Computer science. Module 2
Lecture Notes
Table of contents

1. Loop statements

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

1.1 The FOR loop

For is a C++ construction that allows to repeat a statement, or set of statements, for a known number of times or while some condition is true. The syntax for the **for loop** is as follows:

for (initialization; condition of continuation; modification of parameters)
{statementBlock;}

where

- **initialization** declares and initializes the loop control variable, for example int i = 0;
- **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;
- **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 out the for loop:

i = 1 – initialization;
i <11 – condition of continuation;
i++ – modification of parameters;
Memo1->Lines->Add(IntToStr(i)+ “ ” + IntToStr(i*i)) – statement block.

The sequence of actions in this loop is:

1) At first ***i*** is declared as an integer variable and initialized to 1.
2) Then, we check up the condition i <11.
3) As it is true, we add to Memo1 the values of i and i*i (1 1)
4) After this the value of ***i*** increases by 1 (it becomes 2) and we return on the step 2 (checking the condition)

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

This example comes in many varieties:

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, this time we will simply add two 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 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));
```

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

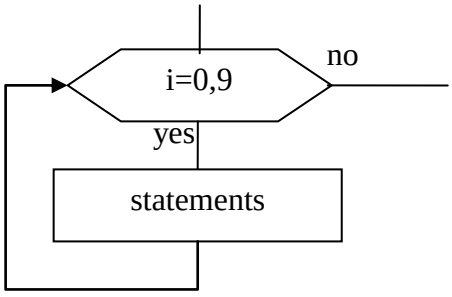
Clearly we need to use a loop, where some variable goes from 1 to 100. This will be variable ***i***. Except of it, we need a variable to store the **sum**. We'll call it sum. As the numbers are integer, the sum should be also integer. At first sum equals to 0. Then we add sequentially numbers one by one to it.

1) Start with sum = 0. If we do not assign 0 to sum, computer will assign any number to it (for example, negative number with 5 digits).
2) Start a loop with i = 1 (the numbers we need to sum begin from 1).
3) Add ***i*** to sum, and save the result back into sum (now sum is 1).
4) Increase i by 1 (now it is 2).
5) 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) 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) ... etc ...
10) until i reaches 100.

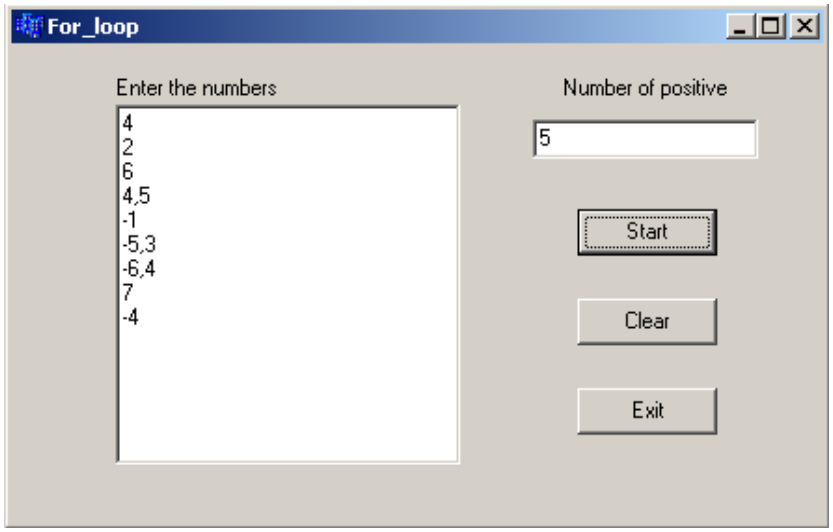
Here is the code that will accomplish this:

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

The scheme of **for** loop:



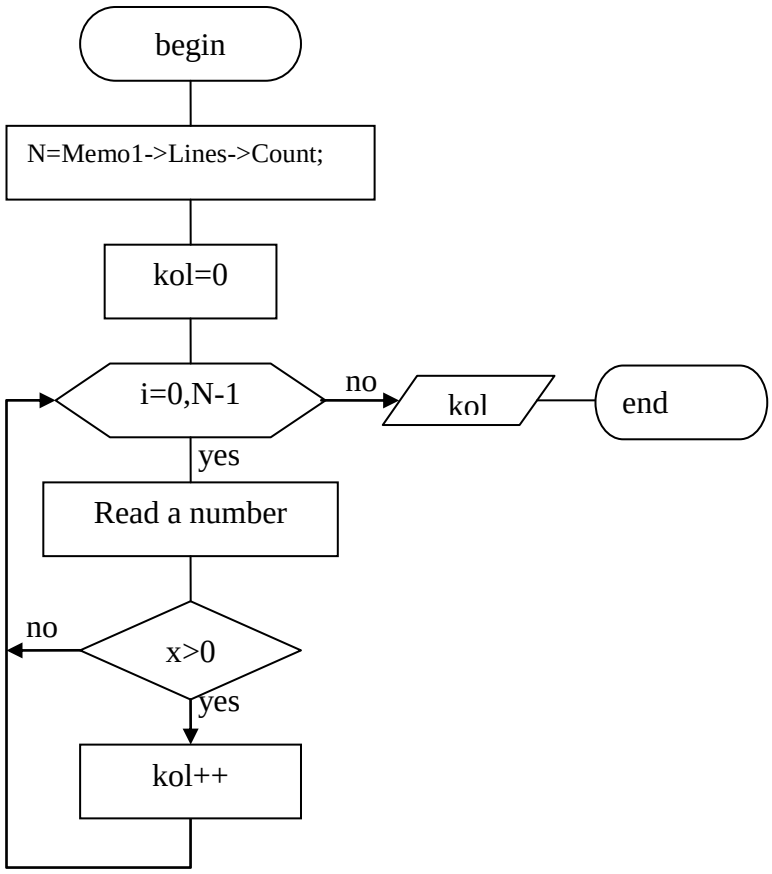
Example 1.5 Write a program that counts the number of positive doubles in Memo.



The code of the program

```
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 with number i
    if (x>0) kol++;               //if the number is positive, kol increases by 1
  }
Edit1->Text=IntToStr(kol);
}
```

The scheme is:



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

1.2 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 for repeating.

Its format is:

while (Condition)
{Statements;}

and its function is simply 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 the *Statements*, the *Condition* is checked again. AS LONG AS the *Condition* is true, it will keep executing the *Statements*. When or once the *Condition* becomes false, it exits the loop.

Let us write examples 1-3 with while loop:

Example 1.1_while

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

Example 1.2_while

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

Example 1.3_while

```
  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 will become 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;
while (sum<=20) {
   Memo1->Lines->Add(IntToStr(x));
   sum+=x;          //sum=sum+x;
   x++;
}
```

The sequence of actions:

1. At first x=1 and sum=0. As sum<20, then the condition is true and while begins to work.

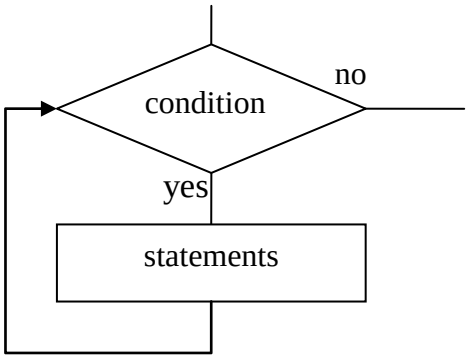
2. Number 1 is added to Memo1.
3. sum becomes 1.
4. x becomes 2
5. Return to condition. It is true. And we do 2.- 4. again and again.

We can show these actions 1-5 with the following table:

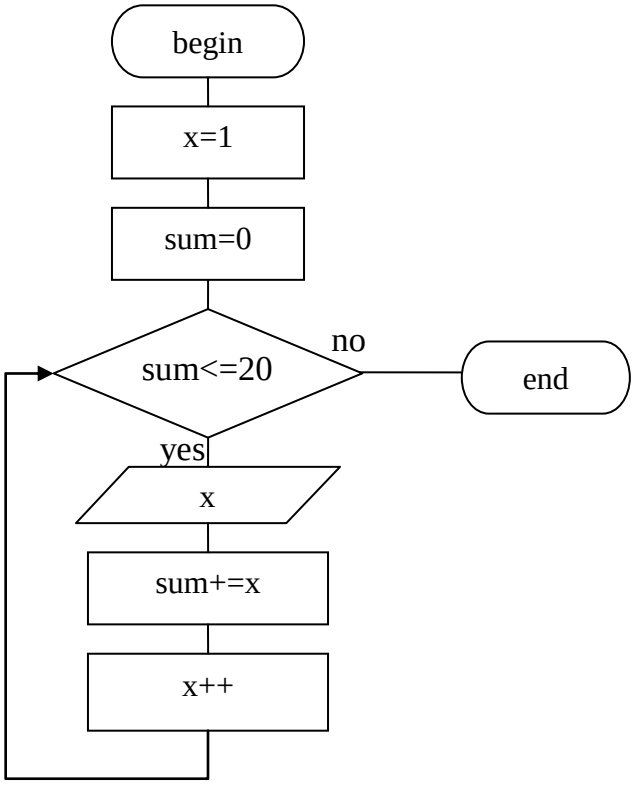
sum	0	1	3	6	10	15	21
x	1	2	3	4	5	6	7

From this table we see that when x=6 the sum becomes 21. When we come to condition it will be false and while stops.

The scheme of while is:



The scheme of the last example is:



1.3 The do-while loop (loop with postcondition)

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

Format:

do{
Statements;
} ***while*** (Condition);

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

Like the **if** and the **while** statements, the *Condition* being checked must be included between parentheses. The whole **do…while** statement must end with a semicolon.

The **do…while** statement can be used to insist on getting a specific value from the user.

The last example with do-while loop is:

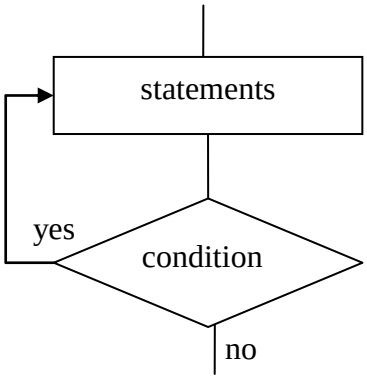
Example 1.6a Make a program to print out positive integer numbers from 1 until their sum will become greater than 20. We will add a number x to sum while sum will be less than or equal 20.

```
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);
```

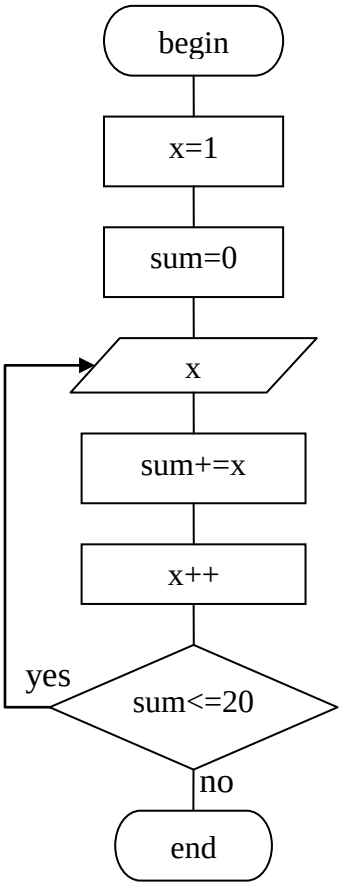
1. At first x=1 and sum=0.
2. Number 1 is added to Memo1 (without any conditions).
3. sum becomes 1.
4. x becomes 2
5. Check up the condition. It is true. And we repeat until sum becomes 21. Then the loop stops.

The **do-while** loop we use if we want to execute statement at first time without checking any condition.

The scheme of do-while is:



The scheme of the last example is:



Jump statements.

With ***break*** instruction we can leave a loop even when the condition for its ending is not fulfilled. It can be use to end an infinite loop, or to force it to end before its natural end.

The ***continue*** instruction causes the program to skip the rest of the loop in the present 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 the running program with an 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 will read numbers from Memo, until number zero is given. Find the smallest number.

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

1. First number is declared smallest and it's value is assigned to temp variable, which presents current minimum.
2. We search through other numbers and if one of them is smaller then our current minimum, we update our temp variable to show new current minimum.
3. After we processed all numbers, temp variable now stores the real minimum number.

Example

If there 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	

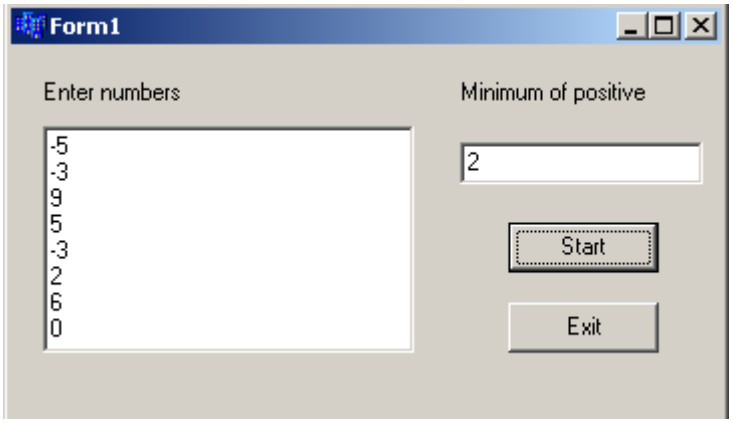
Answer: minimum number is -1.

The program code

```
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 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 will read numbers from Memo, until number zero is given. Find the smallest positive number.

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



The program code

```
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 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);
}
```

Alternative approach

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 positive number is the first and we must 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 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);

}
```

2. Functions.

2.1 Introduction to functions

Functions allow you to group a series of steps under one name. Imagine that you are baking chocolate chip cookies. You have to perform the following steps:

place two eggs in a bowl
add 1.5 c. butter to the eggs
...
bake cookies for 10-12 minutes at 375 degrees or until brown

In C++, and in most programming languages, you can give a name to a series of steps. Let's say we want to call this procedure "bake cookies". Then, our algorithm for baking cookies becomes:

bake cookies

We've just created a function to do the work for us. You still need to know how to bake the cookies. You still need to know that the first step is placing two eggs in a bowl, and that the second step is adding butter to the eggs. In this example, what you would do is write a function called `bakeCookies` (C++ won't let you put spaces in the names of functions or variables) that performs the series of steps above, and then whenever you wanted to bake cookies, you would *call* the function `bakeCookies`, which would execute the lines of code necessary to carry out the procedure.

Note: When we say you are *calling* the function `bakeCookies`, we do not mean that you are giving it the name `bakeCookies` - you've already done that by writing the function. We mean you are *executing* the code in the function `bakeCookies`. "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 she sees that you call a function `bakeCookies`, she knows immediately that you are baking cookies. If, on the other hand, she sees that your code places eggs in a bowl, then adds butter, etc., it will not be clear right away that you are trying to bake cookies. Lots of recipes start out with putting eggs in a bowl, and lots of recipes add butter to the eggs. By the time she reads the last line, she might realize that you are baking cookies, but only if she is familiar with the recipe. It's possible that she won't realize that you are baking cookies at all! The point is, functions make your code much easier to read.

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

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 procedure in order to invite your friend Hank.

write Hank's name on the invitation
write Hank's name and address on the envelope
place the invitation in the envelope
seal and stamp the envelope
drop the envelope in the mail

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 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. It seems unlikely that you'd be able to reduce this at all - each of your friends has got to have 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 does the following procedure:

write Hank's name on the invitation
write Hank's name and address on the envelope
place the invitation in the envelope
seal and stamp the envelope
drop the envelope in the mail

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 Hank eight times, and none of our other friends are going to receive invitations! Hank will get invited eight times because the function invites *Hank* 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 friend's name on the invitation
write friend's name and address on the envelope
place the invitation in the envelope
seal and stamp the envelope
drop the envelope in the mail

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

inviteToParty (friend = Hank)
inviteToParty (friend = Ann)
inviteToParty (friend = Alicia)
inviteToParty (friend = Whitney)
inviteToParty (friend = Greg)
inviteToParty (friend = Mi Young)
inviteToParty (friend = Flavio)
inviteToParty (friend = Brian)

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 inputs, do some stuff, and then produce an output.

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 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 to pass arguments to the function when it is called. The different parameters are separated by commas.
- `statements` is the function's body. It is a block of statements surrounded by braces `{ }`.

Function example:

```
//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 to `addition` passing two values: `5` and `3`, that correspond to the `int a` and `int b` parameters declared for 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 value 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` *plus* `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 the first place (in this case, `main`). At this moment the program follows it regular course from the same point at which it was interrupted by the call to `addition`. But additionally, because the `return` statement in function `addition` specified a value: the content of variable `r` (`return (r);`), which at that moment had a value of `8`. This value becomes the value of evaluating the function call.

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable `z` will be set to the value returned by `addition (5, 3)`, that is `8`. To explain it another way, you can imagine that the call to a function (`addition (5,3)`) is literally replaced by the value it returns (`8`).

2.2 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 with 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 one which 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 the integer variables *x*, *y* and *z* and the float variables *a*, *b* and *c* which are declared outside all functions are global variables and the integer variables *s*, *S* and *u* and the float variables *w* and *q* which are declared inside the function block are called local variables.

Thus the scope of global variables is between the point of declaration and the end of compilation unit whereas 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 Local variable S is known only in sum(), we can not use it in Button1Click. Local variables s, u, w, q are known only in Button1Click and we can not use them in sum().

Let us see an example which has number of local and global variable declarations with number of inner blocks to understand the concept of local and global variables scope in detail.

```
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;      //Local in Button1Click, global in if block
    if (a!=0)
     {
        int b;          //Local in block if
        b=25;
        a=45;           //Global in if block
        g=65;           //Global in program
     }
          a=50;
          ex( );
  }
```

Scope of *b* is till the first braces shaded

Scope of *a* is till the end of Button1Click brace

2.3 Functions with void type. The use of void

If you remember the syntax of a function declaration:

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

you will see that the declaration begins with a `type`, that is the type of the function itself (i.e., the type of the datum 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 absence of type: **void**.

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

…Button1Click…
{
  printmessage ();
}
```

`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!");
}
```

Although it is optional to specify `void` in the parameter list. In C++, a parameter list can simply be left blank if we want 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 would have been incorrect:

```
printmessage;
```

2.4 Arguments passed by value and by reference

Until now, in all the 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 to 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 in 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 from inside a function the value of an external variable. 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 call your attention is that in the declaration of `duplicate` the type of each parameter was followed by an ampersand sign (`&`). This ampersand is what specifies that their 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 in 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 `duplicate`, shows the values of all the three variables of `Button1Clik` 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 not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program 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 return more than one value. For example, here is a function that returns 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);
}
```

3. One-dimensional arrays

3.1 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. To declare an array, you must give it a name.

*DataType ArrayName***[***Dimension***]**;

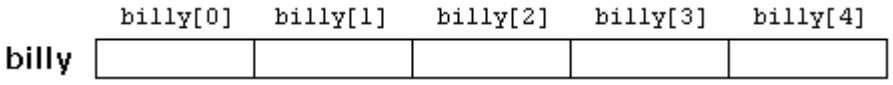
The declaration of array starts by specifying a data type, the DataType in our syntax. This indicates the kind of values shared by the elements of the array. It also specifies the amount of memory space that each member of the array will need to store its value. Like any other variable, an array must 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 must type the number of items that the array is made of; that is the Dimension in our syntax.

Examples of declaration:

int student[5]; *//array of 5 integer elements*

float billy [10]; *//array of 10 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:



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. This system of counting is referred to as "zero-based" because the counting starts at 0. To locate a member, type the name of the variable followed by an opening and closing square brackets. Inside the bracket, 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 store the value 7.5 in the third element of billy, we could write the following statement:
billy[2] = 7.5;
and, for example, to pass the value of the third element of billy to a variable called a, we could write:
a = billy[2];
Therefore, the expression billy[2] is for all purposes like a variable of type int.

Notice that the third element of billy is specified billy[2], since the first one is billy[0], the second one is billy[1], and therefore, the third one is billy[2]. By this same reason, its last element is billy[4]. Therefore, if we write billy[5], we would be accessing the sixth element of billy and therefore exceeding the size of the array.

To initialize the array when declaring:

```
int student[5] = {0, 1, 4, 9, 16};
float billy [] = { 16, 2, 7.7, 40, 120.71 };
int array[ ] = {1, 2, 3};
  //array[0] = 1
  //array[1] = 2
  //array[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 are 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 {}. But such kind of expression can only be used in declaration. You can't use "{0, 1,..}" in assignment.

Instead of directly placing a figure such as "21″ in the braces of the array declaration, it is better to place a constant variable. In such way when you need the change the array size you only need to change one place.

```
const int size = 21;
```

The other reason is to avoid magic number : if number 21 frequently appears in the program, and an other irrelevant 21 happens to appear, it will mislead the reader that this 21 has something to do with the former one.

Only constant variable can be used as array size. Therefore you can not make the array size dynamic by inputting an integer from keyboard in run time and use it as array size.

In C++ an array is just an address of the first array element. 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, 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 of 3 elements, so it may have put other variables in the succeeding memory locations. Therefore by declaring an array of 3 elements then putting a value into the 4th element, you may have overwritten another variables and produced very serious logic errors which is very difficult to find out.

Therefore, the size of an array should be carefully observed.

Some other valid operations with arrays:

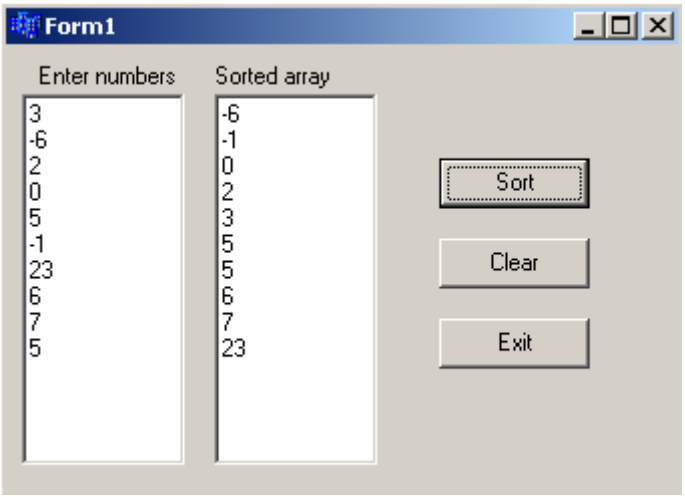
```
billy[0] = a;
billy[a] = 75;
b = billy [a+2];
billy[billy[a]] = billy[2] + 5;
```

3.2 Sorting

3.2.1 Bubble SORT

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.

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].

Bubble sort algorithm:

- Advantage: easy to program
- Disadvantage: runs slowly, not appropriate for large arrays

The program code

```
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) of array
      if ( a[i] > a[i + 1] ) {           // compare it with the next element and if they  are in wrong order
        temp = a[i];                     // swap them
        a[i] = a[i + 1];
        a[i + 1] = temp;
      }
    }
  }
 //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 condition is: while array is not sorted. We need in special variable to indicate, is array sorted or not. We may count how many times we replaced elements during this pass, and 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 it with the next element and if they  are in wrong order
            temp = a[i];           // swap them
            a[i] = a[i + 1];
            a[i + 1] = temp;
            counter++;            //and increase the counter
```

```
        }
      }
    }while(counter>0);      //if array is sorted counter will be 0 after the pass

    for ( i = 0; i < SIZE; i++ )
      Memo2->Lines->Add(IntToStr(a[i]));

}
```

4. Multi-dimensional arrays.

4.1 Two-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 major Australian capital cities (see Table 4.1).

Table 4.1 Average seasonal temperature.

	Spring	Summer	Autumn	Winter	Sydney
Melbourne	24	32	19	13
Brisbane	28	38	25	20

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

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 [ConstIntExpr] [ConstIntExpr] ;

Examples:

```
int table1[5][3];       // 5 rows, 3 columns
                        // (row variable changes from 0 to 4, column variable changes from 0 to 2)

float table2[3][4];     // 3 rows, 4 columns
                        // (row variable changes from 0 to 2, column variable changes from 0 to 3)

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

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

Example:

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

More Examples:

1. Average seasonal temperature for three major Australian capital cities:

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

Sydney's average summer temperature (first row, second column) is given by seasonTemp[0][1].
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.

2. Keep monthly high temperatures for all 50 states in one array.

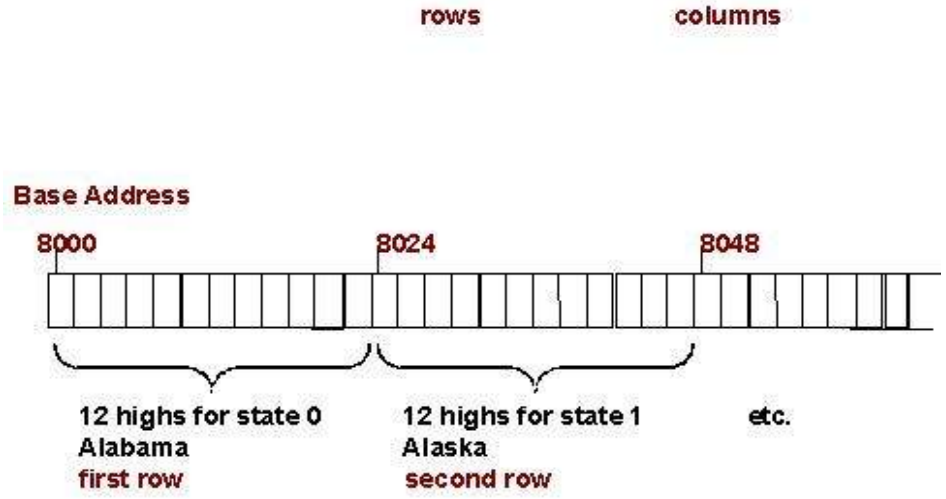
```
const int STATES = 50;
const int MONTHS = 12 ;
int stateHighs [STATES] [MONTHS];
stateHighs[2][7]= 105;        // might be Arizona's (state number 3) high for August (month number 8)
```

Keep highest temperatures for each day of the week in one array:

```
int hiTemp[52][7];
```

Declaration of a 2-dimensional array with 364 integer components (52 * 7); the array has 52 rows and 7 columns and contains the highest temperature for each day in a year. The rows represent one of the 52 weeks in a year, and the columns represent one of the 7 days in a week. For instance, hiTemp[2][6] represents the temperature for the seventh day in the third week.

The organization of this array in memory:



Processing a multidimensional array is similar to a one-dimensional array, but uses nested loops instead of a single loop. Program illustrates this by showing a function for finding the highest temperature in seasonTemp.

```
const int rows  = 3;
const int columns  = 4;

int seasonTemp[rows][columns] = {
  {26, 34, 22, 17},
  {24, 32, 19, 13},
  {28, 38, 25, 20}
};

int HighestTemp (int temp[rows][columns])
{
  int highest = 0;

  for (register i = 0; i < rows; ++i)
    for (register j = 0; j < columns; ++j)
  if (temp[i][j] > highest)
    highest = temp[i][j];
  return highest;
}
```

4.2 Multidimensional arrays

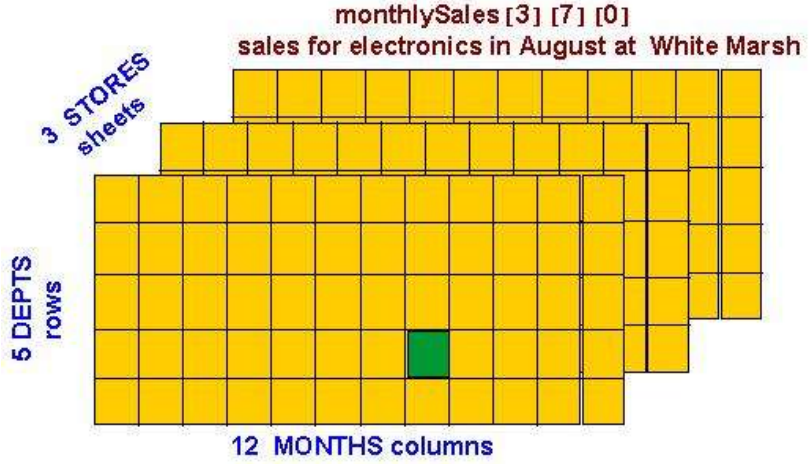
C++ does not have a limit on the number of dimensions an array can have.

Multidimensional array is a collection of components, all of the same type, ordered on N dimensions (N>=1). Each component is accessed by N indexes, each of which represents the component's position within that dimension.

Example: int graph[10][20][30];

Example of three-dimensional array (cube)

```
const int NUM_DEPTS = 5;  // men's, women's, children's, electronics, furniture
const int NUM_MONTHS = 12;
const int NUM_STORES = 3; // White Marsh, Owings Mills, Towson
int monthlySales [NUM_DEPTS] [NUM_MONTHS] [NUM_STORES] ;
```

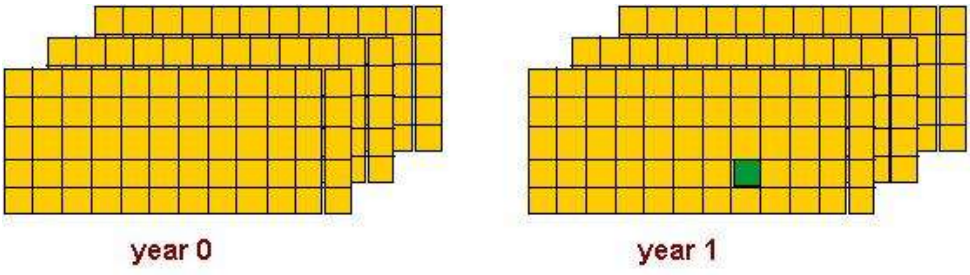


Print sales for each month by department:

```
COMBINED  SALES  FOR  January
DEPT #        DEPT NAME  SALES $
  0           Men's          8345
  1           Women's        9298
  2           Children's     7645
  3           Electronics   14567
  4           Furniture     21016
  ...
  ...
COMBINED  SALES  FOR  December
DEPT #        DEPT NAME  SALES $
  0           Men's         12345
  1           Women's       13200
  2           Children's    11176
  3           Electronics    22567
  4           Furniture      11230
```

Adding a 4-th dimension:

```
const NUM_DEPTS = 5; // men's, women's, children's, electronics, furniture
const NUM_MONTHS = 12;
const NUM_STORES = 3; // White Marsh,  Owings Mills, Towson
const NUM_YEARS  = 2;
int moreSales [NUM_DEPTS] [NUM_MONTHS] [NUM_STORES] [NUM_YEARS];
```



moreSales[3] [7] [0] [1]
for electronics, August, White Marsh, one year after starting year

4.3 Examples of programs with multi-dimensional arrays

Two-dimensional array example

Input array of results of students' exams. Find the student with lower grade.

The program code

```
const int STUDENTS = 3;     // number of students
const int EXAMS = 4;        // number of exams
……Button1Click…….{
  //declare and initialize array
        int studentGrades[STUDENTS][EXAMS] ;
        int i, j;
        for (i=0; i<STUDENTS; i++)
                for(j=0; j<EXAMS; j++)
                        studentGrades[i][j]=StrToInt(SG1->Cells[i][j]);
// Find the minimum grade
  int lowGrade = 100;
  for (int i = 0; i < STUDENTS; i++)
    for (int j = 0; j < EXAMS; j++)
      if (studentGrades[i][j] < lowGrade)
        lowGrade = studentGrades[i][j];
Edit1->Text=IntToStr(lowGrade);
}
```

0 means that the program finished normally and any other value means an error happened.