructor and some of function-member of the class. Constructor **TCylinder** clearly calls the base constructor for the initialization of the radius. This class is inherited by the data-member **r** – the radius of the cylinder base and member-

functions **setRadius**() and **getRadius**(). By the creation of the object the constructor initializes the data-members **r** and **h** by the initial values. In the constructor the data-member **h** is initialized by the value of the argument, and the data-member **r** gets the value by means of the call of the constructor of the base class **TCircle** with the argument **R**.

The function **setHeight**() sets, a **getHeight**() returns the values of the data-member **h** (it is the access- function).

The method **TCircle**::**area**() overloads the inherited function of the base class in order to return the area of the cylinder surface. The function **volume**() calculates the cylinder volume, and the function **show**() prints the radius of the base and height values and the surface area of cylinder.

The function **main**() creates the circle **circle** of the class **TCircle** with the radius 2 and the cylinder **cylinder** of the class **TCylinder** with the height 5 and the radius 1, and then returns to **show**() for the print of the created objects parameters. As it is evident from the program the object-inheritor **cylinder** can address both its methods and the method of the base class, even if these methods are closed by their own methods of the same name. Such methods can be called with the help of the scope operator (**TCircle**::).

Result of the program work:

```
Radius of the circle = 3
Area of the circle = 28.2743

Radius of the base = 1          Height of the cylinder = 5
Area of the surface = 376991
Volume of the cylinder = 15.708


Radius of the circle = 1
Area of the circle = 3.14159
```

Remarks:

- it is possible to address the open members of the class in the same way, as an ordinary member of the derived class;
- while creating the objects of the derived class at first constructors of the base class are called, and then constructor, defined in the derived class. When the object is deleted, destructors are called in the opposite order;

- in order to convey the arguments to the constructors of the base classes, the initializer list is implemented.

3.5. Transformation of the pointers on the objects from one hierarchy of classes

Consider that the derived class is created from the base one with the help of the open inheritance. Then implicit transformation of the types for the pointers on the objects from one hierarchy takes place, in case these transformations are directed up (from the son to the father) Transformations down (from the father to the son) must take place explicitly.

Demonstrate such transformations for the hierarchy of classes from the section 3.4. Rewrite the main function **main**(), in which create dynamically two objects of different types from one hierarchy and make two transformations of corresponding pointers on them.

Example 3.5.1:

```
void main()
{

//creation of objects by the corresponding constructors
TCircle* pCircle=new TCircle(3),*p1 ;
TCylinder* pCylinder= new TCylinder(5, 1);
TCylinder* p2;

// the call of related methods through the pointers
pCircle->show();
pCylinder->show();

// the call of related methods through the objects,
// to which the pointers indicate
(*pCircle).show();
(*pCylinder).show();

// the call of the method class TCircle for cylinder
// through the pointer
// non-obvious transformation of the type
p1 = pCylinder;
p1->show();

// the call of the method class Cylinder for the circle //
through the pointer
// obvious transformation of the type
p2 = (TCylinder*)pCircle;
p2->show();

// delay of the time
cin.get();
```

```
}
```

Result of the program work:

```
Radius of the circle = 3
The circle area = 28.2743

The base radius = 1            Height of the cylinder = 5
The surface area 37.6991
Volume of the cylinder = 15.708

Radius of the circle = 3
Area of the circle = 28.2743

Radius of the base = 1         Height of the cylinder = 5
Area of the surface = 37.6991
Volume of the cylinder = 15.708

Radius of the circle = 1
Area of the circle = 3.14159

Radius of the base = 3          Height of the cylinder =
3.16621e-299
Area of the surface = 56.5487
Volume of the cylinder = 8.95224e-298
```

As it is evident from the results of the program work, implicit transformations of the pointers are conducted in the direction from the inheritor to the father, and in the direction from the father to the inheritor the type of the pointer must be indicated with the help of the type transformation operation.

While printing the cylinder height and volume, created by the function **`show`**() for the instance of the class **`TCircle`** we can print the numbers (height and volume), which could be considered zeros. It is understandable, since variable **`h`** for the circle was not designated, it means that its value equals to zero, which agrees with the fact that from the mathematical point of view the circle may be perceived as a cylinder with the zero height.

3.6. Transformation of the types for the objects from one hierarchy of classes

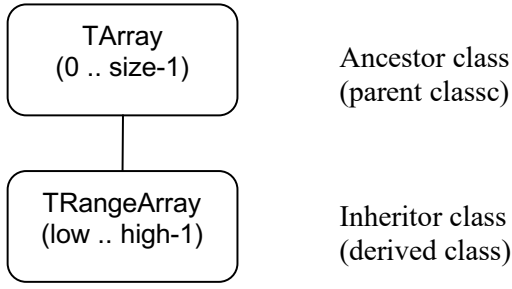
Remark that the derived class can be considered specialized instance of the base class, therefore the transformation of the object of the derived class into the type of the base one, often takes place implicitly. If it is

necessary to implement the reversed transformations of objects, then it is expedient to realize such transformations in the copy constructor.

Therefore further consider the example of more complex inheritance and show the implementation of two different copy constructors in the inheritor. At the expense of these constructors the methods of ancestors' and inheritors' work is ensured without the necessity of writing the whole choice of methods for each of them. The number of methods, written for the derived classes can successfully be implemented for the instances of base classes and the other way round.

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 derived class – **`TRangeArray`** with the limits of the change of elements' index from **`low`** to **`high-1`**. In both classes overload the index operator is overloaded.

Hierarchy of classes:



Example 3.6.1:

```
#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 derived 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
```

3.7. Multiple inheritance

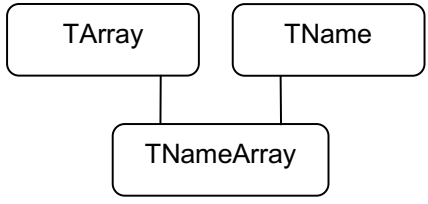
As it was mentioned above, while building the derived class, there is a possibility of inheriting not one class but several. In this case the multiple inheritance is taking place. In order to indicate to what base classes a new class is derived, we introduce the list of base classes (classes-ancestors) in the announcement of the class in front of the list of its members.

Each of the base classes, in its turn, can be derived from the base classes. So, the hierarchy of inheritance can be very complicated.

Consider further the example of elaboration of two base and one derived classes. As one of the base classes we shall implement the simplified variant of **`TArray`** class, considered above in the section 3.4. The other base class **`TName`** – dynamic synonymous string. Their common inheritor **`TNameArray`** – dynamic array with the dynamic name.

While constructing the objects of the type **`TNameArray`** we shall call the constructors of the base classes, but the destructor will be left empty (base classes destructors are called automatically).

Hierarchy of classes:



Ancestors' classes
(parents classes)

Inheritor class
(derived class)

Example 3.7.1:

```
#include <vcl.h>
#include <windows.h>

#include <string.h>
```

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

//the names own space with the types
// TArray, TName, TNameArray
namespace My
{
// base class – dynamic vector
// index - з 0 до size-1)
  class TArray
  {
  protected:
       int  size;
       int* v;

  public:
        TArray(int);              // constructor
        TArray(const TArray&);    // copies generator
        ~TArray() {delete[] v;};  // destructor
        TArray& operator=(const TArray&);
        void print()const;
  };

  TArray::TArray(int n)
  {
       size=n;
       v=new int[n];
       for(int i=0;i<n;i++)
              v[i]=random(10);
  }

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


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

 void TArray::print()const
{
for(int i=0;i<size;i++)
```

```
        cout<<v[i]<<" ";
}

// base class - dynamic string with the name
class TName
{
protected:
        char* s;
public:
        TName(char* str){s=strdup(str);}
        TName(const TName&);
        ~TName(){delete[] s;}
        TName& operator=(const TName&);
        void print()const;
};

void TName::print()const
{
cout<<s<<endl;
}

TName::TName(const TName& a)
  {
   s=new char[strlen(a.s)+1];
   strcpy(s,a.s);
   }

TName& TName::operator=(const TName& a)
  {
         if (this != &a) // self assignment control
         {
         delete[] s;
         s=new char[strlen(a.s)+1];
         strcpy(s,a.s);
         }
         return *this;
  }

 // multiple inheritance //inheritor-class (array with the
name)
  class TNameArray: public TArray, public TName
  {
         public:
         TNameArray(int, int*, char*);    // constructor
         TNameArray(TArray, TName);       // constructor
         TNameArray(const TNameArray&);  // copies generator
        ~TNameArray() {};               // destructor
         void print()const;
  };

  void TNameArray::print()const
{
```

```
cout<<"\nArray with the name: ";
TName::print(); //visibility field of the base class TName
TArray::print();//visibility field of the base class TArray
}

TNameArray::TNameArray(const TNameArray& a):
                          TArray(a), TName(a)
{
}

  TNameArray::TNameArray(int n, int * mas, char* str ):
                          TArray(n), TName(str)
  {
       for(int i=0;i<size; i++)
              v[i]=mas[i];
  }

 TNameArray::TNameArray(TArray mas, TName name):
                          TArray(mas), TName(name)
  {}

int main(int argc, _TCHAR* argv[])
{

TArray a(3),b(3),c(a), d(3);

cout<<"\nNumber arrays\n";
cout<<"\na: "; a.print();
cout<<"\nb: "; b.print();
cout<<"\nc: "; c.print();

d=a;
cout<<"\nd=a: "; d.print();


// the call of the constructors with the parameter
TName s1("abcabc6"),s2("123"),s4("NewRow");

cout<<"\n\nLines\n";
// the call of the copy constructor
TName s3=s2;
s3.print();

// the call of the assignment operator
//(chaining assignment)
s2=s3=s1;

// print of the results
s1.print();
s2.print();
s3.print();
s4.print();
```

```
int v[]={5,4,3,2,1,6,7,8,9,10};
// the call of the constructor with the parameter
TNameArray ns1(4,v,"SmallRow"),ns2(6,v+3,"BigRow"),
          ns3(5,v+2,"Row"), ns4(5,v+2,"Row");

// the call of the copy constructor
TNameArray ns5(ns1);

cout<<"\nNumber arrays with the names\n";
ns1.print();
ns2.print();
ns3.print();
ns5.print();

//the call of the assignment operators
// of ancestors' classes
ns4=ns2;
ns2.print();

// obvious delivery of the parents' classes instances
// to the constructor


TNameArray ns6(a,ns3);
ns6.print();

cin.get();
return 0;
}

};
```

Result of the program work:

```
Number arrays
a: 7 4 7
b: 7 6 0
c: 7 4 7
d=a: 7 4 7

Lines
123
abcabc6
abcabc6
abcabc6
NewRow

Number arrays with the names

Array with the name: SmallRow
5 4 R3 2
```

```
Array with the name: BigRow
2 1 6 7 8 9
Array with the name: Row
3 2 1 6 7
Array with the name: SmallRow
5 4 3 2
Array with the name: BigRow
2 1 6 7 8 9
Array with the name: Row
7 4 7
```

Remark that while implementing multiple inheritance, secondary effects and errors, connected with the usage of the same names in different ancestors may often occur. In order to decrease the number of such errors one can implement the scope operator in which the base class visibility field is specified. It is done so in the body of **`print`**() function of the inheritor-class **`TNameArray`**.

In more complicated situations (rhombus like hierarchy) this problem is solved by means of virtual inheritance, considered in the section 4.6.

Questions for self control

1. What is aggregation? What are the types of aggregation?
2. What examples of aggregation can you give out?
3. Is it possible to implement the pointers at other classes as class-members?
4. Why are templates of classes necessary?
5. What key words can be used to define some generalized type in the class template?
6. What type parameters can be used while describing the template?
7. How many parameters can there be in the template?
8. What is the concretizing of template?
9. Why is the inheritance necessary?
10. What specifiers are implemented while inheriting classes?
11. How many base classes can there be in the derived class?
12. How do access specifies interact with the names of the base classes sections?
13. Why is the section **`protected`** necessary?
14. Why is it not recommended to announce all base classes sections as **`public`**?
15. When is the multiple inheritance needed?
16. What problems can arise with multiple inheritance?
17. How are usually called base classes constructors in the derived

classes constructors?

18. What can be copy constructors types in the derived classes?
19. What transformations of the pointers to the objects from one hierarchy of classes take place non-obviously and which ones are necessary to transform obviously?
20. Is the assignment operator inherited?

Tasks for independent work

1. Inherit the class **`TCylinder`** from the hierarchy of classes section 3.4 and create the class **`TNameCylinder`** "cylinder with the name".
2. Inherit the classes **`TName`** from the hierarchy of classes section 3.7 and **`TCircle`** from the hierarchy of classes ex. 3.4 and create the class **`TNameCircle`** "circle with the name" by using multiple inheritance.
3. Add the main function from the example 3.6.1 by the dynamic creation of objects from the hierarchy of class and the call of corresponding methods through the pointers.
4. Elaborate the class **`TCylinder`** from the example 3.4.1 not by the inheritance method ,but using the aggregation by the pointer to the object of **`TCircle`** type.
5. Elaborate the classc **`TNameArray`** from the example 3.7.1 not by the multiple inheritance method, but using the aggregation by the value (the objects of the type **`TArray`** i **`TName`**).

,

4. Elaboration of family classes. Abstraction

4.1. Static and dynamic polymorphism

Polymorphism – is the last of from three main conceptions on which object-oriented programming is based.

Translated from Greek it means many forms. As to OOP, this term denotes object ability to react to some query (that is the call of the member-function) corresponding to its type, even if on the stage of compilation the type of the object is not yet known.

In `C`++ polymorphism is kept up by two means.

First, during compilation it is sustained by means of overloading functions and operators. Such a type of polymorphism is called *static*, since it is realized before the program is executed by way of the early binding of the identifier functions with the physical addresses on the stage of compilation.

Secondly, while carrying out the program it is sustained with the help of virtual functions. Meeting in the code of the program the call of virtual function, the compiler only marks this call, leaving the function identifier binding with its address up to the final stage. Such a process is called the *late binding*.

Virtual function – is the function, the call of which (and the actions, carried out at the same time) depend on the type of the object for which it is called. The object denotes what function it is necessary to call already during the carrying out of the program. This type of polymorphism is called *dynamic polymorphism*.

If it is planned in the program to work simultaneously with the different types of the hierarchy objects or to add new objects to the hierarchy, the method is announced as virtual by means of the key word **`virtual`**. Virtual methods of hierarchy with one and the same name must have the identical list of arguments.

The class that contains at least one virtual function is called *polymorphic*.

4.2. Virtual functions

Further make a point of a number of important facts, essential for the implementation of polymorphic classes.

- If in some class there is a function, described **`virtual`**, then the compiler adds to such a class the hidden data-member – a

pointer to the table of virtual functions. And a special code is as well will be generated. This code enables to implement the choice of virtual function for this object during the *execution* and not during the compilation.

- If the virtual function is called, then the code, generated by the compiler, finds the pointer to the table of virtual functions, then examines this function and finds the address of the corresponding function. Then one implements the transformation to the appointed address and the call of the function. As it was mentioned above, while creating the object of the derived class, at first the base class constructor is called It is at this stage that the table of virtual functions and the pointer to it is created. After the call of the derived class constructor the pointer to the table of virtual functions is tuned in so, that now it is referred to the redefined version, if it exists.
- If some function is announced in the base class, as virtual, then the function with the same name, the list of parameters and the type of value which returns, automatically becomes in the derived class virtual (the word **`virtual`** may not be repeated).
- If the virtual function was not redefined in the derived class, then when it is called for the object of the derived class, one will call the corresponding function from the base class, nearest by the hierarchy of the base class, where it was defined.
- The function with the name, coinciding with the name of the base class function-member, but with the different list of parameters, may be the derived class member. Then it will be non-virtual ordinary function.

- The constructor of the class can't be virtual, because it is called only at the creation of the object, and the type of this object must be known to the compiler.
- Destructor can be virtual.

4.3. Abstract classes

If while defining the base class some virtual function is announced to be a member of this class, then it must be defined similarly to the ordinary functions or be a pure virtual function, that is, must be announced *abstract* by means of the value = `0`. It means that the function hasn't got the body yet, but its interface is known.

```
class TBase
```

```
{
...
virtual void f(int) = 0; //abstract function
...
};
```

Class that contains at least one abstract function, is called an *abstract class*. It can be only base class for other classes. It is impossible to create the object of this class. In the derived classes, the abstract function must be defined or again announced abstract. If the class, derived from the abstract doesn't denote all pure virtual functions, it remains abstract.

The abstract class can't be used as a function parameter or as the type that returns a value. It can't be used when the types are given explicitly. But it is allowed to define the reference and pointers to the abstract classes.

Example 4.3.1:

Task:

- create the abstract base class **TNumber** (the number, written in the string of symbols) with the pure virtual arithmetic operation "+=" and the function **sum**.
- 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 number of digits of one number string.

Realization program:

```
#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
```

4.4. Projecting and building the class library with virtual functions

Consider the process of creating classes for a graphic library. The classes have to manipulate with ~~the~~ figures, rotating around the given point.

In the projecting process, we are confronted with the task of elaborating class hierarchy. It is obvious that classes must correspond to the modeled objects as close, as possible .Therefore, notions, able to be used as classes, are pointed out in the object field in common cases. Besides classes from the derived field, classes, connected with the environment appear as a rule. We shall create the project in the environment **C++Builder** and implement the component class **TForm1** (class inheritor of the **TForm**) to work with the form and its events handler.

Define the operations on classes, which later would become class methods. In the general case, they can be distributed into groups:

- connected with the objects construction and copying;
- for support connections between the classes, existing in the derived field;
- able to introduce work with the objects in the suitable way.

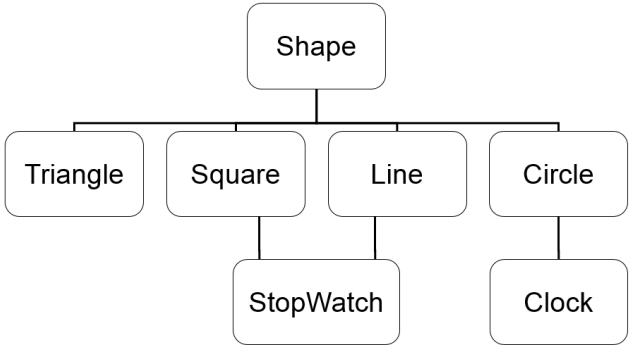
Then one defines functions, which will be virtual.

The process of creating the classes' hierarchy is iteration one. For example, one can point out the main part in two classes, transfer it into the base class, and make the necessary classes to be its derived ones.

Taking into the account the above mentioned thing, let us insert the abstract class **Shape** into the root of the hierarchy, in which define the characteristics, common for the future family. We separately locate the center of future figure rotation and its color in the protected section.

In the open section we announce two real and two abstract functions. The real functions **where**() and **move**() will correspondingly define and transfer the center of figure rotation. As to the abstract functions **draw**() i **rotate**(), they only assign the functions prototypes of painting and rotation which will be realized in the inheritors.

Below the possible hierarchy of classes inheritance is given.



First, demonstrate the implementation of such real classes:

- **`Line`** (clock hand);
- **`Circle`** (face);
- **`Clock`** (watch with two hands).

Note that in the class **`Clock`** the inheritance of the class **`Circle`** will be realized for the creation of the face and aggregation of the class **`Line`** for creating clocks hands.

Create the project in the environment **`C++Builder`** for visual demonstration of the work with the objects of the class **`Clock`**, in which realize the handler of the event **`Click`** for the form: paint the separate objects, create polymorphic functions for the manipulation with the objects of different types from the hierarchy of classes and launch clocks with two handles.

With the help of the element control of the type **`Timer`** we synchronize the work of the clocks with the timer work (the beginning of the work of one of the clocks is defined with the system time, and the other one – with twelve a.m.).

Clocks for visibility work in the accelerated regime.

Example 4.4.1:

```
// connection of the class with the form
#include "Abstract.h"

#include <vcl.h>
#include <math.h>

// the description of the pointer to the form
TForm1 *Form1;

// abstract class (figure)
class Shape
{
```

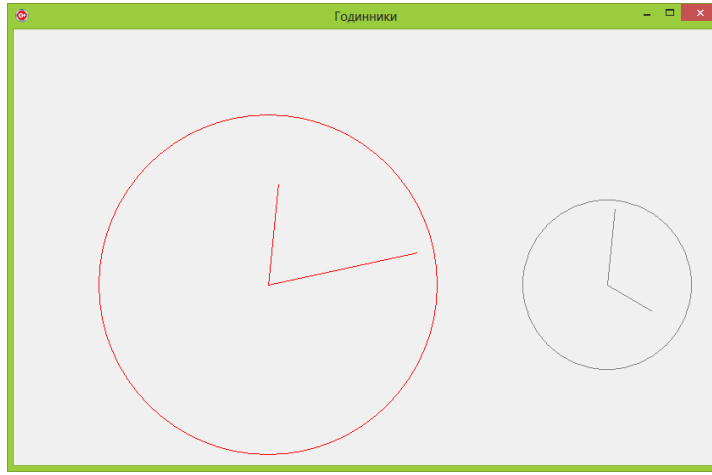
```
protected:
  TPoint center; // the type, defined in the library vcl
  TColor col; // the type, defined in the library vcl

public:
  TPoint where () {return center;}
  void move (TPoint newCenter)
                    {
                    center= newCenter;
                    draw ();
                    }

// abstract functions
    virtual void draw ()=0;
    virtual void rotate (int)=0 ;
};

// real class (circle)
class Circle: public Shape
{
      int radius;
public:
        Circle(){radius=100; col= clDkGray;
                  center.x=700; center.y=300;};
        Circle(int r, TPoint p, TColor c)
                {radius=r; center=p; col=c;}
        void draw ();
        void rotate (int) {} // empty function
};

// real class (segment)
class Line: public Shape
{
        int alpha;
        int lenght;
public:

         Line(){center= TPoint(700,300);
                alpha=0; lenght=90; col=clDkGray;}
         Line(TPoint cen, int a, int len, TColor c)
               {center=cen; alpha=a; lenght=len; col=c;}
         void draw ();
         void rotate (int a){alpha+=a;};
         friend class Clock;

};

__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{}
```

```
TPoint operator+(TPoint p,int r)
{
 p.x+=r;
 p.y+=r;
 return p;
}

void Circle::draw()
{
TPoint p1,p2;
p1=center+(-radius);
p2=center+radius;
Form1->Canvas->Brush->Style=bsClear;
Form1->Canvas->Pen->Color=col;
Form1->Canvas->Ellipse(TRect(p1,p2)) ;
}

void Line::draw()
{
TPoint p;
float fi=alpha*M_PI/180;
 p.x=center.x+sin(fi)*lenght;
 p.y=center.y-cos(fi)*lenght;

Form1->Canvas->Pen->Color=col;
Form1->Canvas->MoveTo(center.x,center.y) ;
Form1->Canvas->LineTo(p.x,p.y) ;
}

// real class (watch)
 class Clock : public Circle // inheritance
{
        Line hour, minute; //aggregation
public:
        Clock(){set_now();};
        Clock(int radius,TPoint cen, TColor color):
                          Circle( radius,cen,color)
            {
             minute=Line(cen, 0, radius*0.9, color);
             hour=Line(cen, 0, radius*0.6, color);
            }
// painting clock
         void draw () ;
// shift of the minute hand
        void rotate(int fi){minute.rotate(fi) ;}
// set time by parameters
         void set (int fi, int psi)
              {
              minute.rotate(fi);
              hour.rotate(psi);
              }
                                  }
```

```
// set current time
         void set_now();

 };

// painting clock
void Clock::draw ()
{       Circle::draw();// face  (inheritance)
        minute.draw(); // drawing of pointers
        hour.draw();   // aggregating

}

// set current time
void Clock::set_now()
{       Word Hour, Min, Sec, MSec;
        DecodeTime(Now(), Hour, Min, Sec, MSec);
        if (Hour>=12) Hour-=12;
        set( Min*6, Hour*30+Min/2);
        hour.lenght=minute.lenght*0.7;
}

// handler of events Click for the form
void __fastcall TForm1::FormClick(TObject *Sender)
{
TPoint T1(300,400), T2(200,100);
Circle C(100,T1,clRed);
C.draw ();
C.move (T2) ; // the call of the function from the abstract
class
C.draw () ;
Shape* p[2];
p[0] = &C;
p[1]=new Line(T2,25,100,clBlue);
p[1]->draw();      // virtual function
p[1]->rotate(45);  // virtual function

Sleep(5000);
Clock timeNnow, time(200, TPoint(300,300),clRed);
timeNow.draw();    // virtual function
time.draw();
Timer1->Enabled=true;
}

// handler of the events from the timer on the form
// demonstrate in of two clocks work
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
static Clock timeNow,time(200, TPoint(300,300),clRed);
static int flag=0;
timeNow.set(6,flag);
```

```
time.set(6,flag);
flag=!flag;
Form1->Canvas->Brush->Color=Color;
Form1->Canvas->FillRect(TRect(0,0,Form1->Width,
                                  Form1->Height));
timeNow.draw(); // virtual function
time.draw();   // virtual function
}
```

Below the screenshot of the program is given at the moment when two clocks are demonstrated (clocks work in the accelerated regime).



4.5. Polymorphic functions

Functions, manipulating similarly with different types of objects from one hierarchy of classes by means of the mechanisms of virtual functions are called polymorphic.

Such functions by means of virtual functions table define the object type and call the necessary functions during the program implementing.

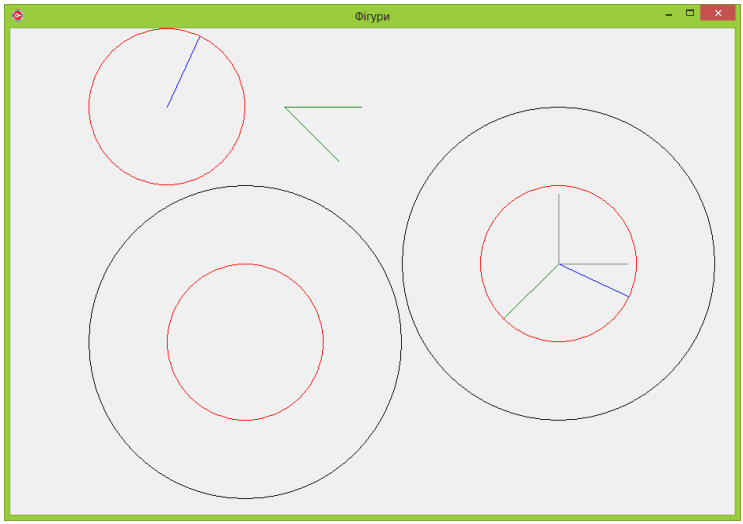
We shall ~~add~~ complete the example given in p. 4.4 with polymorphic functions **`rotate_all`**() and **`move_all`**(), which will correspondingly rotate and transfer all the figures, pointers, on which in the dynamically created area of the pointers type **`Shape**`** are transferred to these functions.

Besides, we shall, demonstrate implementing of dynamically created objects of different types from one hierarchy of classes.

Example 4.5.1:

```
// rotate all the elements of vector v
// parameters size (vector size), alpha (rotation angle)
//polymorphic function

void rotate_all (Shape** v, int size, int alpha)
{
for (int i=0; i<size; i++)
       // called for every object
       //own virtual function
       v[i]->rotate(alpha);
}

// transfer all the elements of vector v in the center p
// polymorphic function

void move_all (Shape** v, int size, TPoint p)
{
for (int i=0; i<size; i++)
       // called for every object
      //own virtual function
       v[i]->move(p) ;
}

// handler of the event Click for the form
void __fastcall TForm1::FormClick(TObject *Sender)
{
TPoint T1(300,400), T2(200,100) ;;
Circle C(100,T1,clRed);
C.draw ();
C.move (T2) ; // the call of the function from abstract
class
C.draw () ;
Shape* p[5];

p[0] = &C;
p[1]=new Circle(200,T1,clBlack);
p[2]=new Line(T2,25,100,clBlue);
p[3]= new Line;
p[4]= new Line(TPoint(350,100),90,100,clGreen);
p[4]->draw();       // virtual function
p[4]->rotate(45);  // virtual function

for(int i=0;i<5;i++)
       p[i]->draw();

rotate_all(p,5,90);  // polymorphic function
move_all(p,5,p[3]->where());  // polymorphic function
}
```

Below the screenshot of the program, demonstrating the work of virtual and polymorphic functions is given.



4.6. Virtual inheritance

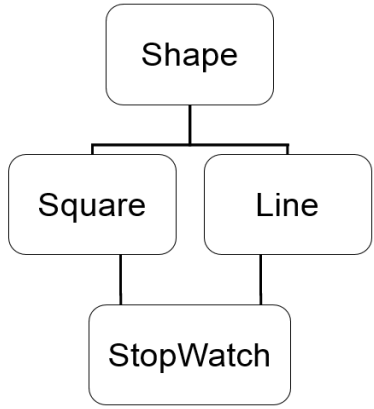
Remark that in section 4.4 while building class **`Clock`** (clock with two hands) class **`Line`** was aggregated into class **`Clock`** twice, and the class **`Circle`** (face) is inherited. If instead of the aggregation to use non-virtual multiple inheritance, then a number of problems, connected with the appearance of the inherited member-classes with identical names, might arise. It is possible to avoid such problems by inheriting the abstract class virtually **`Shape`**. In such a case the inheritors will address the inherited data-members of the class **`Shape`** through implicit pointers which will indicate to one common memory field for the data-members of the same name from the virtually inherited class.

We shall demonstrate such an approach for building class **`StopWatch`** (stopwatch).

We shall implement class **`Line`**, as a type for stopwatch hand, and class **`Square`** (square) as the type for the face.

Remind that classes **`Shape`** and **`Line`** have already been elaborated in section 4.4. It remains to create classes **`Square`** and **`StopWatch`**.

Give a fragment of the hierarchy of classes, having now rhombus like structure.



In the root of the hierarchy, as before, there is an abstract class **Shape** (now inherited virtually), and for building class **StopWatch** we shall implement the multiple inheritance.

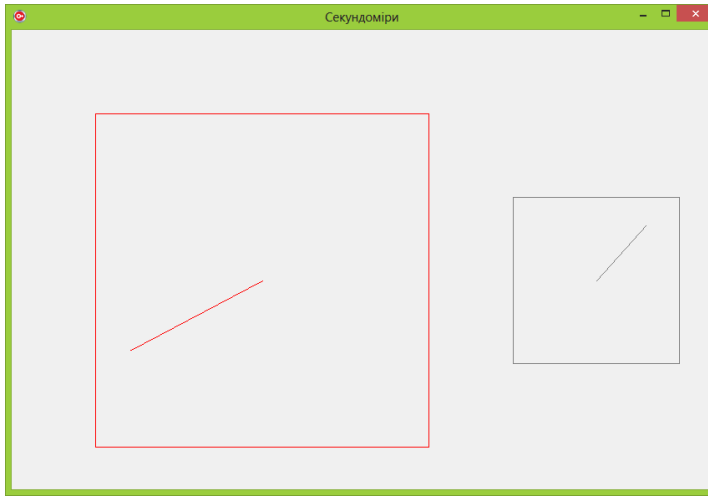
Example 4.6.1:

```
// virtual inheritance (real class – square)
class Square: virtual public Shape
{
     int radius; // radius of inscribed circle
public:
       Square()
              {
              radius=100,
              col= clDkGray;
              center.x=700;
              center.y=300;
              };
       Square(int r, TPoint p, TColor c)
              {
              radius=r;
              center=p;
              col=c;
              }
       void draw ();
       void rotate (int) {} //empty function
};

// painting the face (square)
void Square::draw()
{
TPoint p1,p2;
p1=center+(-radius);
p2=center+radius;
Form1->Canvas->Brush->Style=bsClear;
```

```
Form1->Canvas->Pen->Color=col;
Form1->Canvas->Rectangle(TRect(p1,p2)) ;
}

//multiple inheritance (real class - stopwatch)
class StopWatch : public Square, public Line
{
public:
       StopWatch(){set();};
       StopWatch(int radius,TPoint cen,  TColor color):
               // the list of initializers
               Square(radius,cen,color),
               Line(cen, radius,radius*0.9, color)
              {}

// painting stopwatch
       void draw ();

// shift of the second hand
       void rotate(int fi)
              {
              Line::rotate(fi);
              }

// set the second hand
       void set ()
              {
              Line::rotate(0);
              }
};

// painting the stopwatch
void StopWatch::draw ()
{
 Square::draw(); // painting the face
 Line::draw();   // painting the hand
}

// demonstration  of two stopwatches work
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
static StopWatch secNow,
       sec(200, TPoint(300,300),clRed);
static int flag=0;

secNow.rotate(6);
sec.rotate(6);
flag=!flag;
Form1->Canvas->Brush->Color=Color;
Form1->Canvas->FillRect(TRect(0,0,Form1->Width,
      Form1->Height));
```

```
secNow.draw(); // virtual function
sec.draw();    // virtual function
}
```

Below the screenshot of the program, demonstrating the implementation of virtual multiple inheritance (work of two stopwatches) has been given.



Questions for self-control

1. What is static polymorphism? Give real examples.
2. What is dynamic polymorphism? What functions ensure the mechanism of dynamic polymorphism?
3. What are virtual functions tables? How are they created and implemented?
4. What problems arise and how are they solved if in the classes from one hierarchy appear the members with identical names?
5. What is pure virtual function?
6. What are abstract and real classes?
7. Is it possible to create abstract class objects? And real?
8. Why are abstract classes necessary?
9. Can the derived class be abstract?
10. Is it possible to announce the binary operator as a pure virtual member-function of the abstract base class? And unary operator?
11. Is it possible to implement the type “abstract class” for the parameter transfer into some function? And for the result return?
12. What are the polymorphic functions?
13. When does the necessity to implement virtual inheritance appear?

14. Is it possible to announce the abstract classes if multiple inheritance implementation is planned?
15. What is rhombuslike hierarchy? What problems arise while building such a hierarchy?

Tasks for the independent work

1. Realize class **`Triangle`** from the hierarchy of classes in section 4.4 and create the clock with the triangular face not using the multiple inheritance.
2. Realize class **`Triangle`** from the hierarchy of classes section 4.4 and create the stopwatch with the triangle face, using the multiple virtual inheritance.
3. Add class **`StopWatch`** from section 4.6 by the operator of postfix unary enlargement and diminution the seconds on the stopwatch.
4. Add class **`Clock`** from section 4.4 by the operator of prefix unary enlargement and diminution minutes on the clock.
5. Inherit classes **`Clock`** and **`Square`** from the hierarchy of classes section 4.4 and build class **`FrameClock`** (clock with the frame).

5. Assignments

Assignment № 1

Theme: 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 **input**() 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 $\mathbf{y=-x^2+Ax+B}$. 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 f**unction**() – calculation for the assigned **x** value of the function **y**, **isCrossing** (**ob1**,**ob2**) – definition, or 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** – 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 adding, the integer, nearest to the values **first/second** is received.. Function **check**() must check the equality of the denominator to the zero.

5. 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 **summa**(**ob1**,**ob2**) – calculation of the total money sum of two objects.

6. 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.

7. 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.

8. 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]**.

9. 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 **[Fieldfirst, 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.

10. 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.

Assignment № 2

Theme: 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) [17].

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, in witch the to call creating function.

2. To create the structure, defining the triangle on the surface. To override 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 override 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. Overloading the operations "<<", ">>" for the matrixes input and the output of transposed to them matrixes. Matrixes, rectangular in size **`n×m`** with the integer and real elements. To use the templates.

9. 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.

10. To define the functions:

a & **b** – vector multiplication of vectors;

! **a** – normalization of vector.

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

11. 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.

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

"++" – determination of radius of the circle described round the polygon;

"--" – determination of the radius, inscribed into the polygon circle;

"<" – comparison of two polygons areas.

Assignment № 3

Theme: 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**() або **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 equation **$ax^2+bx+c=0$**;
- print of the parabola coordinates top **$y=ax^2+bx+c$**;
- 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: **TCyrcle** – 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. 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.

6. 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.

7. To define the type **`TTriangle`** for the introduction of the triangle. The data fields must include angles 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 the definition rectangular type (equilateral, **`isosceles or`** rectangular).

8. 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.

9. 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, **st**ring, in the way of "year.month.day" (for example, "2013.08.28") 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, **m**onth, day), comparison of the dates (equal, **to**, past), calculation of the number of days between the dates.

10. 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, adding 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).

Assignment № 4

Theme: 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 **A**(X_1,Y_1), **B**(X_2,Y_2), **C**(X_3,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 (rectangular, 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 polynoms of **n** power with the real coefficients $P(x)=a_0+a_1 x+a_2 x^2+\ldots+a_{n-1} x^{n-1}$, $x\in R$. In order to save the polynoms coefficients to implement the array.

To create constructors and the destructor.

To define the operations:

"+", "-", "*" – addition, subtraction and the multiplication of polynoms:

To define the functions:

- the polynom's print;
- creation of the polynom's copy;
- finding the polynom's value in the point;
- finding the part and the remainder from the division of polynom by polygon ;
- calculation of the polynoms $P'(x)$, $P(Q(x))$, $(x-b)P(x)$, $P(x+b)$ (**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, if 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;
- calculating the first entry into string **st1** 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 save 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 in order of the distance from the points to the beginning of coordinates;
- ordering the set in order of the angle of the turn regarding the axis OX.

6. 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 comparison;
- finding the circle area;
- the arc length for the central circle, assigned in the degrees;
- the circle graph depiction on the screen.

7. To define the class: rectangular 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 rectangular (**x1∈N**, **y1∈N**, **x2∈N**, **y2∈N)**. To create constructors and the destructor.

To define the operations:

"+" – the area of two rectangular union;
"–" – the area of two rectangular difference;
"*" – the area of two rectangular crossing.

To define the functions:

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

8. 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;
- check on the point belonging to the plane;
- defining the coefficients of the normal plane equation;
- check, if the plane is parallel to the to the coordinates of the planes;
- check, if two planes are parallel and perpendicular;
- finding the distance between the point and the plane.

9. 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 salient;
- finding the area of the figure, limited by the salient broken line;
- graph depiction of the broken line.

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

10. 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;
- check 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.

Assignment № 5

Theme: 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: straight 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 derived.

Assignment № 6

Theme: 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 derived real classes in which to realize 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 derived 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 derived 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 derived classes – **`TCylinder, TCone, TTruncatedCone`** (cylinder, cone, conoid) with the own realization of virtual functions.

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

4. To create the abstract base class **TFunction** (function) 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 derived classes **TEllipse** (ellipsis) and **THyperbola** (hyperbola) with the own calculation functions and the print of the curve equations.

5. 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 derived classes **TTime** (running date) with the own realization of the above-mentioned functions.

6. To create the 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.

7. 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 derived classes **THeximal** (the integer number in the hexadecimal form), **TBinary** (the integer number in the octal form) and to overload for them the above-mentioned functions.

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

9. To create the abstract base class **TNorm** with the virtual functions of the normalization and calculation of the vector's length. To define the derived 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.

10. To create the abstract class **TShape** with the data-members real number **r** and the array of the points. To create the derived classes **TCircle** (with the functions, defining the number of the array points **count**() , belonging to the circle of radius **r** and the center at the coordinates beginning and **sort**() the array sorting 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.

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. C++. 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. http://tanet.tneu.org/phocadownloadpap/osnovy_progrsmuvannja_C_C++.pdf
23. http://bookwebmaster.narod.ru/cplusplus.html
24. http://www.intuit.ru/department/pl/ccpp/print.lit.html
25. http://bulletinsite.net/index.php?id1=6&category=programmer&author=pavlovskaya-ta&book=2003
26. http://khpi-iip.mipk.kharkiv.edu/library/pgm/cpp/index.html
27. http://articles.org.ru/docum/builder/3.php
28. http://www.codenet.ru/progr/cpp/ipn.php