Sub-faculty of information technologies

COMPUTER SCIENCE
Module 2

Programming of problems with loops and arrays

Part 1

Lecture notes

Odessa 2012

Compiler – **Y. V. Prokop**

These lecture notes contain theoretical information and examples of programs in C++ Builder with loops and arrays. The lecture notes will be useful for students of the Academy of Telecommunications who are studying in English, fixing theoretical material, preparing to laboratory training and exercises in the discipline of Computer science in the second module.

It is intended for the acquisition of skills for operation on a personal computer and programming by students of the academy studying in English, with the purpose of further usage of these skills in daily professional work. Also, it will be useful for users of personal computers wishing to learn programming in C ++ Builder environment.

The lecture notes have been approved by the sub-faculty IT meeting

Minutes № 7 from 27.02.2012

The lecture notes are considered and approved by the faculty of Informational Networks meeting

Minutes № 16 from 23.03.2012

Contents

Introduction

Structure of the module

The discipline of Computer science is studied in I - II semesters and is intended for the training of students for the professional use of personal computers.

The purpose of the Computer science course is to form knowledge and skills in areas such as:

- architecture of a personal computer
- operation of Windows operating system
- algorithmization of computing processes
- compilation of programs in C ++ programming language
- acquaintance to object-oriented programming on the example of solution of elementary tasks in C ++ Builder programming environment.

The course program consists of four modules:

- module 1 – "The basics of the personal computer and the organization of computing processes";
- module 2 – "Programming loops and arrays";
- module 3 – "Programming structured data";
- module 4 – "Programming lists and files. Basics of object-oriented programming".

According to the curriculum, the structure of the module is:

Type of activity	Hours
Lectures	8
Exercises	16
Laboratory Training	16
Total Classwork Hours	40
Individual Work and Self-study	27
TOTAL	67

The subject schedule of lectures in module 2

Lecture 1. Loop statements.
Lecture 2. Functions.
Lecture 3. Arrays.
Lecture 4. Multi-dimensional arrays.

The list of exercises in module 2

1. Programming loop statements.
2. Programming functions.
3. Programming one-dimensional arrays.
4. Programming multi-dimensional arrays.

The list of laboratory trainings in module 2

1. The organization of cyclic calculations by means of the FOR loop statement.
2. The organization of cyclic calculations by means of the FOR loop statement.
3. Calculations with the use of conditional loop statements.
4. Calculations with the use of conditional loop statements.
5. Programming functions.
6. Programming one-dimensional arrays.
7. Programming one-dimensional arrays.
8. Programming multi-dimensional arrays.

Recommendations for students' self-study

Sort of activity	Hours
Working up lectures	4
Studying additional information	4
Preparing to exercises	4
Preparing to laboratory trainings	9
Execution of the complex task on the theme: "Creating algorithms and programs with cyclic structure and arrays".	6
Total:	27

The teacher gives out individual variants of the complex task.

The complex task must be written in a separate exercise-book. Each problem must contain:
✓ scheme of the algorithm;
✓ form of the project;
✓ text of the program in C++ Builder;
✓ results of calculation.

Prerequisites

The study of computer science is based on the high-school course of computer science (students should possess the knowledge in the size of the school course of computer science according to the program of the Ministry of Education) and is based on the school courses of mathematics and some topics of higher mathematics, such as functions, formulas of conversion, factorial, series, integral calculus, matrices, etc.

It is supposed that, starting learning the content of the module 2 of Computer Science course, the student has already studied module 1 material, and has acquired the following knowledge and skills:

knowledge on the themes:
✓ architecture of computer;
✓ OS Windows;
✓ C++ Builder IDE;
✓ algorithm, its properties and means of description;
✓ elements of C++ programming language and programs with linear structure;
✓ logical expressions and their priority;
✓ flow control;

skills:
- ✓ operating with files in Windows;
- ✓ creating algorithms and programs with linear and branching structure in C++ Builder and executing them on a computer.

Literature

1. Prokop Y. V. Computer Science. Module 1. – OHAC, 2009.
2. Леонов Ю.Г., Угрік Л.М., Швайко І.Г. Збірник задач з програмування. – Одесса: УДАЗ, 1997.
3. Трофименко О.Г., Прокоп Ю.В., Швайко І.Г. та ін. С++. Теорія та практика. – Одесса: ОНАЗ, 2011. – 587 с.
4. Архангельский А.Я., Тагин М.А. Программирование в C++ Builder 6 и 2006. – М.: «Бином», 2007. – 1184 с.
5. Архангельский А.Я. Программирование в C++ Builder 5.– М.:«Бином», 2000.– 1152с.
6. Березин Б.Н., Березин С.Б. Начальный курс С и С++. – М.: «Диалог-МИФИ», 2000. – 288 с.
7. Бьерн Страуструп Язык программирования С++. – СПб.– М.: Бином, 1999. – 991 с.
8. The cplusplus.com tutorial – http://www.cplusplus.com

1. Loop statements

Loops are used to repeat a *statement* or *a series of statements* a certain number of times or while a condition is true.

The `for` loop

`For` loop is a C++ construction that allows you to repeat a statement or a set of statements for a known number of times or until some condition is true. It is usually used to construct loops that must execute a specified number of times. The syntax for the **for** loop is as follows:

```
for (Initialization; Condition of continuation; Modification of
parameters)
      {StatementBlock;}
```

where

- *Initialization* initializes the loop control variable, for example

  ```
  i = 0;
  ```

  This expression is executed only once. Control then passes to *Condition of continuation*;

- *Condition of continuation* is a test that will stop the loop as soon as it is false. Or, in other words, the repetition will continue as long as the condition is true;

- *Modification of parameters* is a statement that modifies the loop control variable appropriately, for example `i = i+1` or `i++;`

- *StatementBlock* is either a single statement or a group of statements inside the braces {...}.

Example 1.1: Make a piece of program that prints out the first 10 positive integers, together with their squares.

```
for (int i = 1; i < 11; i++)
      Memo1->Lines->Add(IntToStr(i) + "  " + IntToStr(i*i));
```

You can see the three parts that make the `for` loop.

1. i is initialized to 1.

2. Then the condition i < 11 is checked. As it is true, the values of i and i*i (1 1) are added to Memo1.

3. After that, the value of i increases by 1 (it becomes 2), and the program returns to the step 2 (checking the condition).

Repetition will stop when on the second step the condition is false. The loop will repeat for these values of variable i: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Afterwards, i will be equal to 11, and the condition 11<11 will be false.

Example 1.2: Write a piece of program that prints out all positive odd integers between 1 and 100, together with their squares.

The program is similar to the above; except that this time we will simply add 2 to i instead of just 1. Here's the code:

```
for (int i = 1; i < 11; i += 2)
    Memo1->Lines->Add(IntToStr(i) + "  " + IntToStr(i*i));
```

Example 1.3: Write a program that prints out all positive even integers between 1 and 100, together with their squares.

Again, the program is as before, but we simply start the loop at 2 instead of 1 to catch all even integers. Here's the code:

```
for (int i = 2; i < 11; i += 2)
    Memo1->Lines->Add(IntToStr(i) + "  " + IntToStr(i*i));
```

Initialization and *Condition of continuation* in for loop can contain multiple statements separated by the *comma*. For example:

```
for (int i = 5, int j = 10 ; i + j < 20; i++, j++ )
    Memo1->Lines->Add(IntToStr(i + j));
```

If *Condition of continuation* is omitted, it is considered true and the for loop will be infinite. For example:

```
for( int i=0; ;i++ )
{
    // Statements to be executed.
}
```

Infinite loop runs endlessly and is a bug of the program. One of the ways of quitting the loop is a `break` statement.

Example 1.4: Write a program that finds the sum of the first 100 positive integers.

We need to use a loop where some variable changes from 1 to 100. This variable is `i`. Besides that, we need a variable to store the sum. We'll call it `sum`. As the numbers are integer, the sum is also integer. In the beginning, the <u>sum is 0</u>. Then we add numbers to it sequentially one by one.

1. Start with `sum = 0`. If we do not assign 0 to the sum, compiler may assign any number to it (for example, a negative number with 5 digits).
2. Start a loop with `i = 1` (the numbers we need to add begin from 1).
3. Add `i` to `sum` and save the result back into the `sum` (now `sum` is 1).
4. Increase `i` by 1 (now it is 2).
5. Again add `i` to `sum` and save the result back into `sum` (now `sum` is 3).
6. Increase *i* by 1 (now it is 3).
7. Again add `i` to `sum`, and save the result back into `sum` (now `sum` is 6)
8. Increase `i` by 1 (now it is 4)
9. And so on, until `i` reaches 100.

Here is the code that will accomplish this:

```
int sum = 0;
for (int i = 1; i <= 100; i++)
  sum += i;
Edit1->Text = IntToStr(sum);
```

The scheme of **for** loop is:

Although the three fields of the `for` statement are normally used for initialization, testing for termination, and incrementing, they are not restricted to these uses. For example, the following code calculates the sum of the first 20 positive integers. In this case, *StatementBlock* is the null statement:
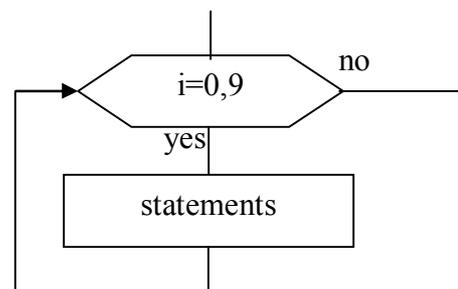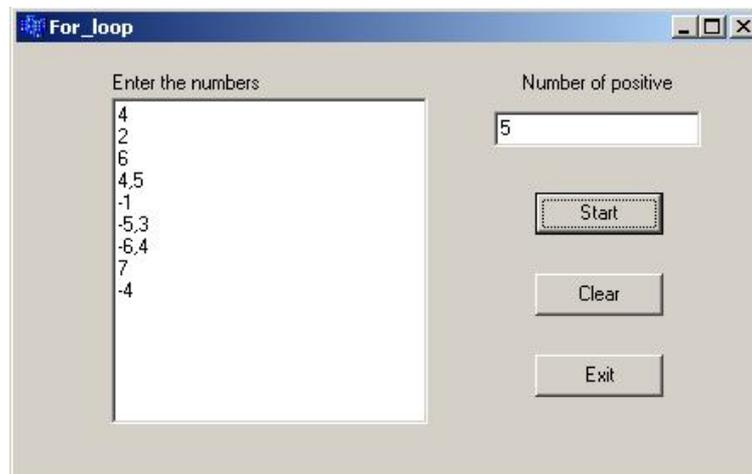
```
int i, sum;
for (sum = 0, i = 1; i <= 100; sum += i, i++)
  ;
```

The following code is equivalent:

```
int sum = 0, i = 1;
for (; i <= 100;)
  { sum += i;
    i++;
  }
```

Example 5: Write a program that counts the number of positive doubles in Memo.


```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
int N = Memo1->Lines->Count;      //number of lines in Memo
double x;
int kol = 0;                      //at first kol must be 0
for (int i = 0; i < N; i++)
  {x = StrToFloat(Memo1->Lines->Strings[i]);  //read a number from Line i
    if (x > 0) kol++;             //if the number is positive, kol increases by 1
  }
Edit1->Text = IntToStr(kol);
}
```

The scheme is:

begin

N = Memo1->Lines->Count;

kol = 0

i = 0, N-1 → no → kol → end

yes

Read a number

x > 0 → no

yes

kol++
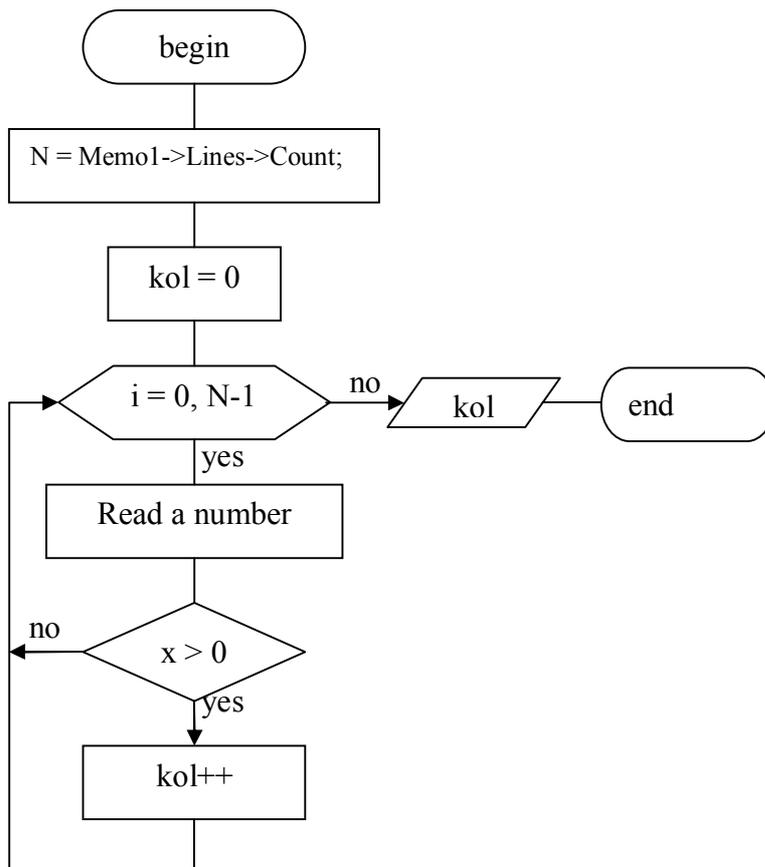
Each repetition of the loop is called **iteration**.

The `while` loop (loop with precondition)

While loop is used when we do not know how many times we need to repeat statements, but we do know the condition of repeating. Its format is:

```
while (Condition)
    {statements;}
```

and its purpose is to repeat *statements* while *Condition* is true.

The compiler first examines the *Condition*. If the *Condition* is true, then it executes the *Statements*. After executing *Statements*, *Condition* is checked again. As long as the *Condition* is true, it will keep executing the *Statements*. Once the *Condition* becomes false, it exits the loop.

Examples 1-3 can be rewritten with `while` loop:

Example 1.1a: Make a piece of program that prints out the first 10 positive integers, together with their squares.

```
int i = 1;
while(i < 11)
  {Memo1->Lines->Add(IntToStr(i) + "  " + IntToStr( i*i));
   i++;
  }
```

Example 1.2a: Write a piece of program that prints out all positive odd integers between 1 and 100, together with their squares.

```
int i = 2;
while (i <11)
  {Memo1->Lines->Add(IntToStr(i) + "  " + IntToStr (i*i));
   i += 2;
  }
```

Example 1.3a: Write a program that prints out all positive even integers between 1 and 100, together with their squares.

```
int sum = 0, i = 1;
while (i <= 100){
  sum = sum + i;
   i++;
}
Edit1->Text = IntToStr(sum);
```

Example 1.6. Make a program to print out positive integer numbers from 1 until their sum becomes greater than 20. We will add a number x to the sum while the sum is less than or equal to 20.

```
int x = 1;                //first number is 1
int sum = 0;
while (sum <= 20) {
  Memo1->Lines->Add(IntToStr(x));
  sum += x;               //sum = sum + x;
  x++;
}
```

1. At first, $x = 1$ and sum = 0. As the sum < 20, the condition is true and while loop begins to work.
2. Number 1 is added to Memo1.
3. sum becomes 1.
4. x becomes 2
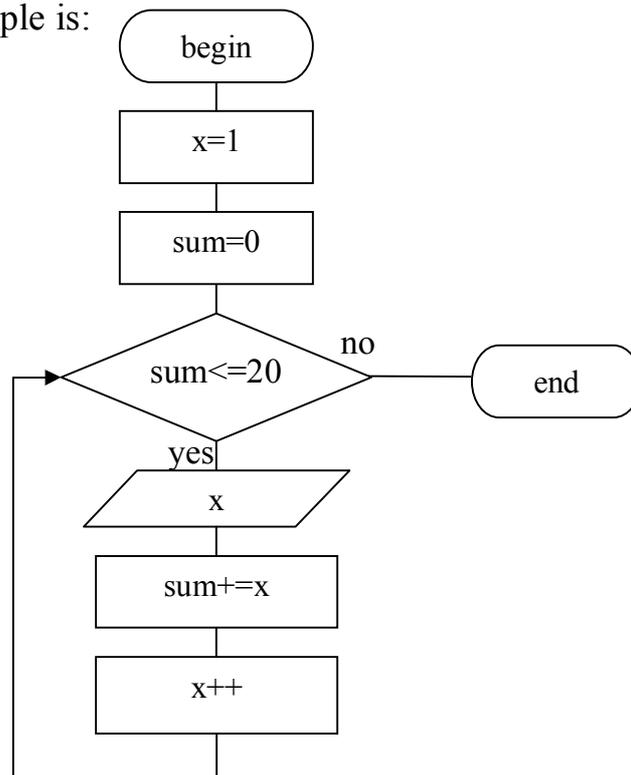5. Return to condition. It is true.

Items 2.– 4. repeat again and again.

sum	0	1	3	6	10	15	21
x		1	2	3	4	5	6	7

From the table above we see that when $x = 6$, the sum becomes 21. When we come to the condition sum < 20, it is false and while stops.

The scheme of while loop is:
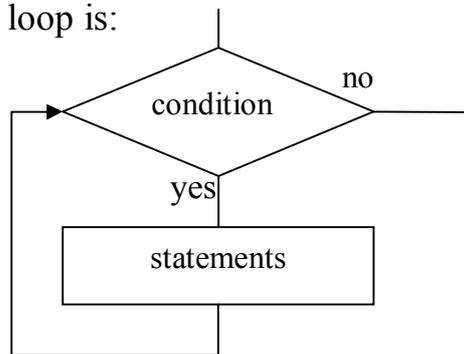
The scheme of the last example is:

13

The `do-while` loop (loop with postcondition)

The `do-while` loop is used when it is convenient to check up the condition **after** the statements.

Format:

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

The `do-while` loop executes a *Statement* first. After the first execution of the *Statement*, it examines the *Condition*. If the *Condition* is true, then it executes the *Statement* again. It will keep executing the *Statement* AS LONG AS the *Condition* is true. Once the *Condition* becomes false, the looping (the execution of the *Statement*) would stop.

If the *Statement* is made of one line, you can simply write it after the `do` keyword. Like ~~the~~ `if` and `while` statements, the *Condition* being checked must be included between parentheses. The whole `do-while` loop must end with a semicolon. If the *Statement* spans more than one line, start it with an opening curly bracket and end it with a closing curly bracket.

The previous example with `do-while` loop is:

Example 1.6b: Make a program to print out positive integer numbers from 1 until their sum becomes greater than 20. We will add a number `x` to sum while sum will be less than or equal 20.

```
int x = 1;              //first number is 1
int sum = 0;
do{
  Memo1->Lines->Add(IntToStr(x));
  sum += x;             //sum = sum + x;
  x++;
}while(sum <= 20);
```
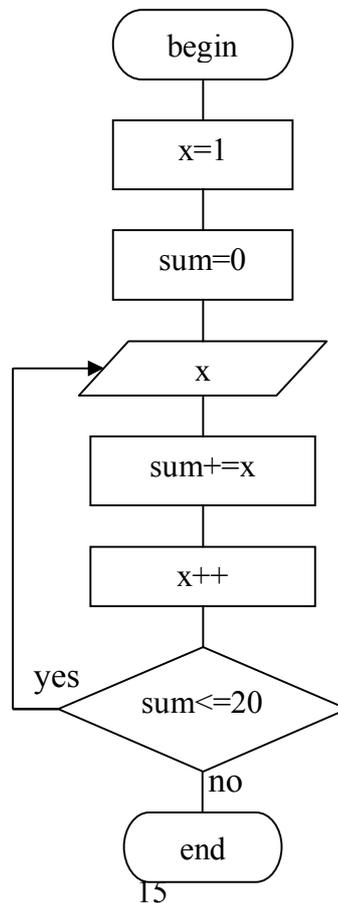
1. At first, x = 1 and sum = 0.

2. Number 1 is added to Memo1 (without any conditions).

3. sum becomes 1.

4. x becomes 2

5. Check up the condition. It is true. And we repeat until sum becomes 21. Then the loop stops.

We use the do-while loop if we want to execute a statement at the first step without checking any condition.

The scheme of do-while is:
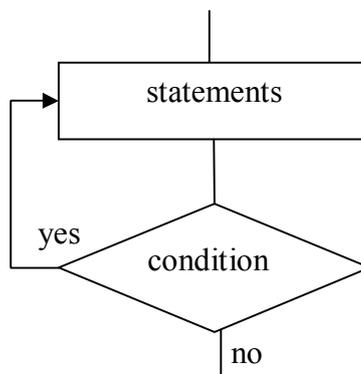
The scheme of the previous example is:

15

Jump statements

Using **break**, we can exit the loop even if the condition for its end is not fulfilled. It can be used to end loop that would otherwise become infinite, or to force a finite loop to end before its natural end.

The **continue** instruction causes the program to skip the rest of the loop in the current iteration as if the end of the statement block would have been reached, causing it to jump to the following iteration.

The purpose of **exit** is to terminate a running program with a specific exit code. Its prototype is:

```
void exit (int exit code);
```

The exit code is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means an error happened.

Example 1.7: Write a program that reads numbers from Memo until number zero is found. Find the smallest number.

While searching for the smallest (largest) number, we use the following algorithm:

1. The first number is declared as the smallest, and its value is assigned to a temporary variable, which presents the current minimum.

2. We search through the other numbers, and if one of them is smaller than our current minimum, we update our temporary variable to the new current minimum.

3. After we process all numbers, the temporary variable will store the real minimum number.

Example:

The numbers are: 5, 6, 3, 9, 4, 7, 2, -1, 5.

5	6	3	9	4	7	2	-1	5
	6<5?	3<5?	9<3?	4<3?	7<3?	2<3?	-1<2	5<-1
min=5		min=3				min=2	min=-1	

Answer: minimum number is -1.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int min, x,i=0;
    int n=Memo1->Lines->Count;
    x=StrToInt(Memo1->Lines->Strings[0]);
    min = x;                    //we assume the first given number is the
smallest
    while (x != 0 && i<n) {
        i++;
        x = StrToInt(Memo1->Lines->Strings[i]);
        if(x < min )
            min = x;
     }
    Edit1->Text = IntToStr(min);
}
```

Example 1.8: Write a program that reads numbers from Memo until number zero is found. Find the smallest positive number.

The difficulty is to find the first positive number and assign it to min. We use two loops: the first one to exclude the negative numbers in the beginning of Memo and the second – to search for the minimum.
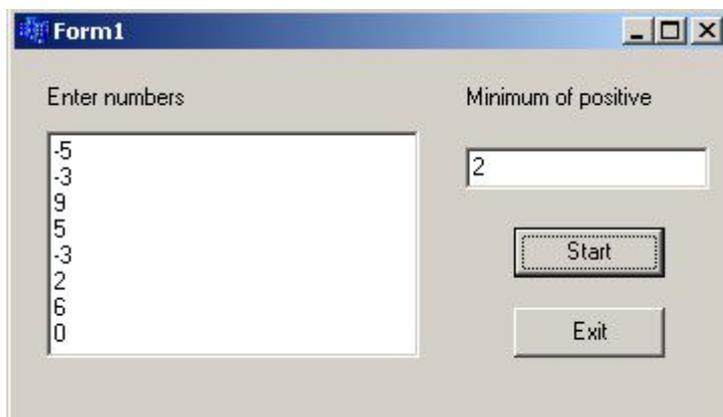

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
```

```
    int min, x, i = 0;
    int n = Memo1->Lines->Count;
    do{
      x = StrToInt(Memo1->Lines->Strings[i]);
      i++;
    }while(x <= 0 && i < n);
    min = x;                        // we assume the first given number is the
smallest
    while (x != 0 && i < n) {
       x=StrToInt(Memo1->Lines->Strings[i]);
       if(x > 0 && x < min )
           min = x;
       i++;
    }
    Edit1->Text =IntToStr(min);
}
```

This is one more way to find the minimum.

Example 1.8a: Write a program that reads numbers from Memo until number zero is found. Find the smallest positive number. Use another method to find the minimum

We read numbers from Memo. If the number is negative, we ignore it. If the number is positive and min is 0, it means that this is the first positive number, and we need to assign it to min without any comparison. If the number is positive and min is not 0, it means that this positive number is not the first one and min has some value (temporary minimum). In this case, we compare the number with min and assign the number to min if it is smaller than min.

```
    void __fastcall TForm1::Button1Click(TObject *Sender)
    {
    int min=0, x,i=0;
    int n=Memo1->Lines->Count;
    do{
```

```
        x=StrToInt(Memo1->Lines->Strings[i]);

        if(x > 0)

            if(min == 0 || x < min )

                min = x;

        i++;

    }while(x != 0 && i < n);

    Edit1->Text = IntToStr(min);

}
```

Nested loops

The placing of one loop inside the body of another loop is called nesting. When you "nest" two loops, the outer loop takes control of the number of complete repetitions of the inner loop. While all types of loops may be nested, the most commonly nested loops are for loops.

When working with nested loops, the outer loop changes only after the inner loop is completely finished (or is interrupted).

The syntax for a nested `for` loop statement in C++ is as follows:

```
for (Initialization; Condition of continuation; Modification of
parameters)
{
    for (Initialization; Condition of continuation; Modification of
parameters)
    {
       statements;
    }
    statements;
  }
```

The syntax for a nested `while` loop statement in C++ is as follows:

```
while(condition)
{
```

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

The syntax for a nested `do...while` loop statement in C++ is as follows:

```
do
{
  statements;
  do
  {
     statements;
  }while( condition );
}while( condition );
```

You may nest different flavors of loops. For example:

```
while(condition)
{
    for (Initialization; Condition of continuation; Modification of
parameters)
    {
      statements;
    }
  statements;
}
```

You should never use the same `loop` variable for both inner and outer loops.

Example 1.9: Write the code that uses a nested for loop to find the prime numbers from 2 to 20.

```
int i, j;
for(i = 2; i < 20; i++) {
```

```
    for(j = 2; j*j <= i; j++)

      if(!(i%j)) break; // if factor found, not prime

      if(j > (i/j)) Memo1->Lines->Add(IntToStr(i) + " is prime");

    }
```
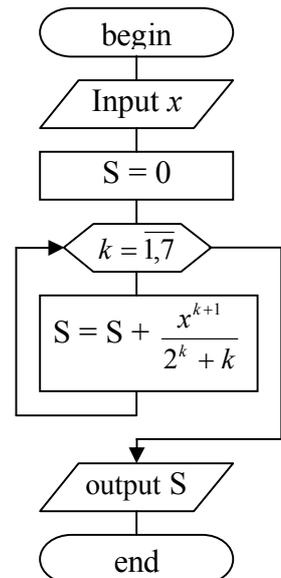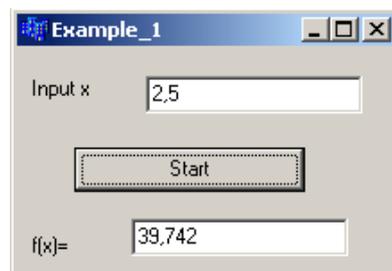
This would produce the following result:

2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime

Example 1.10: Calculate the value of $f(x) = \sum_{k=1}^{7} \frac{x^{k+1}}{2^k + k}$, enter from the screen the value of x.

The number of addends in this example is 7 and we have to repeat the loop 7 times. The addends are $\frac{x^{k+1}}{2^k + k}$, where k is an order number of addend. We may use k as a loop counter (usually it is variable i).

Let x = 2. Instruction

```
for (int k = 1; k <= 7; k++)
  s += pow(x,k+1)/(pow(2,k)+k);
```

means that:

1. At first k = 1. We put this value in the addend's formula ($\frac{x^{1+1}}{2^1 + 1} = \frac{2^2}{3} = \frac{4}{3} = 1.33$) and add it to sum (s becomes 1.33). After this k increases by 1 and becomes 2.

2. We check up the condition k<=7. It is true, therefore we repeat again. Put the value of k=2 in the addend's formula $\frac{x^{2+1}}{2^2 + 2} = \frac{2^3}{6} = \frac{8}{6} = 1.33$ and add it to sum: s=1.33+1.6=2.93. After this, k increases by 1 and becomes 3.

21

3. Again check up the condition `k<=7`. It is true. We put the value of `k=3` in the addend's formula $\dfrac{x^{3+1}}{2^3+3}=\dfrac{2^4}{11}=\dfrac{16}{11}=1.45$ and add it to sum: `s=2.66+1.45=4.11`. After this, `k` increases by 1 and becomes 4.
4. Repeat the same for values of `k`: 4, 5, 6, 7. After this `k` becomes 8. The condition `k<=7` is false and the loop stops.
5. Now we have only to output the result.

The code of this example is:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  float x=StrToFloat(Edit1->Text);
  float s=0;
  for (int k=1; k<=7; k++)
    s+=pow(x,k+1)/(pow(2,k)+k);        //s=s+pow(x,k+1)/(pow(2,k)+k);
  Edit2->Text = FormatFloat("0.000",s);
}
```

Example 1.11: Calculate the value of $S=\sum\limits_{n=-2}^{m}\dfrac{n+1}{n}\prod\limits_{k=1}^{n+3}\dfrac{k}{k+1}$, enter from the screen the value of m.
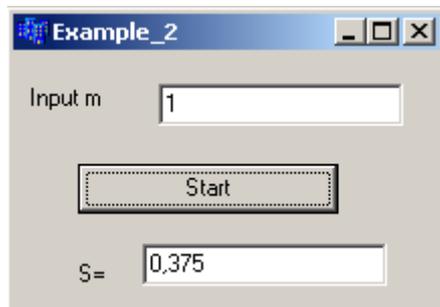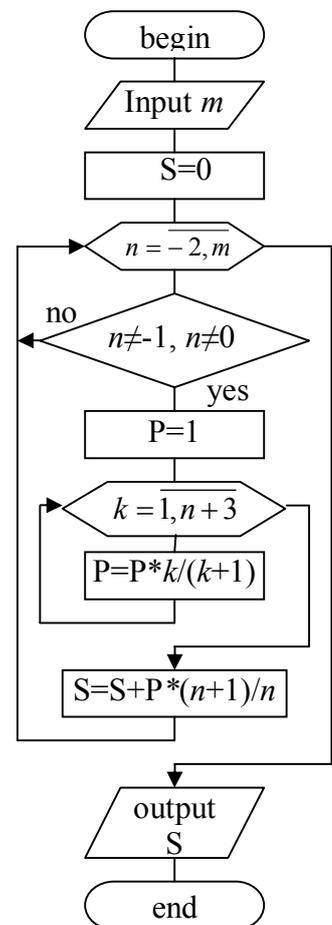
This example is similar to the previous one, but a little more complex.

The number of addends is `m+3`. The addend is $\dfrac{n+1}{n}\prod\limits_{k=1}^{n+3}\dfrac{k}{k+1}$.

Therefore, on each step we have to add $\dfrac{n+1}{n}\prod\limits_{k=1}^{n+3}\dfrac{k}{k+1}$ to the sum.

To do this, we have to calculate the product $\prod\limits_{k=1}^{n+3}\dfrac{k}{k+1}$. It consists

22

of $n+3$ multipliers: $\dfrac{k}{k+1}$.

To calculate each addend, we write the `for` loop:

```
for (k = 1; k <= n+3; k++)
   if  (k != -1 && k != 0)
      p *= (float) k/(k+1);        //or p = p*k/(k+1.);
```

When `k = -1`, there is zero in denominator. When `k = 0`, the multiplier equals to 0 (and all product also equals 0). To exclude these values we check up the condition:

```
if  (k != -1 && k != 0).
```

To calculate the sum, we must write another for loop:

```
for (n = -2; n <= m; n++)        //loop for calculation of the sum
   if (n != -1 && n != 0)
   { p = 1;                    //first value of product is 1
      for (k = 1; k <= n+3; k++)//loop for calculation of the product
         if  (k != -1 && k != 0) p *= (float) k/(k+1);
      S += (float)(n+1)*p/n;//adding the addend to the sum
   }
```

The condition `if (n != -1 && n != 0)` is used to exclude division by 0 and zero addend.

Note that the loop with parameter `k` is inside the loop with parameter `n` (parameters are different!). When one loop is inside another, their parameters must be different.

The code of the program is:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int n, k, m = StrToInt(Edit1->Text);
   float S = 0, p;
   for (n = -2; n <= m; n++)
      if (n != -1 && n != 0)
      { p = 1;
```

```
        for (k = 1; k <= n+3; k++)
            if  (k != -1 && k != 0) p *= (float) k/(k+1);
        S += (float)(n+1)*p/n;
    }
  Edit2->Text = FloatToStrF(S, ffGeneral, 4, 3);
}
```

Function `FloatToStrF(S, ffGeneral, 4, 3)` converts the float number `S` to the string. Parameter `ffGeneral` is the General number format.

2. Functions

Functions allow us to group a series of steps under one name. Imagine that you are washing your clothes in a washing machine. You have to perform the following steps:

- *put clothes into the washing machine*

- *put a powder*

- *select a washing mode*

- *run the machine*

- *wait until the washing machine stops*

- *take out your clothes*

In C++, as well as in most modern programming languages, you can give a name to a series of steps. Let's say we want to call this procedure "wash clothes". We've just created a function to do the work for us. In this example, what you would do is write a function called washClothes (note that C++ won't let you use spaces in the names of functions or variables) that performs the series of steps above, and then, whenever you wanted to wash clothes, you would *call* the function washClothes, which would execute the lines of code necessary to carry out the procedure.

Note: When we say that you are *calling* the function washClothes, we do not mean that you are giving it the name washClothes – you've already done that by

writing the function. We mean you are *executing* the code in the function washClothes. "Calling a function" really means "telling a function to execute".

The first reason why functions are useful is that functions let us create logical groupings of code. If someone is reading your code and he sees that you call a function washClothes, he knows immediately that you are washing clothes. The point is that <u>functions make your code much easier to read</u>.

There is an even better reason to use functions: they can <u>make your code shorter</u>. Having fewer lines of code is not always desirable, but every time you write a line of code, there is a possibility that you are introducing a bug. Functions start to reduce the number of lines of code when you call them repeatedly.

There are other reasons to use functions, at least same or even more important. First, by eliminating redundant code, you reduce the compilation time and the size of the executable. Second, you make it easier to maintain the program.

Suppose that you want to mail out invitations to eight of your friends for a cocktail party. Let's assume that you need to do the following steps in order to invite your friend Alex.

- *write Alex's name on the invitation*

- *write Alex's name and address on the envelope*

- *place the invitation into the envelope*

- *seal and stamp the envelope*

- *drop the envelope into the mail box*

It takes five lines of pseudo-code to invite one friend, so it takes 40 lines of pseudo-code to invite eight friends. That's a lot of repeated code, and any time you repeat the code like this, you are more likely to add a bug to your program.

Functions can substantially reduce the amount of pseudo-code you need to write to invite your eight friends to the party. At the first glance, it seems surprising that you'd be able to reduce that amount at all – each of your friends should receive their own personally addressed invitation, and all of the envelopes have to be sealed and stamped and placed in the mail. How are we going to reduce the number of lines

of code? Let's create a function called `inviteToParty` which performs the following procedure:

- *write Alex's name on the invitation*

- *write Alex's name and address on the envelope*

- *place the invitation into the envelope*

- *seal and stamp the envelope*

- *drop the envelope into the mail box*

Now that we have this function, we can call it eight times to invite our eight friends:

inviteToParty

inviteToParty

inviteToParty

inviteToParty

inviteToParty

inviteToParty

inviteToParty

inviteToParty

You probably noticed a problem with doing it this way. We're inviting Alex eight times, and none of our other friends is going to receive invitations! Alex will get invited eight times because the function invites *Alex* to the party, and the function is being called eight times. The solution is to modify the function so that it invites *friend* to the party, where *friend* can be any of your friends. We'll change our function so that it looks like this:

- *write <u>friend's name</u> on the invitation*

- *write <u>friend's name</u> and address on the envelope*

- *place the invitation into the envelope*

- *seal and stamp the envelope*

- *drop the envelope into the mail box,*

and then we'll change the way in which we call the function:

inviteToParty (Alex)

inviteToParty (Anna)

inviteToParty (Mark)

inviteToParty (Serge)

inviteToParty (Tanya)

inviteToParty (Boris)

inviteToParty (Peter)

inviteToParty (Nick)

Now, each time we call the function, `friend` is a different person, and each of our eight friends will be invited. We've just reduced the number of lines of pseudo-code from 40 to 13 by using a function, and our code became much easier to read.

A function can take some *input*, do some *stuff*, and then produce an *output*.

If you later decide to add the return address to the envelope, you will need to do that in one place rather than looking for occurrences of invitation-sending throughout your program.

Function definition:

```
type name ( parameter1, parameter2, ...) { statements }
```

where:

- `type` is the data type specifier of the data returned by the function.
- `name` is the identifier by which it will be possible to call the function (name of the function).
- `parameters` (as many as needed): each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: `int x`) and which acts within the function as a regular local variable. They allow passing arguments to the function when it is called. Multiple parameters are separated by commas.
- `statements` is the function's body. It is a block of statements surrounded by braces {}.

Example: function `addition` calculates the sum of two integer numbers.

//function definition

```
int addition (int a, int b)
```

```
{
    int r;
    r = a + b;
    return r;
}


...Button1Click...
{
    int z;
    z = addition (5, 3);  //call of function addition with parameters 5 and 3
    Edit1->Text = IntToStr(z);
}
```

The parameters and arguments have a clear correspondence. We called the function `addition` passing two values: `5` and `3`, which correspond to `int a` and `int b` parameters declared for the function `addition`.

At the point at which the function is called from `Button1Click`, the control is lost by `Button1Click` and passed to function `addition`. The values of both arguments passed in the call (5 and 3) are copied to the local variables `int a` and `int b` within the function.

Function `addition` declares another local variable (`int r`), and by means of the expression `r = a+b`, it assigns to `r` the result of `a+b`. Because the actual parameters passed for `a` and `b` are 5 and 3 respectively, the result is 8.

The following line of code:

```
return r;
```

finalizes function `addition`, and returns the control back to the function that called it (in this case, `Button1Click`). At this moment, the program follows its regular course from the same point at which it was interrupted by the call to function `addition`. The `return` statement in function `addition` specified a value – the content of variable `r`, which at that moment had a value of 8; this value becomes the value of the function. The call to a function (`addition (5,3)`) is literally replaced by the value it returns (8).

The return statement (regardless of whether it returns a value) may be anywhere within the function, but it is considered a good style by many to have a single one at the end of the function. Even if not, it is mandatory for functions that return something that all their execution paths have a return statement.

Scope of Variables in Function

The scope of the variables can be broadly classified as

- Local Variables
- Global Variables

Local Variables

The variables defined local to the block of the function would be accessible **only within the block** of the function and not outside the function. Such variables are called **local variables**. That is, in other words, the scope of the local variables is limited to the function in which these variables are declared.

Let us see this in a small example:

```
int ex(int x, int y)
{
  int z;
  z = x + y;
  return(z);
}
…Button1Click…
{
  int b;
  int s = 5, u = 6;
  b = ex(s, u);
  Edit1->Text = IntToStr(b);
}
```

In the above program the variables x, y, z are accessible only inside the function ex( ) and their scope is limited only to the function ex( ) and not outside

the function. Thus, the variables `x`, `y`, `z` are local to the function `ex()`. Similarly, one would not be able to access variable `b` inside the function `ex()` as such. This is because variable `b` is local to `Button1Click`.

Global Variables

Global variables are visible in any part of the program code and can be used within all functions and outside all functions used in the program. The method of declaring global variables is to declare the variable outside the function or block. For instance:

```
int x, y, z;        //Global Variables
float a, b, c;      //Global Variables
int sum(){
  int S = x + y;
  return S;
}


..Button1Click..
{
  int s, u;         //Local Variables
  float w, q;       //Local Variables
  s = sum();
  …
}
```

In the above code, the integer variables `x`, `y`, and `z` and the float variables `a`, `b`, and `c` declared outside all functions are global variables, and the integer variables `s`, `S`, and `u` and the float variables `w` and `q` declared inside the function block are local variables.

Thus, the scope of global variables is between the point of declaration and the end of compilation unit whereas the scope of local variables is between the point of declaration and the end of innermost enclosing compound statement. We can use

global variables (x, y, z, a, b, c) in both sum() and Button1Click. Local variable S is known only in sum(), we can not use it in Button1Click. Local variables s, u, w, and q are known only in Button1Click and we can not use them in sum().

In order to understand the concept of local and global variables scope in detail, let's look at an example that has a number of local and global variable declarations with a number of inner blocks.

```
int g;                    //Global variable
void ex()
{
  g = 30;                 //Scope of g is throughout the program
                          //and so is used between function calls
}
…Button1Click…
{
  int a=1;                //Local in Button1Click, global in if-block
  if (a!=0)
  {
     int b;               //Local in if block
     b=25;

     a=45;                //Global in if block
     g=65;                //Global in program
  }
  a=50;
  ex();
}
```

In this example, the scope of b is till the first braces shaded. The scope of a is till the end of Button1Click brace.

Global variables should be avoided as much as possible.

Functions of `void` type

In the syntax of a function declaration:

```
type name ( argument1, argument2, ...) statement
```

you see that the declaration begins with a `type`, which is the type of the function itself (i.e., the data type that will be returned by the function with the `return` statement). But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case, we should use the void type specifier for the function. This is a special specifier that indicates the absence of any type: **void**.

```
void printmessage ()
{
  ShowMessage("I'm a function!");
}
…Button1Click…
{
  printmessage ();
}
```

Void functions can have void return statements.

`void` can also be used in the function's parameter list to explicitly specify that we want the function to take no actual parameters when it is called. For example, function `printmessage` could have been declared as:

```
void printmessage (void)
{
  ShowMessage("I'm a function!");
}
```

It is optional to specify `void` in the parameter list, however. In C++, a parameter list can simply be left blank if we want to write a function with no parameters.

The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement. The following call of a function is incorrect:

```
printmessage;
```

Arguments passed by value and by reference

Until now, in all functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our function `addition` using the following code:

```
int x = 5, y = 3, z;
z = addition(x, y );
```

What we did in this case was the call to function `addition` passing the values of `x` and `y`, i.e. 5 and 3 respectively, but not the variables `x` and `y` themselves.

This way, when the function `addition` is called, the value of its local variables `a` and `b` become 5 and 3 respectively, but any modification to either `a` or `b` within the function addition will not have any effect on the values of `x` and `y` outside it because variables `x` and `y` were not themselves passed to the function, but only copies of their values at the moment the function was called.

But there might be some cases where you need to manipulate the value of an external variable from inside a function. For that purpose, we can use arguments passed by reference, as in the function `duplicate` of the following example:

```
void duplicate(int& a, int& b, int& c)
{
  a *= 2;
  b *= 2;
  c *= 2;
}
…Button1Click…
```

```
{
    int x = 1, y = 3, z = 7;
    duplicate(x, y, z);
    Edit1->Text = "x=" + IntToStr(x) + ", y=" + IntToStr(y) + ", z=" +
IntToStr(z);
}
```

The first thing that should draw your attention is that in the declaration of `duplicate` the type of each parameter was followed by an ampersand sign (`&`). This ampersand specifies that the corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference, we are not passing a copy of its value, but we are somehow passing the variable itself to the function, and any modification that we do to the local variables will have an effect on their counterpart variables passed as arguments in the call to the function.

To explain it in another way, we associate `a`, `b` and `c` with the arguments passed on the function call (`x`, `y` and `z`) and any change that we do on `a` within the function will affect the value of `x` outside it. Any change that we do on `b` will affect `y`, and the same with `c` and `z`.

That is why our program's output that shows the values stored in `x`, `y` and `z` after the call to function `duplicate`, shows the values of all three variables of `Button1Click` doubled.

If, when declaring the following function:

```
void duplicate (int& a, int& b, int& c)
```

we had declared it this way:

```
void duplicate (int a, int b, int c)
```

i.e., without the ampersand signs (&), we would have passed not the variables by reference, but a copy of their values instead, and therefore, the output of our program on the screen would have been the values of `x`, `y` and `z` without having been modified.

Passing by reference is also an effective way to allow a function to output more than one value. For example, here is a function that outputs the previous and next numbers of the first parameter passed.
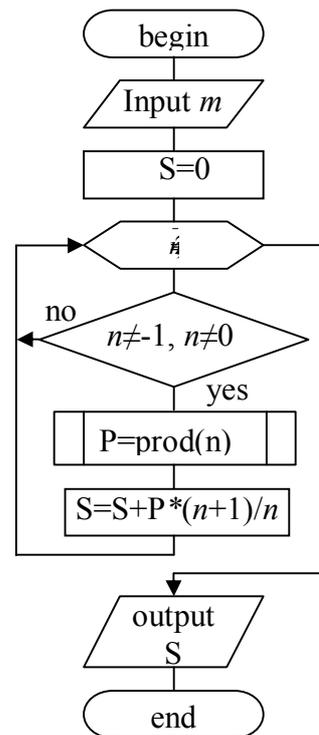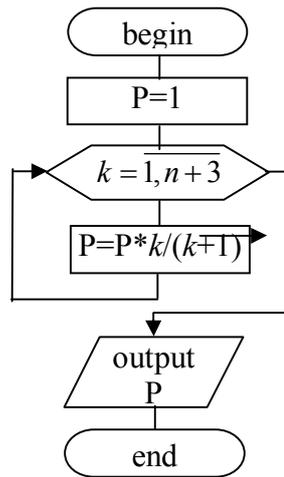
```
void prevnext (int x, int& prev, int& next)
{
  prev = x-1;
  next = x+1;
}
..Button1Click..
{
  int x = 100, y, z;
  prevnext (x, y, z);
  Edit1->Text = "Previous=" + IntToStr(y);
  Edit2->Text = "Next=" + IntToStr(z);
}
```

Example 1.11 can be written with a function that calculates the product.

Example 2.1: Calculate the value of $S = \sum_{n=-2}^{m} \frac{n+1}{n} \prod_{k=1}^{n+3} \frac{k}{k+1}$, enter from the screen the value of m.
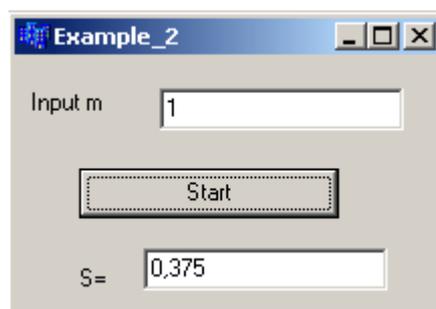
These are the schemes of the function prod and Button1Click

Flowchart 1 (left)

```
      begin
        │
       P=1
        │
   ┌──────────────┐
   │  k = 1, n+3  │◄──┐
   └──────────────┘   │
        │             │
   P=P*k/(k∓1) ────────┘
        │
     output
        P
        │
       end
```

Flowchart 2 (right)

```
        begin
          │
       Input m
          │
         S=0
          │
   ┌────────────┐◄──────┐
   │     n      │        │
   └────────────┘        │
          │              │
   no   ┌──────────────┐ │
   ◄────│ n≠-1, n≠0    │ │
        └──────────────┘ │
          │ yes          │
   ┌──────────────┐      │
   │   P=prod(n)  │      │
   └──────────────┘      │
          │              │
   S=S+P*(n+1)/n ────────┘
          │
       output
          S
          │
         end
```

The code of the program is:

//function prod calculates the product

```
double prod(int n)
{
  double p = 1;
  for (int k = 1; k <= n + 3; k++)
    if (k != -1 && k != 0) p *= (float) k/(k + 1);
  return p;              //return calculated value of product to Button1Click
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int n, k, m = StrToInt(Edit1->Text);
  float S = 0, P;
  for (n = -2; n <= m; n++)
    if (n != -1 && n != 0)
    {
      P = prod(n);       // call to the function prod
      S += (float)(n + 1) * p/n;
    }
  Edit2->Text = FloatToStrF(S, ffGeneral, 4, 3);
}
```

In this example, function `prod` has one parameter – integer number `n`. It returns a value of double type. The value of `n` is entered in `Button1Click`. Parameter `n` is passed to function `prod` by value.

Note that the variable `p` is local in function `prod` and its value is not accessible in `Button1Click`.

Example 2.2: Calculate the sum of the series $y = \sum\limits_{k=1}^{\infty} \dfrac{(-1)^k x^{2k}}{2k(2k-1)!}$, add only those terms of the series whose absolute values are greater than the given exactitude $\varepsilon = 10^{-4}$. Define the number of addends. Enter the value of `x` (-2<x<2) from the screen. Use a function to calculate the sum.

$$u_k = \frac{(-1)^k x^{2k}}{2k(2k-1)!}, \quad u_{k-1} = \frac{(-1)^{k-1} x^{2(k-1)}}{2(k-1)(2(k-1)-1)!} = -\frac{(-1)^k x^{2k}}{2x^2(k-1)(2k-3)!}$$

Recurrent:

$$r = \frac{u_k}{u_{k-1}} = -\frac{x^2}{2k(2k-1)}.$$

In a program to calculate u_2, we have to know the value of u_1 (addend with `k=1`) before it:

$$u_1 = \frac{(-1)^1 x^2}{2(2-1)!} = -\frac{x^2}{2}.$$
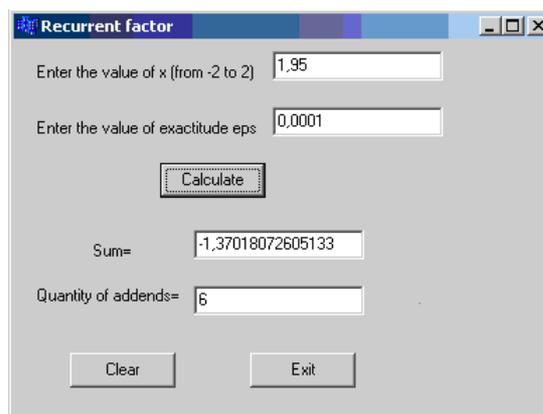
//*function calculates the sum*

```
double sum (double x, double eps, int &k)
{ double u, r, y;
```

```
      u=-x*x/2; y=u;          //calculate the first addend and output it in
```
Memo1
```
      do
      {k++;                   //increment k to calculate the next addend
       r = -x*x/(2*k*(2*k-1));     //calculate the recurrent factor
       u *= r ;   //calculate the addend
                    //(multiply the previous addend by recurrent factor)
       y += u;                //add the calculated addend to sum
      }while(fabs(u)>=eps);    //the loop continues while fabs(u)
                               //is greater than eps
      return y;                //return calculated value to Button1Click
    }
```

*//Button1Click inputs **x** and **eps**, calls to the function **sum** and outputs the result*

```
    void __fastcall TForm1::Button1Click(TObject *Sender)
    { int k = 0;
      float x, Y, eps;
      Memo1->Clear();
      x = StrToFloat(Edit1->Text);
      eps = StrToFloat(Edit2->Text);
      Y = sum(x, eps, k);      // call to function sum
      Edit3->Text = FloatToStr(Y);
      Edit4->Text = IntToStr(k);
    }
```

In this example function `sum` has three parameters – double numbers `n` and `eps` and integer number `k` It returns a value of double type. Parameters `n` and eps are passed to function `sum` by value. Parameter `k` changes in function `sum` and its new value is used in `Button1Click`. In such cases, the parameter is usually passed by reference (with & - symbol) and not by value.

3. Arrays

If you want to use a group of objects that are of the same kind, C++ allows you to identify them as one variable.

An **array** is a group of values of the same data type. Because the items are considered in a group, they are declared as one variable, but the declaration must indicate that the variable represents various items. The items that are part of the group are also referred to as members or **elements** of the array.

Declaration of array:

```
DataType ArrayName[Dimension];
```

The declaration of array starts by specifying a data type, the `DataType` in our syntax. This indicates the kind of values shared by the elements of the array. It also specifies the amount of memory space that each member of the array will need to store its value. Like any other variable, an array must have a name, which is the `ArrayName` in our syntax. The name of the array must be followed by an opening and closing square brackets "`[]`". Inside of these brackets, you have to type the number of items that the array is made of; that is the `Dimension` in our syntax.

Examples of declaration:

```
int student [10];       //array of 10 integer elements
double a [5];           //array of 5 double elements
float b [5];            //array of 5 float elements
```

When declaring an array, before using it, we saw that you must specify the number of members of the array. This declaration allocates an amount of memory space to the variable. The first member of the array takes a portion of this space. The second member of the array occupies memory next to it:

```
     a[0]  a[1]  a[2]  a[3]  a[4]
  a ┌─────┬─────┬─────┬─────┬─────┐
    │     │     │     │     │     │
    └─────┴─────┴─────┴─────┴─────┘
```
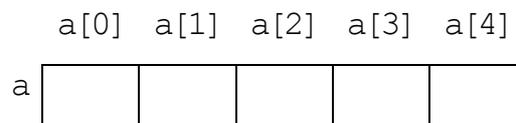
Each member of the array can be accessed using its position. The position is also referred to as an **index**. The elements of an array are arranged starting at index 0, followed by index 1, then index 2, etc. To locate a member, type the name of the variable followed by opening and closing square brackets. Inside the brackets, type the zero-based index of the desired member. After locating the desired member of the array, you can assign it a value, exactly as you would any regular variable.

For example, to assign the value 7.5 to the third element of array `b` declared above, we could write the following statement:

```
b[2] = 7.5;
```
and, for example, to pass the value of the third element of `b` to a variable called `a`, we could write:

```
a = b[2];
```

Therefore, the expression `b[2]` is for all purposes like a variable of type `float`.

Notice that the third element of `b` is specified `b[2]` since the first one is `b[0]`, the second one is `b[1]`, and therefore, the third one is `b[2]`. For the same reason, its last element is `b[4]`. Therefore, if we write `b[5]`, we would try to access the sixth element of `b` and will therefore exceed the size of the array.

To initialize the array at the time of declaration:

```
int student[5] = {0, 1, 4, 9, 16};
float b [] = {16, 2, 7.7, 40, 120.71};
int a[ ] = {1, 2, 3};
//a[0] = 1
//a[1] = 2
//a[2] = 3
int array [4] = {1, 2};
//array[0] = 1
//array[1] = 2
//array[2] = 0
//array[3] = 0
```

The numbers in `{}` are called initializers. If the number of initializers is less than the number of the array elements, the remaining elements are automatically initialized to zero. There must be at least one initializer in the `{}`. Such kind of expression can only be used in a declaration. You can't use "`{0, 1,...}`" in an assignment.

Instead of directly placing a figure such as 10 in the braces of the array declaration, it is better to place a constant variable. That way, when you want to change the array size, you only need to change it in one place.

```
const int size = 10;
int array[size];
```

The other reason is to avoid hard-coded numbers: if number 10 frequently appears in the program and other irrelevant 10 happens to appear, it can mislead the reader those tens have something to do with each other.

Only a constant variable can be used as array size. Therefore, you can not make the array size dynamic by inputting an integer from keyboard at run time and use it as array size. You cannot even use a non-constant variable assigned deterministically in the code, but you can `#define` preprocessor directives for array size.

In C++, an array is just an address of the first array element in memory. Declaring the size of the array can only help compiler to allocate memory for the array, but the compiler never checks whether the array bound or size is exceeded:

```
int a[3] = {10, 100, 27};        //we may use indexes 0, 1, 2
a[4]=3;           //if we try to use index 4, the compiler will not notice this
error
```

The problem that will happen if you exceed the array bound is: because the compiler was told that the array was only consisting of 3 elements, so it may have put other variables in the succeeding memory locations. Therefore, by declaring an array of 3 elements and then putting a value into the 4th element, you may have overwritten other variables and produced very serious logical errors which are very difficult to find out.

The size of an array should be carefully observed.

We usually use `for` loop to process arrays.

Input array from Memo

```
for(int i = 0; i < size; i++)
    a[i] = StrToInt(Memo1->Lines->Strings[i]);
```

Input array from horizontal StringGrid

```
for(int i = 0; i < size; i++)
    a[i] = StrToInt(StringGrid1->Cells[i][0]);
```

Output array to Memo

```
for(int i = 0; i < size; i++)
    Memo1->Lines->Add(IntToStr(a[i]);
```

Output array to horizontal StringGrid

```
for(int i = 0; i < size; i++)
    StringGrid1->Cells[i][0]=IntToStr(a[i]);
```

Sum of array:

```
int sum=0;
for(int i = 0; i < size; i++)
    sum += a[i];
```

Very few operations are legal for arrays. Some valid operations with arrays are:

```
a[0] = x;
a[x] = 75;
b = a[x+2];
a[a[x]] = a[2] + 5;
```

No assignment. Use a loop to copy elements from one array to another or to assign to array elements some values.

No comparisons. Use a loop to compare elements of two arrays.

No arithmetic operations. Use a loop to perform arithmetic operations between two arrays.

Example 3.1: Input an array of 10 float elements and calculate the product of non-zero elements of this array.
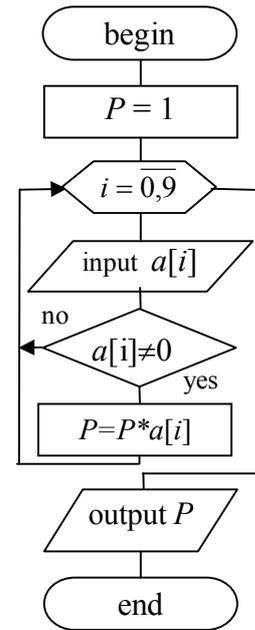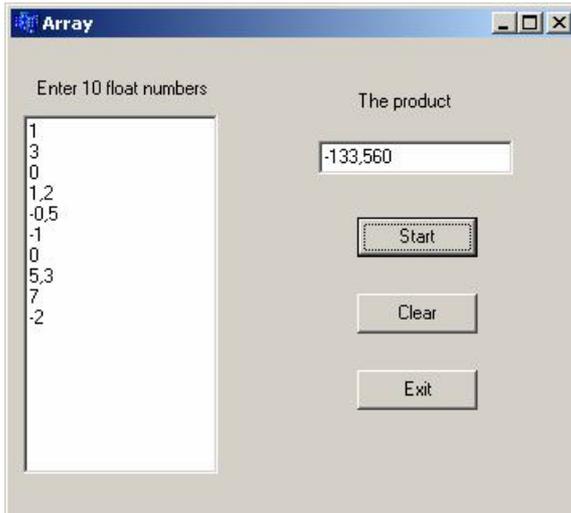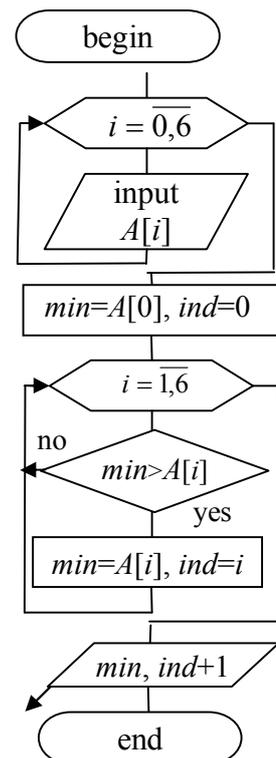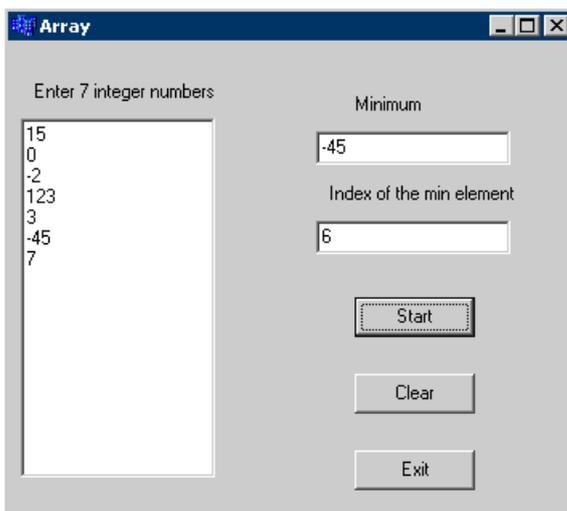
The program code:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float a[10], P = 1;
  for(int i = 0; i < 10; i++)
  { a[i] = StrToFloat(Memo1->Lines->Strings[i]);
    if(a[i] != 0)  P *= a[i];
  }
  Edit1->Text = FormatFloat("0.000", P);
}
```

Example 3.2: Input an array of 7 integer elements. Find the minimum of array elements and its index.

43

The program code:
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int A[15] ;  int i;
  for(i = 0; i < 7; i++)
    A[i] = StrToInt(Memo1->Lines->Strings[i]);
  int min = A[0], ind = 0;
  for(i = 1; i < 7; i++)
    if(min > A[i])
      {min = A[i];  ind = i;}
  Edit1->Text = IntToStr(min);
  Edit2->Text = IntToStr(ind+1);
}
```

Example 3.3: Input an array of 10 integer numbers. Create a new array of the first array elements divided by the sum of its elements with odd indexes.

To get a new array, at first we have to calculate the sum of elements with odd indexes. After that, we can sequentially divide all elements by sum and assign the result to elements of the new array. New array has the double type (after division).

Let's input the first array and output the new array in StringGrid (from Additional panel). This component is used to represent tables. It consists of Cells. Each Cell has Column and Row indexes. In this example StringGrid consists of 1 row and 10 columns. To use StringGrid in the program do the following:

- Place StringGrid on the form. Its Name property is StringGrid1. Enter the new name for it: SG1.

- Change the properties ColCount (количество столбцов) = 10 and RowCount (количество строк) = 1.

- Change the properties FixedCols (количество фиксированных столбцов) = 0 and FixedRows (количество фиксированных строк) = 0.

- Resize your StringGrid

- We want to input numbers into Cells of StringGrid Change the property Options – goEditing to true value and goTabs – also to true.

- Copy this StringGrid and Paste. Rename new StringGrid to SG2. This StringGrid is for the new array.
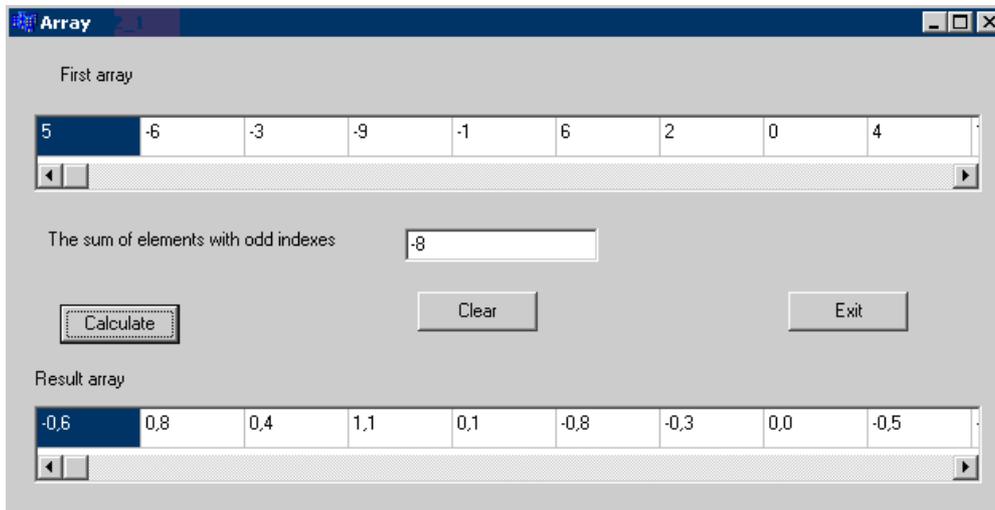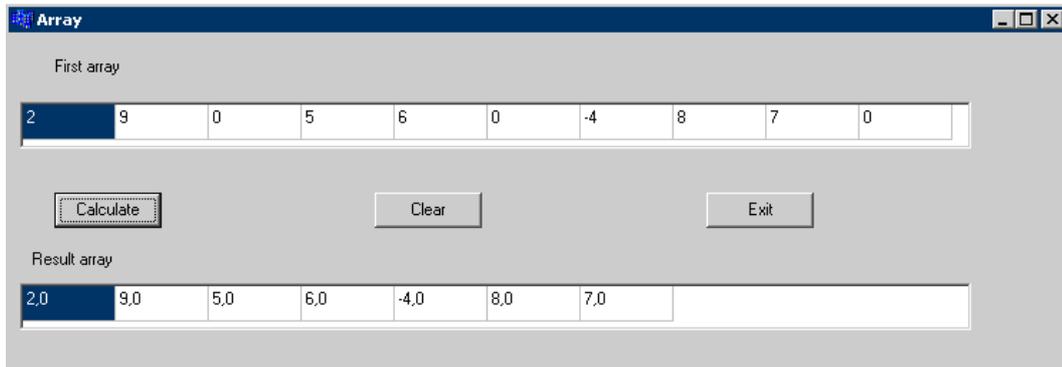

Text of the program:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  const int N = 10;
  int a[N], i, sum = 0; //a is the first array
  float b[N];              //b is the new array
  for (i = 0; i < N; i++)
    a[i] = StrToInt(SG1->Cells[i][0]);
  for (i = 0; i < N; i++)
    if (i%2 != 0) sum += a[i];
  Edit1->Text = IntToStr(sum);
  if (sum != 0)
    { for (i = 0; i < N; i++)
        b[i] = 1.*a[i]/sum;       //calculate b[i]
      for (i = 0; i < N; i++)
        SG2->Cells[i][0] = FormatFloat("0.0", b[i]);
    }
 else ShowMessage ("sum=0, cannot divide");
}
```

Example 3.4: Input an array of 10 or less double numbers. Create a new array of the non-zero elements.

We do not know the number of non-zero elements in the first array. Therefore, we declare a new array with 10 elements. In Example 3.3, indexes of the elements in both arrays were equal. In this example, indexes are not equal because zeroes will be dropped.

Text of the program:
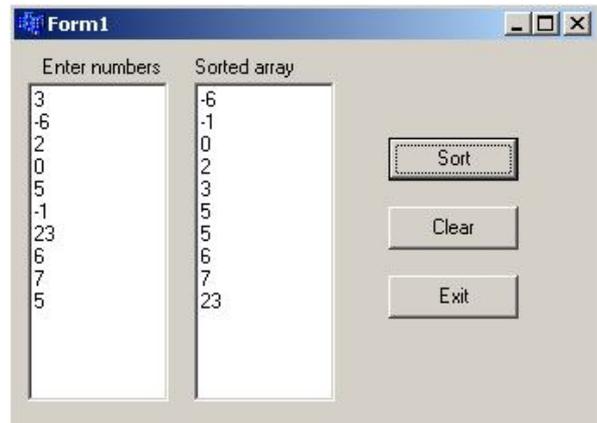
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  const int N = 10;
  float a[N], b[N];
  int  i, j = 0, sum = 0;
  for (i = 0; i < N; i++)
   { a[i] = StrToInt(SG1->Cells[i][0]);
     b[i] = 0;                //elements of the new array are at first 0
   }
  for (i = 0; i < N; i++)
     if (a[i] != 0) {
      b[j] = a[i];//if element a[i] is non-zero, we assign it to b[j]and
increment j
       j++;}
  SG2->ColCount = j;      //now we know the real number of cells in SG2
  for (i = 0; i < j; i++)
     SG2->Cells[i][0] = FormatFloat("0.0", b[i]);
}
```

SORTING

The problem is to place array elements in ascending order.

The technique: Make several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or values are identical), do nothing. If a pair is in decreasing order, swap the values. This algorithm is called bubble-sorting.

Assume you have an array a[10]. First, the program compares a[0] to a[1], then a[1] to a[2], then a[2] to a[3], and so on, until it completes the pass by comparing a[8] to a[9]. Although there are 10 elements, we need only 9 comparisons. On the first pass, the largest value is guaranteed to move to a[9]. On the second pass, the second largest value is guaranteed to move to a[8]. On the 9-th pass, the 9-th largest value will move to a[1], which will leave the smallest value in a[0].

The advantage of bubble sort algorithm is that it is easy to program. And the disadvantage is that it runs slowly, not appropriate for large arrays.

Text of the program:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  const int SIZE = 10;
  int a[SIZE];
  int i, temp;                    //temp will be used in swap
  //Input the array to sort
  for ( i = 0; i < SIZE; i++ )
     a[i]=StrToInt(Memo1->Lines->Strings[i]);
  //Sorting
  for (int j = 0; j < SIZE - 1; j++)   //repeat SIZE-1 times
  {
     for (i = 0; i < SIZE - 1; i++)     //for each element (except the last)
     {  if (a[i] > a[i + 1])//if thet element the next element  are in wrong order
        {  temp = a[i];                 // swap them
           a[i] = a[i + 1];
           a[i + 1] = temp;
```

47

```
        }
      }
   }
   //Output the sorted array
   for ( i = 0; i < SIZE; i++ )
      Memo2->Lines->Add(IntToStr(a[i]));
}
```

This algorithm can be optimized: repeat passes until the array becomes sorted (there can be less than SIZE-1 passes). We may write `do-while` loop instead of first `for`, and the condition is: while array is not sorted. We need to indicate in a special variable whether the array is sorted or not. We may count how many times we replaced elements during this pass, and a non-zero value shows that array is not sorted.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  const int SIZE = 10;
  int a[SIZE];
  int i, temp;
  int counter;                    //number of replacements on each pass
  for ( i = 0; i < SIZE; i++ )
     a[i] = StrToInt(Memo1->Lines->Strings[i]);
  do{                             // passes
     counter = 0;                 // before each pass counter is 0
     for (i = 0; i < SIZE - 1; i++)
     {  if(a[i] > a[i + 1]) // compare elements and if they are in wrong
order
        {  temp = a[i];           // swap them
           a[i] = a[i + 1];
           a[i + 1] = temp;
           counter++;             //and increment the counter
        }
     }
  }while(counter > 0);   //if the array is sorted, the counter will be 0 after
the pass
  for (i = 0; i < SIZE; i++)
     Memo2->Lines->Add(IntToStr(a[i]));
}
```

Example 3.5: Input an array of 11 or fewer integer numbers in `Memo`. Write function to sort these numbers in ascending order.

The elements of the array will change after sorting, and we want to save this changing. Arrays are passed to functions by reference, therefore it is **not** necessary to write symbol **&** before the array's name in function parameters.
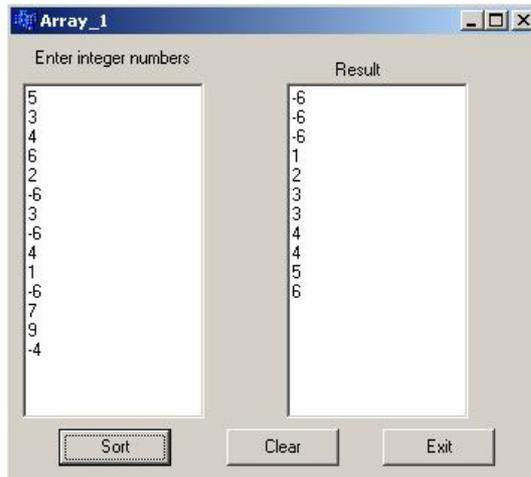
The program code:
```
sort(int a[ ], int n)
{ int i, j, tmp;
  for (i = 0; i < n - 1; i++)
    for (j = i + 1; j < n; j++)
      if (a[i] > a[j]) // compare two elements
      { tmp = a[i];   // replace elements a[i] and a[j]
         a[i] = a[j]; // variable tmp is necessary for
         a[j] = tmp;   // temporary storage of a[i]
      }
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int n, i, a[11];
  n = Memo1->Lines->Count;
  if (n > 11) n = 11;
  for (i = 0; i < n; i++)
    a[i] = StrToInt(Memo1->Lines->Strings[i]);
  sort (a, cn);
  Memo2->Clear();
  for (i = 0; i < n; i++)
    Memo2->Lines->Add(IntToStr(a[i]));
}
```

# 4. Multi-dimensional arrays

An array may have more than one dimension (i.e., two, three, or higher). The organization of the array in memory is still the same (a contiguous sequence of elements), but the programmers perceived organization of the elements is different.
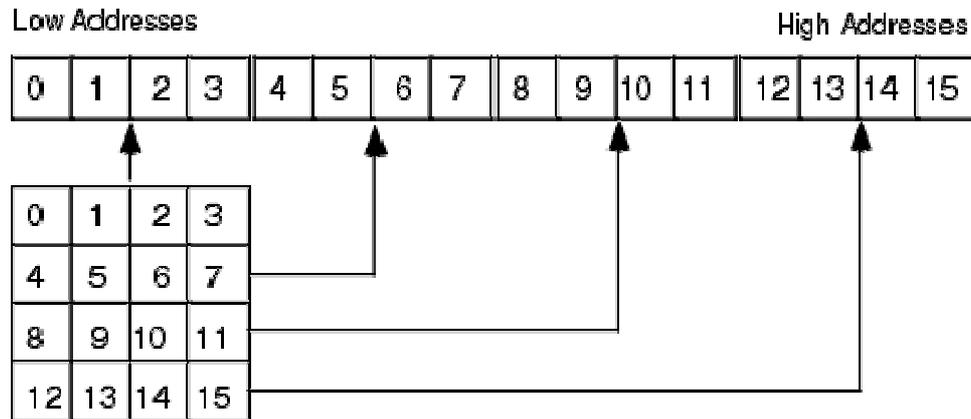


For example, suppose we wish to represent the average seasonal temperature for three Ukrainian cities (see Table 4.1).

Table 4.1  Average seasonal temperature.

|        | Spring | Summer | Autumn | Winter |
|--------|--------|--------|--------|--------|
| Odessa | 10     | 22     | 12     | 0      |
| Kiev   | 9      | 20     | 8      | -3     |
| Lviv   | 8      | 17     | 8      | -2     |

This may be represented by a two-dimensional array of integers.

## Two-dimensional arrays

A **two-dimensional array** is a collection of components, all of the same type, structured in two dimensions, (referred to as **rows** and **columns**). Individual components are accessed by a pair of indexes representing the component's position in each dimension.

The common convention is to treat the first index as denoting the row and the second as denoting the column.

Syntax:

```
DataType ArrayName [Rows][Columns] ;
```

Examples of declaration:

```
int table[5][3];        // 5 rows, 3 columns
// (row variable changes from 0 to 4, column variable changes from 0 to 2)
float a[3][4];          // 3 rows, 4 columns
// (row variable changes from 0 to 2, column variable changes from 0 to 3)


double b[52][7];        // 52 rows, 7 columns
// (row variable changes from 0 to 51, column variable changes from 0 to 6)
```

Average seasonal temperature for three Ukrainian cities:

```
int seasonTemp[3][4];
```

The organization of this array in memory is as 12 consecutive integer elements. The programmer, however, can imagine it as three rows of four integer entries each.

We can access elements of a two-dimensional array by 2 indexes (row and column). For example, Odessa's average summer temperature (first row, second column) is given by `seasonTemp[0][1]`.

Initializing a two-dimensional array in declaration (works for small arrays) includes sets of braces for each row. Example:

```
int table[2][3] = {{14, 3, -5}, {0, 46, 7}};
```

Processing a multidimensional array is similar to a one-dimensional array, but uses nested loops instead of a single loop.

We usually use a `StringGrid` component for two-dimensional arrays.

Input of a float array `a` with 3 rows and 4 columns declared above from StringGrid:

```
for(int i = 0; i < 3; i++)
  for(int j = 0; j < 4; j++)
    a[i][j] = StrToFloat(StringGrid1->Cells[j][i]);
```

The inner `for` loop totals the elements of the array one row at a time. It fills all columns of a row. The outer `for` loop increments the row index after each iteration. When the outer loop starts with the row index 0, the inner loop is executed the number of times equal to the number of columns, i.e. 4 in our program. Thus, the first row is completed for the 4 columns with positions [0,0], [0,1], [0,2] and [0,3]. Then the outer loop increments the row index to 1, and the inner loop is again executed, which completes the second row (i.e. the positions [1,0], [1,1], [1,2] and [1,3]). Then the outer loop increments the row variable to 2 and the inner loop is again executed, which completes the second row (i.e. the positions [2,0], [2,1], [2,2] and [2,3]).

Output of array `a` to StringGrid:

```
for(int i = 0; i < 3; i++)
  for(int j = 0; j < 4; j++)
```

```
        StringGrid1->Cells[j][i] = FloatToStr(a[i][j]);
```
Sum of array `a` elements:
```
float sum = 0;
for(int i = 0; i < 3; i++)
  for(int j = 0; j < 4; j++)
    sum += a[i][j];
```
Sums of array `a` columns:
```
float sum[4];
for(int j = 0; j < 4; j++)
{ sum[j]=0;
  for(int i = 0; i < 3; i++)
    sum[j] += a[i][j];
}
```

C++ does not have a limit on the number of dimensions an array can have. Example of the three-dimensional array:
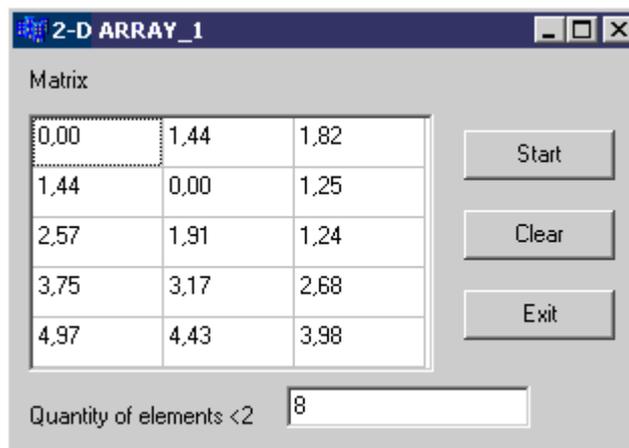```
int graph[10][20][30];
```

Example 4.1: Fill the matrix 5x3 with numbers by formula:

$$a_{ij} = \frac{\sqrt{\left|i^3 - j^2\right|}}{\ln(i + j + 1)}.$$

Calculate the number of elements, which are less than 2.

This formula can be calculated for all values of `i` and `j` except `i=0` and `j=0` simultaneously (let in this case $a_{ij}=0$).


```
#include <math.h>
```
//"Start" button
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
```

```
double a[5][3];
int i, j, k = 0;
```
//Calculation of matrix elements
```
for (i = 0; i < 5; i++)
  for (j = 0; j < 3; j++)
    if (i == 0 && j == 0) a[i][j] = 0;
    else a[i][j] = sqrt(fabs(pow(i,3)-j*j))/log(i+j+1);
```
//Output the elements
```
for (i = 0; i < 5; i++)
  for (j = 0; j < 3; j++)
    SG1->Cells[j][i] = FormatFloat("0.00", a[i][j]);
```
//Counting elements <2
```
for (i = 0; i < 5; i++)
  for (j = 0; j < 3; j++)
    if (a[i][j]<2) k++;
Edit1->Text = IntToStr(k);
}
```
//"Clear" button
```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
for (i = 0; i < 5; i++)
  for (j = 0; j < 3; j++)
    SG1->Cells[j][i] = "";
Edit1->Clear();
}
```
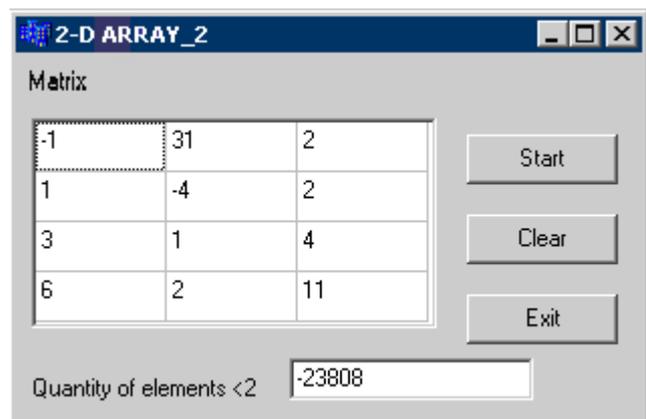
Example 4.2: Input an integer matrix 4x3. Calculate the product of elements which are multiple at least to one of its indexes.

Text of the program:
```
void __fastcall
TForm1::Button1Click(TObject
*Sender)
{
int a[4][3];
int i, j, k = 0;
long p=1;
```
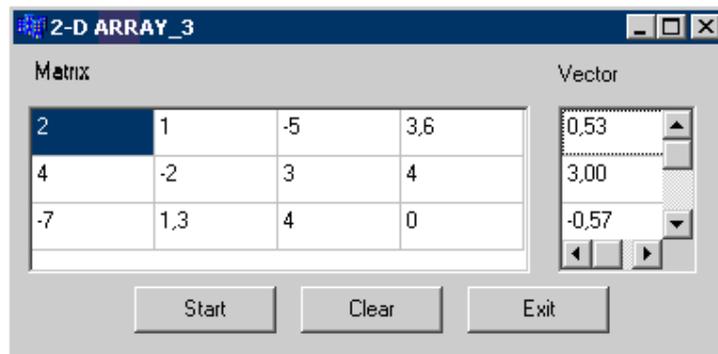*//Input matrix elements*

53

```
for (i = 0; i < 4; i++)
  for (j = 0; j < 3; j++)
     a[i][j] = StrToInt(SG1->Cells[j][i]);
```
//Calculate the product
```
for (i = 0; i < 4; i++)
  for (j = 0; j < 3; j++)
     if (i != 0 && a[i][j]%i == 0 || j != 0 && a[i][j]%j == 0)
        {p *= a[i][j];
         k++;}
if (k > 0) Edit1->Text = IntToStr(p);
else ShowMessage ("No numbers");
}
```

Example 4.3: Input float matrix 3x4. Create a vector of an average of row elements.


Text of the program:
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
float a[3][4],s, b[3];
int i, j, k;
long p = 1;
```
//Input matrix elements
```
for (i = 0; i < 3; i++)
  for (j = 0; j < 4; j++)
     a[i][j] = StrToFloat(SG1->Cells[j][i]);
```
//Calculate the averages and assign them to elements of new array
```
for (i = 0; i < 3; i++)
{ s = 0;
  for (j = 0; j < 4; j++)
    s += a[i][j];
  b[i] = s/3;
}
```

```
//Output the new array
for(i = 0; i < 3; i++)
  SG2->Cells[0][i] = FormatFloat("0.00", b[i]);
}
```

Example 4.3: Input two integer matrices 2x4 and 4x3. Multiply these matrices.

The result is a matrix with 2 rows and 3 columns.
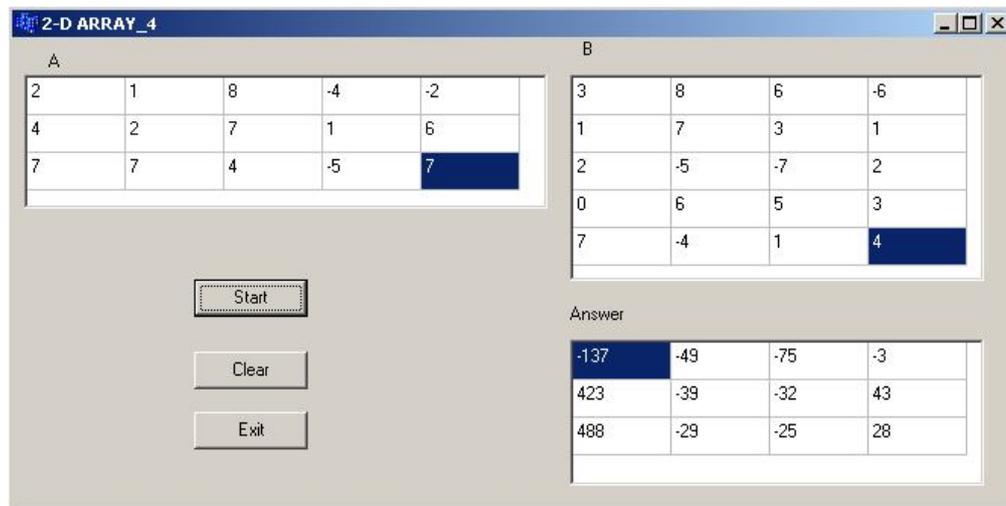
The program code:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  int a[3][5];
  int b[5][4];
  int result[3][4];
  int i, j, k;              // Loop counters in FOR loops
  int temp;                 // Used to build result
  // Read values for first array
  for (i = 1; i < 3; i++)
    for (j = 1; j <5; j++)
      a[i][j]=StrToInt(SG1->Cells[j][i]);
  // Read values for second array
  for (i = 1; i < 5; i++)
    for (j = 1; j <4; j++)
        b[i][j]=StrToInt(SG2->Cells[j][i]);

  //Calculate results in result array
  for (i = 1; i <3; i++)
```

```
    for (j = 1; j <4; j++)
    {
       temp = 0;
       for (k = 1; k <5; k++)
          temp += a[i][k] * b[k][j];
          result[i][j] = temp;
    }
//Now display the results
  for (i = 1; i <3; i++)
    for (j = 1; j <4; j++)
      SG3->Cells[j][i] = IntToStr(result[i][j]);
}
```