STEP computer ACADEMY

PROGRAMMING C

Lesson No. 2

Programming

C

Contents

1. Concept of an operator

In the previous lesson we got acquainted with the concept of variable and data type. Besides, in use cases during the lessons, as well as in home assignments we have exercised some actions with the variables, meaning we operated data. It is quite obvious that the words operator and operate have the same origin, so due to simple logic.

Operator is a language construction allowing exercising various actions with data leading to a particular result.

All operators are usually divided into the groups by признаку их действия. For example, arithmetic operations are operations allowing exercise arithmetic actions with data (adding, deducting, etc.). Later we will talk more about all подобных groups represented within language C. At the moment we should discuss a more extensive classification of operators accepted regardless their influence on variables contents. Thus, all operators are divided into:

1. **Unary** — operators that need one operand only (data they operate). You are familiar with the example of унарного operator from the school mathematics — unary minus allowing converting a number into negative (3 and –3), or positive (–(–3)). I. e. general syntax of unary operator is the following:

```
operator operand;        or        operand operator;
```

2. Binary — operators, which need two operands on the left and right from the operator. You know many of such operators – +, –,*, etc. and their general syntax can be represented in the following form:

```
operand operand operand;
```

3. Ternary operators, which need three operands. There is only one such operator in programming language C and we will get familiar with its syntax later.

Priority

All operators have priority. Below there are operators according to the priorities. We will get acquainted close with some of them in the present lesson, and we will learn others during our further education. Of course there are not all language operators within a table, so far there are the most relevant for us.

Symbolic indication of operations	
The highest priority	The highest priority
() [] . ->	^
! *(ун) – (ун) ~ ++ ––	|
% * /	&&
+ –	||
<< >>	?:
< > <= >=	= += –= *= /= %= &= |= ^= >>= <<=
!= ==	
&	The lowest priority

Now when the basis of knowledge in the field of operators has been established, you can proceed to a more detailed study of the later, the next lesson unit in particular.

2. Digital arithmetic

Well-forgotten old…

Let's start. As we noted before — arithmetic operations — are operations allowing exercise arithmetic operations with data. You have known the majority of them since the childhood; nevertheless let's systematize our knowledge using a table represented below.

Operation name	Indication symbol in language C	Short description. Example.
Addition	+	Adds two values and the result is a sum of operands: 5+18 result 23
Deduction	–	Deducts a value on the right from the value on the left of the operator. Result is — difference of operands: 20–15 result 5
Multiplication	*	Multiply two values and result is product of operands: 5*10 result 50
Division	/	Divides the value on the left by the value on the right of the operator. For example: 20/4 result 5
Division modulo	%	A result of this operation is excess from integer division, for example if we divide 11 by 3, then we have 3 integers, (because 3*3=9), and 2 in excess, this number will be a result a quotient modulo: 11/3 = 3 integers and 2 in excess 11%3 = 2 (excess)

Note:

1.The operation of the modulus can be applied to integer data only. Attempts to violate this rule will result in an error at the compile stage.

2. If a smaller number is divided by a bigger one using%, a smaller number will be in a result. 3%10 = 3

3. It is not allowed to divide in modulo by zero, this leads to incorrect operation of the software at the execution stage.

Increment and decrement.

All of the above described operations are binary, but there are also unary arithmetic operations. There are no such operations in the school course, but in fact they are very simple:

1. Increment — is denoted by construction ++. This operator increases the contents of any variable by one and rewrites its value. For example:

```
int a=8;
cout<<a; // number 8 at the display
a++;
cout<<a; // number 9 at the display
```

2. Decrement — is denoted by construction ––. This operator increases the contents of any variable by one and rewrites its value. For example:

```
int a=8;
cout<<a; // number 8 at the display
a--;
cout<<a; // number 7 at the display
```

Simple enough, isn't it?! Such formulas can be represented in the form: a=a+1 or a=a–1. It should be noted that for neither increment, nor decrement are used for literals, for it is absolutely illogic to do like that 5=5+1. This is an obvious mistake. Though, this is not the end to our acquaintance with increment and decrement. In the previous unit we found out that unary operator syntax may be not only in the following form,

```
operand operator;
```

But also in the following form

```
operator operand;
```

Such record formats are called postfix, (operator is situated after the value) and prefix (operator is situated before the value). Both increment and decrement have both forms. Let's find out what the differences between the formats are, and when these differences matter.

Example 1.

```
int a=8;
cout<<a; // number 8 at a display
a++;
cout<<a; // number 9 at a display
++a;
cout<<a; // number 10 at a display
```

In this example, there is no difference between the prefix and postfix form. Both in the first and in the second case the variable value just increased by one. The reason for using different forms of operator appears only when there is any other command in a string except an operator.

Example 2.

```
int a=8;
cout<<++a; // number 8 at a display
cout<<a++; // number 8 at a display
cout<<a; // number 10 at a display
```

OPERATOR	DIRECTION
() [] . ->	FROM THE LEFT TO THE RIGHT
* / % + –	
<< >> & ^ |	
<< = >> = ==! =	
&& ||	
Unary – binary + ! ++ ––	FROM THE RIGHT TO THE LEFT
?:	
+ = – = / = * = % = & = ^ = | =	

Now in details about an example:

- Initially variable value is equal to 8.
- Command cout<<++a; contains operator prefix form — increment, consequently using the third rule described above, we increase a variable value by one, and then display it using the command cout<<.
- Command cout<<a++; contains operator postfix form — increment, consequently using the second rule described above, first we display variable value (still 9) using the command cout<<, and then increase variable value by one.
- Upon execution of the next command cout<<a; a changed (increased) value will be displayed, that is number 10.

Proceeding from the previous topics of the lesson unit, we already know how to simplify an inconvenient and «bulky» record of type x=x+1 or x=x–1, converting it into x++, or x––. Therefore, we can increase and decrease variable value only by one, but what to do with any other number? For example, how to simplify the record:

X=X+12;

In this case there is also a simple solution — to use a so called combined operators or contracted arithmetic forms. They look like:

Form name	Combination	Standard record	Contraction
Assignment with multiplication	* =	A=A*N	A*=N
Assignment with division	/ =	A=A/N	A/=N
Assignment with division in modulo	% =	A=A%N	A%=N
Assignment with deduction	– =	A=A–N	A– =N
Assignment with adding	+ =	A=A+N	A+=N

We would recommend you in future to use contracted forms, for it is not only a good ton in the programming, but also increases a readability of software code. Besides, some resources mention that computer process the contracted forms faster, exceeding the rate of program execution. Now it is time to confirm the above said on practice, because a picture is worth a thousand words. You already know how to create projects and add file therein, and this is all you have to do now. There are several programs represented further, which you have to type in order to see practical application of arithmetic operations. Let's start from the project called **Game**.

3. Application of arithmetic operations

Example №1. Game.

```
// primitive game for kids
#include <iostream>
using namespace std;
void main()
{
    int buddies; // number of pirates before the battle
    int afterBattle; //number of pirates after the battle

    cout<<"You the pirate. How many the person in your command, without you?\n\n";
    cin>>buddies;

    cout<<"Suddenly you are attacked by 10 musketeers \n\n";

    cout<<"10 musketeers and 10 pirates perish in fight.\n\n";

    afterBattle=1+buddies-10;

    cout<<»Remains only "<<afterBattle<<» pirates\n\n";

        cout<<"The condition killed totals 107 gold coins \n\n";

    cout<<"It on "<<(107/afterBattle)<<"coins on everyone";

    cout<<"Pirates arrange greater fight because of remained\n\n";
    cout<<(107%afterBattle)<<"coins \n\n";
}
```

In this example we use a rule of division an integer by integer — upon this division fractional part, if any, is cut. In more details we will talk about it in the lesson unit «Types conversion». In the formula (107/afterBattle) — we will find out how many coins each pirate gets if divide them equally. Besides, operator of division by modulo helps us to find out how many undividable coins are left, in other words we get

the remainder of division 107 per number of survived pirates. These are all peculiarities of an example.

Example №2. Circle.

This example demonstrates use of arithmetic operators in the programs exercising mathematical calculations. Name of the project is **Circle**.

We will make sure that a value of arithmetic operators allows solving simple problems. Though, it is not enough to use operators, it is necessary to understand what the result of their use is. And we will talk about it in the next unit.

```
// software for finding circle parameters
#include <iostream>
using namespace std;
void main()
{
      const float PI=3.141592;//denotation of constant — Pi character

      //declaration of variables for storing the parameters
      float radius, circumference, area;

      // invitation to pit in a radius
      cout<<"Welcome to program of work with rounds\n\n";
      cout<<"Put the radius from rounds\n\n";
      cin>>radius;
      cout<<"\n\n";

      area=PI*radius*radius; // calculation of a circle area
      circumference=PI*(radius*2); // calculation of circle length

      // display of the results
      cout<<"Square of round: "<<area<<"\n\n";
      cout<<"length of round: "<<circumference<<"\n\n";
      cout<<"THANKS!!!  BYE!!!\n\n";

}
```

4. Types conversion

When we do something it is obvious that we need to know what result we reach. It is also obvious that ingredients for a soup will be hardly appropriate for a cake with wiped cream. Consequently, the result directly depends on its integral parts. And it is the same with variables. If we add two numbers of int type, it is clear that the result will be of int type too. And what to do if we have the data of different types? And we are going to talk about it in the present unit of the lesson.

Thus, first of all we need to find out what data types can interact with each other. There is a so called type hierarchy, where all data types are arranged in order of precedencey.in order to understand types conversion we need to remember a hierarchy order.

```
bool, char, short-int-unsigned int-long-unsigned
long-float-double-long double
```

In spite of the fact that some types are of the same value, they hold a various range of the values, for example, unsigned int compared to int can house two times more positive values, and that is why it is superior within the hierarchy, whereby both types have a size of 4 bytes. We also should note an important peculiarity represented here, if such types as v take part in types conversion, then they automatically converted to int type.

Now let's study various conversion classifications.

Classification by the range of contained values.

All conversions can be divided into two groups regarding the place of the hierarchy types taking part in conversion.

1. Contracting conversion This conversion type implies that bigger data volumes within the hierarchy is converted to the smaller one, certainly some data may be lost and that is why it is important to with the contracting conversion. For example:

```
int A=23.5;
cout<<A; // on a screen 23
```

2. Widening conversion. This conversion type leads to a so called widening of data type from smaller to bigger value range. Here is a situation as an example.

```
unsigned int a=3000000000;
cout<<a; // on a screen  3000000000
```

In the case 3000000000 is literal of int type, which is easily widened to unsigned int, which allows us to see exactly 3000000000 and not something else. While int type may not comprise such a number.

Classification by the method of conversion exercising.

Regardless of conversion direction it can be exercised by means of two methods.

1. Implicit conversion. All described above examples related to this very conversion type. This conversion type is also called an automatic one, because it runs automatically without programmer interference, in other words we do nothing for it to happen.

```
float A=23,5; - double стал  float без
double became  float without any additional actions
```

2. Explicit conversion. (second name is type casting). In this case a conversion is made by a programmer when it is necessary. Let us study a simple example of the action:

```
double z=37.4;
float y=(int) z;
cout<<z<<"*** "<<y; // on a screen 37.4 ***37
```

(int)z — is explicit contracting conversion of double type to int type. Widening implicit conversion happens within the same string from the obtained int type to float type. We should remember that any conversion has a temporary character and it is valid within the framework of the current string only. In other words, variable z was and remains double during the whole program, and its conversion into int had a temporary character.

Type conversion in expressions.

At last we have reached what we have been talking about in the beginning of the current lesson unit — how to know the type of an expression result. Let us try to calculate it using obtained knowledge. Assume we have the following variables:

```
int I=27;
short S=2;
float F=22.3;
bool B=false;
```

Using these variables we are to make up the following expression:

```
I-F+S*B
```

And in what data type variable the result will be written? It is easy to resolve, if we assume an expression in the form of data types:

```
int-float+short*bool
```

Remember that short and bool will assume int type, and an expression will look as follows:

```
int-float+int*int, where false becomes 0
```

Multiplying int by int gives without any doubts the result of int type. But adding float and int will give float in the result, because a new rule works here:

If there are different data types are used in an expression, the result is to be corrected to the bigger from the types.

An at last deducting float from int type will give float again according to the mentioned rule.

Therefore, the expression result will be of float type.

```
float res= I-F+S*B; // 27-22.3+2*0
cout<<res; // on the screen the number 4.7
```

Now when you are familiar with the rule, you don't have to analyze an expression, it is rather easy to find the biggest type and it will be resultant.

Note: be very attentive also when combining with the same data types. For example, remember if an integer divides by integer, then you will get an integer in the result. Meaning i.e. int A=3; int B=2; cout<<A/B; // on a screen 1, because a result is int and fractional part is lost. cout<<(float)A/B; // on a screen 1.5, because the result is float.

Example that uses types conversion.

Now let us reinforce our knowledge on practice. We are going to create a project and write the following code.

```
#include <iostream>
using namespace std;
void main(){
    // declaration of variables and data input request
    float digit;
    cout<<"Enter digit:";
    cin>>digit;

    /* even if a user inputs a number with the real part,
    The result of the expression will be written in int and the real part will be lost,
    After dividing a number by 100 we are getting a number of hundreds within it. */
    int res=digit/100;
    cout<<res<<" hundred in you digit!!!\n\n";
}
```

Now when we have studied an example, you got the confirmation that using type conversions we can not only arrange a temporary transition from one type to another, but also to resolve a simple logic problem. Consequently, you should regard the topic with decent attention. Understanding of conversion will help you in future to resolve interesting tasks, as well as avoiding unnecessary mistakes.

Uniform initialization

In C++ 11 there was added a mechanism of uniform initialization allowing setting up a value to various programming constructs (variables, arrays, objects) by means of uniform method. Let's review it using variable initializations as an example.

```
int a = {11};  // in a value 11 is written
int b{33};     // in b value 33 is written
```

In order to set up the values to variables we use {}. As we can see from the example it can be done in two way. This initialization form is also called **list-initialization**.

Contraction and list-initialization

What will happen upon code execution?

```
int x = 2.88;
cout<<x; // will be displayed on a screen
```

As you might know from the studied material there is implicit contracting conversion within this example because we assign to x variable of a **double** type value. However, if we use **list-initialization** compiler generates an error at the compile stage, because this initialization form protects from the contraction. It does not allow recording a big size value into the type that does not support such a value range.

```
int x = { 2.88 };  // error at a compile stage. 2.88 — double, and x variable of
integer type
char ch = { 777 };  // error at a compile stage. 777 — int, and variable of
character type
// 777 does not get into a value range char
```

On the other hand:

```
char ch2 = { 23 };    // everything is correct. 23 gets into a range char
double x = { 333 };  // everything is correct 333 - int and gets into a range
double
```

If you want to reveal potential data loss problems with at the compile stage you can use list-initialization.

5. Logical operations

In programming we not only do calculations, but also compare values. For this purpose we use so called logical operations. The result of any logical operations is always a value true or value false. Logical operations are divided into three subgroups:

1. Comparison operators
2. Equality operators
3. Logical union operators and negative inversion.

Now let's study each operators group in more details.

Comparison operations.

They are used when we need to find out how two values relate each other.

Symbol denoting an operator	Assertions
<	Left opernad is less than the right one
>	Left operand is bigger than the right one
<=	Left operand is less or equal to the right one
>=	Left operand is bigger or equal to the right one

The point of comparison operations (second name is relational operations) is that if an assertion set up using an operator is true, then an expression, which contains it will be changed for the value true, if it is false then it will be changed for the false. For example:

```
cout<<(5>3); // will be displayed on a screen, because an assertion (5>3) is true.
cout<<(3<2); //0 will be displayed on a screen, because (3<2) is false.
```

Note: instead of false and true values, 0 and 1 values will be displayed, because they are equivalent to the values false and true. In language C any other number except 0 and 1 can stand for the true value, it can be either positive or negative.

They are used to verify full conformity or non-conformity of two values.

They are used to verify full conformity or non-conformity of two values.

Symbol denoting an operator	Assertions
==	Left operand is equal to right operand
!=	Left operand is not equal to right operand

Application of these two operators correspond the principle of application of the previous group, i.e. at the expression output an expression is substituted either by true or false depending on the assertion.

```
cout<<(5!=3); // на экране будет единица,
так как утверждение (5!=3) истина.
cout<<(3==2); //на экране будет 0, так как (3==2) ложь.
```

Logic operation of объединения and negative inversion.

In majority of cases we can't do with only one assertion. Very often we have to combine assertions this way or another. For example, in order to check if a number is within the range from 1 to 10, it is necessary to check two assertions: a number should be at the same time > = 1 and <= 10. In order to realize this combination it is necessary to input additional operators.

Operator	Name
&&	and
||	or
!=	not

Logical AND (&&)

Logical AND unites two assertion and returns true only in case if the right and left assertions are true. If at least one or both assertions are false then a united assertion is replaced to false. Logical AND works on the basis of a contracted scheme, meaning if the first assertion is false, then the second one is not verified.

Assirtion 1	Assirtion 2	Assirtion1&&Assirtion 2
true	true	true
true	false	false
false	true	false
false	false	false

Now let's review an example where the program receives a number and defines whether this number gets within the range from 1 to 10.

```
#include <iostream>
using namespace std;
void main()
{
      int N;
      cout<<"Enter digit:\n";
      cin>>N;
      cout<<((N>=1)&&(N<=10));
      cout<<"\n\nIf you see 1 your digit is in diapazone\n\n";
      cout<<"\n\nIf you see 0 your digit is not in diapazone\n\n";
}
```

In this example if both assertions are true then 1 is placed to the expression position, otherwise — 0. Consequently a user can analyze the current situation by means of the program constructions.

LOGICAL IF (||)

Logical IF unites two assertions and returns true only in case if at least one of the assertions is true and there is false only in case both assertions are false. Logical OR works on the basis of the contracted scheme, meaning if the first assertion is true, then the second one is not verified.

Assirtion 1	Assirtion 2	Assirtion1&&Assirtion 2
true	true	true
true	false	true
false	true	true
false	false	false

Let's review one more time an example where a program receives a number and defines whether this number is in the range from 1 to 10. But now we use OR.

```
#include <iostream>
using namespace std;
void main()
{
      int N;
      cout<<"Enter digit:\n";
      cin>>N;
      cout<<((N<1)||(N>10));
      cout<<"\n\nIf you see 0 your digit is in diapazone\n\n";
      cout<<"\n\nIf you see 1 your digit is not in diapazone\n\n";
}
```

In this example both assertions are false (meaning that the number is not less than 1 and not bigger than 10) to expression place 0 is put, otherwise — 1. Consequently, a user just like in the previous example can analyze a сложившуюся situation and make a conclusion.

Logical NO(!)

Logical NO is an unary operator and that is why it cannot be called a union operator. It is used in case if we need to change the result of assertion verification into the opposite.

Assirtion	! Assirtion
true	false
false	true

```
// 1 is on a screen because (5==3) is false and its inversion is true.
cout<<!(5==3);
// 0 is on a screen because (3!=2) is true and its inversion is false.
cout<<!(3!=2);
```

Logical returns a false assertion if the latter is true and vice versa it returns true if an assertion is false. This operator can be applied for contraction of condition setting. For example, an assertion

```
b==0
```

in a contracted form can be written using inversion:

```
!b
```

Both records give a true in the result, in case in b is equal to zero.

In this unit we have studied all possible logical operations, which allow defining the truth of any assertion. However, the examples described above ate inconvenient for an amateur user, because a program and not a programmer should make the analysis of the results. Besides, depending on the assertion, not only verification result should be displayed, but do some actions and a programmer can't do anything about it. Therefore, due to knowledge about logic operations it is necessary to obtain additional information for realization of one or another action depending on the condition. This is what we are going to talk about in the next lesson unit.

6. Construction of logical choice if

Now we are going to get acquainted with an operator, which will allow converting a common linear program into «intellectual» program. This operator allows checking a truth of an assertion (expression) and depending on the obtained result we can do one or another action. At first let us review a general syntax of the operator:

```
if (assertion or expression)
{
  action  1;

else
{
  action  2;
}
```

Basic principles of operator work if.

1. Any construction containing logical operators or arithmetic expressions can act in a capacity of assertion or expression.

- if(X>Y) — common assertion will be true if X is indeed more than Y

```
int X=10,Y=5;
if(X>Y){ // true
cout<<"Test!!!";// Test on a screen
}
```

- if(A>B&&A<C) — combined assertion consisting of two parts will be true, if two parts are true

```
int A=10,B=5,C=12;
if(A>B&&A<C){ // true
      cout<<"A between B and C";// on a screen A between B and C
}
```

- if(A–B) — arithmetic expression will be true if A is not equal to B, because otherwise (if they are equal) their difference will give zero, and zero is false

```
int A=10,B=15;
if(A-B){ // -5 is true
      cout<<"A != B";// on a screen A != B
}
```

- if(++A) — arithmetic expression will be true if A is not equal to –1, because if A is equal to –1 and increasing by one will give zero in a result, and zero is false

```
int A=0;
if(++A){ // 1 is true
      cout<<"Best test!!";// Best test!! on a screen
}
```

- if(A++) — arithmetic expression will be true if A is not equal to 0, because in this case we use a postfix increment form; at first a condition will be verified and zero will be found out, and then increased by one.

```
int A=0;
if(A++){ // 0 is false
cout<<"Best test!!";// we will not see this phrase because if will not be
executed
}
```

- if(A==Z) — common assertion will be true if A is equal to Z
- if(A=Z) — assignment operation, an assertion will be true if Z is not equal to zero

Note: Typical mistake. Very often instead of verify operation for equality == can be inadvertently indicated assignment operation =, and assertion meaning can be radically changed. This banal mistype can lead to an incorrect program operation. Let's review two seemingly identical examples.

Correct example.

```
#include <iostream>
using namespace std;
void main(){
  int A,B; //declare two variables

    //ask a user to input data within them
    cout<<"Enter first digit:\n";
    cin>>A;
    cout<<"Enter second digit:\n";
    cin>>B;

    if(B==0){ // is B contains zero
      cout<<"You can't divide by zero!!!";// notify about error
    }
    else{ // otherwise
  cout<<"Result A/B="<<A<<"/"<<B<<"="<<A/B;// we output a result of division A by B
    }
    cout<<"\n The end. \n"; // the end
}
```

Erroneous example.

```
#include <iostream>
using namespace std;
void main(){
          int A,B; //declare two variables
          //we ask a user to input data within them
           cout<<"Enter first digit:\n";
           cin>>A;
           cout<<"Enter second digit:\n";
           cin>>B;

            // Equate B to zero and check a condition, it is automatically false
    if(B=0){ // this part will never be executed, because the condition is false
                 cout<<"You can't divide by zero!!!";
           // notify about an error
             }
else{  // this part is always executed, in which A is divided by newly-minted zero
 /* There will an error at the execution stage within the string, because the
computer will attempt to divide the number by zero  */
 cout<<"Result A/B="<<A<<"/"<<B<<"="<<A/B;
              }
              cout<<"\n The end. \n"; // This phrase we will never see.
}
```

2. As you might have noticed if parentheses contents is true we do action 1, which is enclosed in curly brackets of construction if, whereupon action of block 2 else will be ignored.

3. if the contents of the parentheses is false we do action 2, which is enclosed in curly brackets of construction else, whereupon action 1 will be ignored.

4. Construction else is not obligatory. It means that if there is no need to do anything if the statement is false the construction may not be indicated. For example, a program, using division by zero defenses, can be written as follows:

```
#include <iostream>
using namespace std;
void main(){
         int A,B; //declare two variables
         //ask a user to input data within them
         cout<<"Enter first digit:\n";
         cin>>A;
         cout<<"Enter second digit:\n";
         cin>>B;
         if(B!=0){ // if B is not equal to zero
                cout<<"Result A/B="<<A<<"/"<<B<<"="<<A/B; // we make calculations
         }
         // otherwise we do nothing
         cout<<"\nThe end.\n";
}
```

5. If only one command pertains to the block if or else then we can omit curly braces. Let's make the program even simpler using this rule:

```
#include <iostream>
using namespace std;
void main(){
       int A,B; //declare two variables

        //we ask a user to input data within them
        cout<<"Enter first digit:\n";
        cin>>A;
        cout<<"Enter second digit:\n";
        cin>>B;

       if(B!=0) // if B is not equal to zero
           cout<<"Result A/B="<<A<<"/"<<B<<"="<<A/B; // // we make calculations
       // otherwise we do nothing
       cout<<"\nThe end.\n";
}
```

We have just got acquainted with the operator if and discussed its operating principles. Before proceeding to study peculiarities of if and its use cases, we should make a slight detour and review another operator that would be also useful if we need to make a simple condition.

Note: be attentive: operator if and operator else are inseparable! An attempt to write a code string in between them will lead to an error at the compile stage.

Code fragment with an error.

```
....
    if(B==0){ // if B contains zero
        cout<<"You can't divide by zero!!!";// notify about an error
    }
    cout<<"Hello";// Error!!!! Construction break if - else!!!
    else{ // otherwise
              cout<<"Result = "<<A/B;// output a quotient after dividing A by B
    }
            ....
```

Ternary operator.

Some conditions are very primitive. For example, let's take our program for dividing two numbers. It is simple both from action viewpoint and code. There is one code-action string per each operator if and else. This program can be even more simplified by means of using a ternary operator.

In the beginning let's review its syntax:

```
ASSERTION OR EXPRESSION?ACTION1:ACTION2;
```

Operation principle is simple — if an ASSERTION OR EXPRESSION is true, then an ACTION1 is executed, if false then ACTION2 is executed.

Let's review an action of the operator on the basis of the following example:

```
#include <iostream>
using namespace std;
void main(){
    int A,B; //declare two variables

    //ask a user to input data within them
    cout<<"Enter first digit:\n";
    cin>>A;
    cout<<"Enter second digit:\n"
    cin>>B;

    /* In this case if B is equal to zero then the command situated after inquiry
character and the quotient will be displayed. Otherwise, the command situated after
two spot mark will be executed and there will be division by zero error notification
will be displayed.*/
 (B!=0)?cout<<"Result A/B="<<A<<"/!"<<B<<"="<<A/B:cout<<"You can't divide by zero!!!";

    //the end of the program
    cout<<"\n The end. \n";
}
```

The code has become even more optimized, hasn't it? To reinforce the information we just reviewed let' study another more difficult example. The program will verify which of the two numbers input by a user will be bigger and which is smaller.

```
#include <iostream>
using namespace std;
void main(){
    int a,b; //declare two variables

    //ask a user to input data within them
    cout<<"Enter first digit:\n";
    cin>>a;
    cout<<"Enter second digit:\n";
    cin>>b;

    /*If, (b>a), then b will be put to the position of operator ?:,
    otherwise the operator's place will be taken by a,
    thus, the bigger number will be written
    into variable max.*/
    int max=(b>a)?b:a;

    /*If (b<a), then b will be put to the position of operator ?:,
    otherwise the operator's place will be taken by a,
    thus, the bigger number will be written
    into variable min.*/
    int min=(b<a)?b:a;

    // Displaying the result on a screen.
    cout<<"\n Maximum is \n"<<max;
    cout<<"\n Minimum is \n"<<min<<"\n";
}
```

So, let's fully understand the following: if a condition and action depending thereon the quite simple, then we use a ternary operator. If we need a complex construction, then we should obviously use operator if.

7. Ladder if – else if

From the previous lesson unit you have found out about the existence of conditional operators. Now it would be good to get some information about their work features.

Assume that we need to write a program for accounting money discount, depending on the total amount. For example, if a buyer purchases the goods for the amount more than 100 UAH, he gets a 5% discount. For the amount more than 500 UAH — 10%, and at last for the amount more than 1000 UAH — 25%. Application should propose the amount a buyer has to pay if there is a discount. Now we have to find an optimal candidate solution. Project name is **Discount.**

Candidate solution № 1.

```
#include <iostream>
using namespace std;
void main(){
	// a variable is declared for storing an initial amount
	int summa;

	// request for amount input using a keyboard
	cout<<"Enter item of summa:\n";
	cin>>summa;

	if(summa>100){ // if the amount exceeds 100 UAH a discount is 5%
				cout<<"You have 5% discount!!!\n";
		cout<<"You must pay - "<<summa-summa/100*5<<"\n";
	}
	if(summa>500){ // if the amount exceeds 500 UAH a discount is 10%
				cout<<"You have 10% discount!!!\n";
		cout<<"You must pay - "<<summa-summa/100*10<<"\n";
	}
	if(summa>1000){ // if the amount exceeds 1000 UAH a discount is 25%
				cout<<"You have 25% discount!!!\n";
		cout<<"You must pay - "<<summa-summa/100*25<<"\n";
```

```
        }
        else{ // otherwise you don't have a discount
                              cout<<"You have not discount!!!\n";
                            cout<<"You must pay - "<<summa<<"\n";
        }

}
```

At a first glance a programmer-beginner wouldn't notice any discrepancies, but let's analyze a situation when the program works incorrectly. An amount input from the keyboard is equal to 5000. His number exceeds 1000, consequently we have to get a 25% discount. Therefore, the opposite will happen.

1. Each operator if is an independent and does not depend on other if, consequently regardless which of if executes, condition verification will be executed for all operators.

2. At first condition if (summa>100) is verified. 5000 is obviously more than 100, condition is true and body if is executed. We get on a screen the following:

```
You have 5% discount!!!
You must pay - 4750
```

3. Therefore, program does not stop here, afterwards it will analyzes the condition if (summa>500). 5000 is bigger than 500, condition is true again and body of if is executed. We get on a screen the following:

```
You have 10% discount!!!
You must pay - 4500
```

4. And at last the program verifies a condition if (summa>1000), which will be true, because 5000 is more than 1000. And action connected with if is also currently. And we can see on a screen the following:

```
You have 25% discount!!!
You must pay - 3750
```

Therefore, instead of one information record we get three. Such candidate solution is not нерентабельным. Let's try to optimize it. Project name **Discount2**.

Candidate solution № 2.

```
##include <iostream>
using namespace std;
void main(){
        // a variable is declared for storing an initial amount
        int summa;
        // request for input of an amount from a keyboard
        cout<<"Enter item of summa:\n";
        cin>>summa;
        // if an amount in the range of 100 UAH to 500 UAH a discount is 5%
        if(summa>100&&summa<=500){
                cout<<"You have 5% discount!!!\n";
                cout<<"You must pay - "<<summa-summa/100*5<<"\n";
        }
        // if an amount in the range of 500 UAH to 1000 UAH a discount is 5%
        if(summa>500&&summa<=1000){
                cout<<"You have 10% discount!!!\n";
                cout<<"You must pay - "<<summa-summa/100*10<<"\n";
        }
        if(summa>1000){ // if an amount more than 1000 UAH - a discount is 25%
                cout<<"You have 25% discount!!!\n";
                cout<<"You must pay - "<<summa-summa/100*25<<"\n";
        }
        else{ // otherwise there is not discount
                cout<<"You have not discount!!!\n";
                cout<<"You must pay - "<<summa<<"\n";
        }
}
```

At first let's assume that a user input an amount of 5000 UAH.

1. First if condition will be verified (summa>100&&summa<=500). 5000 in not within the range, so the condition is false and body of if will not be executed.

2. Next a condition if will be analyzed (summa>500&&summa<=1000). 5000 is not within the range, so the condition is false again and body of if will not be executed.

3. ИAnd at last the program will check condition if (summa>1000), which will appear true, because 5000 is more than 1000. And action connected with if will be executed. We can see on a screen the following:

```
You have 25% discount!!!
You must pay - 3750
```

It could seem the end of AN operation, but let's check one more variant. For example a user inputs a value of 600. The following data will be displayed on a screen:

```
Enter item of summa:
600
You have 10% discount!!!
You must pay - 540
You have not discount!!!
You must pay - 600
Press any key to continue
```

Such turn of event is easy to explain:

1. At first condition if is verified (summa>100&&summa<=500). 5000 in not within the set up range, so the condition is false and body of if will not be executed.

2. Next step is analyzing condition if (summa > 500 && summa <=1000). 5000 is within the range, the condition is true and body of if ill be executed and we will get on the screen a message about 10% discount.

3. And at last the program will verify the condition if (summa>1000), which is false. Action related to if will not be executed, but this individual operator if has its own «else», which will work off in our case. And there will be message on a screen without any discount.

Conclusion: first we have found out that operator else relates only to the last if. Second, we have realized that such program realization is not convenient for us. Let's review one more possible solution. Project name **Discount3**.

Candidate solution № 3.

```
#include<iostream>
using namespace std;
void main(){
// a variable is declared for storing the initial amount
int summa;

// reques for amount input from a keyboard
cout<<"Enter item of summa:\n";
cin>>summa;

if(summa>1000){ // if amount exceeds 1000 UAH a discount is 25%
    cout<<"You have 25% discount!!!\n";
    cout<<"You must pay - "<<summa-summa/100*25<<"\n";
}
else{ // if amount does not exceed 1000 UAH we continue analyzing
if(summa>500){ // if amount exceeds 500 UAH a discount is 10%
    cout<<"You have 10% discount!!!\n";
    cout<<"You must pay - "<<summa-summa/100*10<<"\n";
}
else{ // if amount does not exceed 500 UAH we continue analyzing
if(summa>100){ // if amount exceeds 100 UAH a discount is 5%
    cout<<"You have 5% discount!!!\n";
    cout<<"You must pay - "<<summa-summa/100*5<<"\n";
}
else{  // if amount does not exceed 100 UAH There is no discount
    cout<<"You have not discount!!!\n";
    cout<<"You must pay - "<<summa<<"\n";
       }
     }
   }
}
```

After attentive analysis of the example you can notice that each next if can be executed only in case if its «predecessor» was not executed because it is situated within the construction of else of the latter. Thus, we have found an optimal realization code at last. Structure we have just created is called «Ladder if else if», because conditions therein are arranged in a form of a ladder. Now we know what a useful construction it is. There is one last stroke left:

Code optimization.

In previous unit we have faced the rule: if only one command relates to the block if or else then we can curly braces. The thing is that construction if else is considered to be an integral command structure. Therefore, if there is nothing except nested construction within some else, then we can omit curly braces for such else:

```
#include <iostream>
using namespace std;
void main(){
// a variable is declared for storing an initial amount
int summa;

// request for amount input from a keyboard
cout<<"Enter item of summa:\n";
cin>>summa;

if(summa>1000){ // if amount exceeds 1000 UAH a discount is 25%
        cout<<"You have 25% discount!!!\n";
        cout<<"You must pay - "<<summa-summa/100*25<<"\n";
 }

// if amount does not exceed 1000 UAH we continue analysis
else if(summa>500){  if amount exceeds 500 UAH a discount is 10%
                cout<<"You have 10% discount!!!\n";
                cout<<"You must pay - "<<summa-summa/100*10<<"\n";
 }

 // if amount does not exceed 500 UAH we continue analysis
else if(summa>100){ // if amount exceeds 100 UAH a discount is 5%
                        cout<<"You have 5% discount!!!\n";
                        cout<<"You must pay - "<<summa-summa/100*5<<"\n";
 }
else{  // if amount does not exceed 100 UAH you don't have a discount
                        cout<<"You have not discount!!!\n";
                        cout<<"You must pay - "<<summa<<"\n";
 }
}
```

That's it! We've solved the problem. We obtained an integral construction of multiple choices consisting of separate interconnected conditions. Now we can proceed to the next lesson units, and we will study in details a few more examples of if else use.

8. Use case: creating text quest

Problem statement

Definitely we are familiar with quest game genre. A hero of such a game should fulfill different tasks, answer the questions, and take decisions that influence game result. Now we will try to create a so called text quest (quest without graphics). Our task is to propose a hero action options and depending on his choice to build a situation. Project name is **Quest.**

Realization code.

```
#include <iostream>
using namespace std;
void main()
{
    // Welcome. Three trials of honor. An evil magician has kidnapped
    //princess and her fate is in your hands. He proposes to you
    //to pass 3 trials of honor in his labyrinth.
    cout<<"Welcome. Three tests of honour. The malicious magician has stolen\n\n";
    cout<<"\nprincess and its destiny in your hands. It suggests you\n";
    cout<<"\nto pass 3 tests of honour in its labyrinth.\n";


    bool goldTaken, diamondsTaken, killByDragon;
    //You enter the first room and there is much gold.
    cout<<"You enter into the first room, here it is a lot of gold.\n\n";
    // Will you take it?
    cout<<"Whether you will take it?(1-yes, 0-no)\n\n";
    cin>>goldTaken;
    if(goldTaken) //if you take
    {
        //You can keep the gold, but you have failed a trial. GAME OVER!!!
        cout<<"Gold remains to you, but you have ruined test. GAME is over!!!\n\n";
    }
    else // if no
    {
        //Congratulations, you have passed the first trial of honor!
```

```
        cout<<"I congratulate, you have passed the first test abuse!\n\n";
        //You move on to the next room. It is full of brilliants
        cout<<"You pass in a following room. It is full of brilliants \n\n";
        //Would you take the brilliants?
        cout<<"Whether you will take brilliants? (1-yes,0-no)\n\n";
        cin>>diamondsTaken;
        if(diamondsTaken)// if you take
        {
                //You can keep the brilliants, but you have failed the trial
                cout<<"Brilliants remain to you, but you have ruined the second test\n\n";
                //GAME OVER!!!
                cout<<"GAME is over!!!\n\n";
        }
        else //if no
        {
                //Congratulations, you have passed the second trial of honor!!!
                cout<<"I congratulate, you have passed the second test abuse!!!\n\n";
                //You enter the third room.
                cout<<"You enter into the third room. \n\n";
                //A dragon attacked a peasant! To move further
                cout<<"The person was attacked by a dragon! To move further \n\n";
                // not paying attention on them
                cout<<"Not paying to them of attention (1-yes,0-no)?\n\n";
                cin>>killByDragon;
                if(killByDragon)//if you take
                {
                        //You try to steal by but a dragon
                        cout<<"You try to pass past, but a dragon \n\n";
                        //notices your presence.
                        cout<<"notices your presence\n\n";
                        //It turns you into ash. You are dead!!!
                        cout<<"It transforms you into ashes. You are dead!!!\n\n";
                        //GAME OVER!!!
                        cout<<"GAME is over!!!\n\n";
                }
                else//if no
                {
                        //Congratulations, you have passed all the trials!!!
                        cout<<"I congratulate, you with honour have was tested all!!! \n\n";
                        //Princess is yours!!!
                        cout<<"Princess gets to you!!!\n\n";
                }
        }
    }
}
```

In spite the primitiveness of the example you can be assured that having some minimal knowledge we can write a program that can entertain an average kid. It happened because we have powerful tool — conditional operators.

9. Use case on belongment of a point to a circle

Problem statement

There is a circle drawn on a plane with the center in point (x0,y0). And edge radiuses r1<r2. Besides, there is a point with the coordinates on the same plain (x, y). It is necessary to find out whether this point belongs to a circle. Project name is **Circle2.**

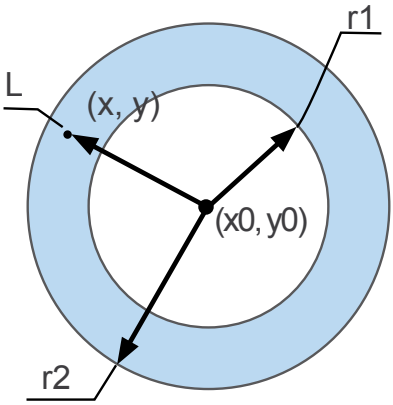
Problem solution

To solve the problem we need to calculate a distance from the circle center to the point and compare it with its radiuses:

1. If the length of a segment from the center to the point is less than radius of external circle but bigger than inner circle, then the point belongs to a circle.

if(L>=r1&&L<=r2)

Otherwise point does not belong to a circle.

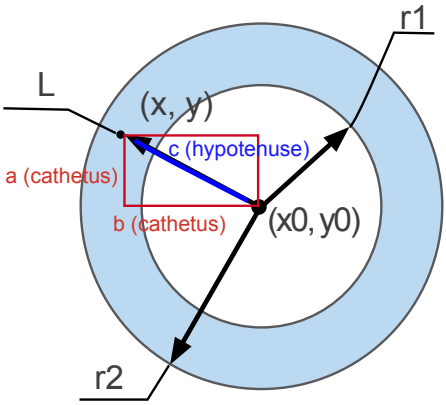


2. In order to find a distance from the center to the point we use the theorem of Pythagoras — **the square of the hypotenuse is equal to the sum of the squares of the other two sides. Consequently — the length of the hypotenuse is equal to square root of the sum of other two sides.**

Note: we will need additional knowledge to calculate the power and square root.

1. It is necessary to connect up a library to the program in order to use mathematical functions math.h.

2. To raise to a power we use a function pow (double num,double exp), where num is a number for raising to power, and exp — a power.



3. To extract the square root a function sqrt is used (double num), where num is a number from which the root is extracted;

L=c;

3. Now we are left to find out the length cathetus, and we can see from a picture how to do it.

a=y–y0;

b=x–x0;

How we just have to assemble all parts of the solution into one integral unit.

```
#include <iostream>
#include <math.h>
using namespace std;
void main() {
  // Declaration of variables
  int x0, y0, r1, r2, x, y;
  float L;

  // Request for necessary data input
cout<<"Input coordinates of circle's center (X0, Y0):";
  cin>>x0>>y0;
  cout<<"Input circle radiuses R1 and R2:";
  cin>>r1>>r2;
  cout<<"Input point coordinates (X, Y): ";
  cin>>x>>y;

// Derivation of a formula
  L = sqrt(pow(x - x0, 2) + pow(y - y0, 2));

//Analysis of the results
  if ((r1 < L)&& (L < r2 )) {
       cout<<"This point is situated inside the circle.\n";
      }
 else {
       cout<<"This point is not situated inside the circle.\n";
      }
}
```

10. Structure of multiple choice switch

So far we are familiar with the conditions analyzing — construction if, as well as with the ternary operator. Another choice operator — switch operator. Let's assume that we need to write a program with five options menu. For example, a small kids application that can add, deduct, etc. we can realize choice processing using a ladder if else if as follows:

```
# include <iostream>
using namespace std;
void main(){
// declaration of variables and value input from a keyboard
float A,B,RES;
cout<<"Enter first digit:\n\";
cin>>A;
cout<<"Enter second digit:\n":;
cin>>B;

// program menu realization
char key;
cout<<"\nSelect operator:\n»;
cout<<"\n + - if you want to see SUM.\n";
cout<<"\n - - if you want to see DIFFERENCE.\n";
cout<<"\n * - if you want to see PRODUCT.\n0";
cout<<"\n / - if you want to see QUOTIENT.\n";

//waiting for user's choice
cin>>key;

if(key=='+') { // if a user chose adding
 .      RES=A+B;
 .      cout<<"\nAnswer: "<<RES<<"\n";
  }
else if(key=='-'){ // if a user chose deducting
```

```
.       RES=A-B;
.       cout<<"\nAnswer: "<<RES<<"\n";
 }
else if(key=='*'){ // if a user chose multiplying
.       RES=A*B;
.       cout<<"\nAnswer: "<<RES<<"\n";
 }
else if(key=='/'){ // if a user chose dividing
.       if(B){ // if a factor is not equal to zero
.               RES=A/B;
.               cout<<"\nAnswer: "<<RES<<"\n";
.       }
.       else{ // if a factor is equal to zero
.               cout<<"\nError!!! Divide by null!!!!\n";
.       }
 }
else{ // if input symbol is incorrect
.       cout<<"\nError!!! This operator isn't correct\n";
 }
}
```

An example described above is quite correct, but looks a bit bulky. We can considerably simplify the code. And switch is used exactly with this purpose. It allows comparing variable value with a range of values and having met a match and making a particular action.

General syntax and principle of operation.

At first, let's review a general syntax of an operator:

```
switch(expression){
case value1:
            action1;
              break;
case value2:
            action2;
              break;
case value3:
            action3;
              break;
......
default:
default_action;
              break;
}
```

Let's analyze this record form:

1. Expression is the data that have to be validated against. Here we can indicate a variable (but only of char type or integer) or expression with the integers in a result.

2. Case Value1, case value2, case value3 — integer of character constant value an expression validates against.

3. Action1, action2, action3 — Actions that should be executed if expression value matched case value.

4. If there is a match and the action related to the matched case, switch finishes its work and the program shifts to the next string behind the closing curly brace of switch operator. Break operator is responsible for this function and it also terminates execution of switch.

5. If there were no matches during the analysis then default section is activated and a default_action is executed. Default operator is an analogue of else operator.

Now let's see how to simplify an example given in the beginning of the topic.

Optimizing an example.

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

// variables declarations and value input from a keyboard
float A,B,RES;
cout<<"Enter first digit:\n";
cin>>A;
cout<<"Enter second digit:\n";
cin>>B;

// realization of program menu
char key;
cout<<"\nSelect operator:\n";
cout<<"\n + - if you want to see SUM.\n";
cout<<"\n - - if you want to see DIFFERENCE.\n";
cout<<"\n * - if you want to see PRODUCT.\n";
cout<<"\n / - if you want to see QUOTIENT.\n";

//waiting for user's choice
cin>>key;

// verifying the value of key variable
switch(key){
case  '+':    // if user chooses adding
        RES=A+B;
        cout<<"\nAnswer: "<<RES<<"\n";
        break; // termination of switch
case  '-':     // if user chooses deduction
        RES=A-B;
        cout<<"\nAnswer: "<<RES<<"\n";
        break; // termination of switch
case  '*':    // if user chooses multiplying
        RES=A*B;
        cout<<"\nAnswer: "<<RES<<"\n";
        break; // termination of switch      case  '/':
        // if user chooses dividing
        if(B){ // if a factor is equal to zero
                 RES=A-B;
                 cout<<"\nAnswer: "<<RES<<"\n";
        }
        else{ // if a factor is equal to zero
                 cout<<"\nError!!! Divide by null!!!!\n";
        }
        break; // termination of switch
default:    // if input value is incorrect
        cout<<"\nError!!! This operator isn't correct\n";
        break; // termination of switch
  }
}
```

So you can see the code now looks simpler and more convenient and it is more readable.

Switch operator is rather easy to use, though it is necessary to know some peculiarities of its operation:

1. If there are character values used within the case they have to be indicated in single quotes, and in case of integer values — without quotes.

2. default operator can be positioned in any place within switch system, anyways it will be executed only in case if there is not match. Though a «good manner» is to indicate the default in the end of the construction.

```
switch(expression){
case value1:
      action1;
      break;
case value2:
      action;
      break;
default:


      default_action;
      break;
case value3:
      action3;
      break;
}
```

3. After the very last operator within a list (either case or default) break operator can be omitted.

```
switch(expression){
case value1:
      action1;
      break;
case value2:
      action2;
      break;
default:
      default_action;
      break;
case value13:
      action3;
}
switch(expression){
case value1:
      action1;
      break;
case value12:
      action2;
      break;
case value3:
      action3;
      break;
default:
      default_action;
}
```

4.Default operator can be omitted in case if there is no matches, simply nothing will happen.

```
switch(expression){
case value1:
     action1;
       break;
case value2:
     action2;
       break;
case value3:
     action3;
       break;
}
```

5. In case it is necessary to execute the same set of actions for different values of verified expression, we can write a few marks in a raw. Let's review an example of a program that converts a character assessment system into digital.

```
# include <iostream>
using namespace std;
void main(){
// declaration of a variable for storing character esessment
char cRate;

// a request to input a character assessment
cout<<"Input your char-rate\n";
cin>>cRate;

//analysis of introduced value
switch (cRate) {
case 'A':
case 'a':
        // assessment A or a is equal to 5
        cout<<"Your rate is 5\n";
        break;
case 'B':
case 'b':
        // assessment B or b is equal to 4
        cout<<"Your rate is 4\n";
        break;
case 'C':
case 'c':
        // assessment C or c is equal to 3
        cout<<"Your rate is 3\n";
        break;
case 'D':
case 'd':
        // assessment D or d is equal to 2
        cout<<"Your rate is 2\n";
        break;
default:
        // other characters are incorrect
        cout<<"This rate isn't correct\n";
    }
}
```

This example is notable for the fact that что by means of two consecutive case we can reach case independence. Meaning it — case is not important whether a user inputs a capital or lower case letter.

Common mistake.

We have already reviewed all the necessary information about switch operator, now we need only information about the problem a developer may face while using the operator.

If by chance we omit break in any case block, except the last one, and this block will finish off in the end, then execution of the switch will not be broken. The block of operator case, which will be next to the executed one, also will be executed without check.

Error example.

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

// realization of software menu
int action;
cout<<"\nSelect action:\n";
cout<<"\n 1 - if you want to see course of dollar.\n";
cout<<"\n 2 - if you want to see course of euro.\n";
cout<<"\n 3 - if you want to see course of rub.\n";

//waiting of user choice
cin>>action;

//verification of the value of variable action
switch(action){
case  1:    // if user chose dollar
        cout<<"\nCource: 5.2 gr.\n";
        break; // switch break
case  2:    // if user chose euro
        cout<<"\nCource: 6.2 gr.\n";
         //break;  commented out break switch
case  3:    // if user chose rubles
        cout<<"\nCource: 0.18 gr.\n";
        break; // break switch
default:     // if a choice is erroneous
        cout<<"\nError!!! This operator isn't correct\n";
        break; // break switch
 }
}
```

There will be an error in case we choose menu option 2. In case of value 2, break operator is commented out. And this error result will be displayed as follows:

```
Select action:

 1 - if you want to see course of dollar.

 2 - if you want to see course of euro.

 3 - if you want to see course of rub.
2

Cource: 6.2 gr.

Cource: 0.18 gr.
Press any key to continue
```

Except necessary information the screen displays what was within a block case that is situated after erroneous construction. It is important to avoid such mistypes because they lead to the errors at the execution stage.

Within the present lesson we have acquainted with the operators, which allow analyzing any data. Now you can proceed to your home assignment. Good luck!

11. Home assignment

1. . Write a program checking the number input from a keyboard for its even parity.

2. There is a natural number a (a<100). Write a program displaying a number of digits within this number and the sum of the digits.

3. We know that 1 inch is equal to 2.54 cm. Develop an application converting inches into centimeters and vice versa. Dialog with a user is realized through menu system.

4. Write a program realizing a popular TV game «Who wants to be a millionaire».