Computer science. Module 3
Lecture Notes
Table of contents

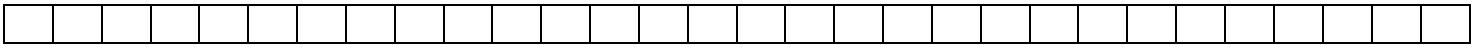
Lecture 3.1 Memory. Pointers. Dynamic arrays

Vocabulary

Alias – псевдоним
Argument passing – передача параметров
Deallocate, **release**, **free** – освобождать (память)
Dereference - разыменовывать
Memory allocation – выделение памяти
Pointer – указатель
Point to – указывать на
Reference - ссылка

Pointer. Reference.

We can think of computer memory as just a continuous block of cells:



Each cell has a **value** (contents) and **address**.

A **pointer** is the **address** of a memory location and provides an indirect way of accessing data in memory.

The declaration of pointers follows this format:

type * name;

where *type* is the data type of the value that the pointer is intended to point to.

This type is not the type of the pointer itself! But the **type of the data** the pointer points to. For example:

```
int * n;        //n is a pointer to int
float * g;      //g is a pointer to float
```

These are two declarations of pointers. Each one is intended to point to a different data type, but in fact both of them are pointers and both of them will occupy *the same amount of space in*. Nevertheless, the data, which they point to, do not occupy the same amount of space nor are of the same type: the first one points to an *int*, the second one to a float. Therefore, although these two example variables are both pointers which occupy the same size in memory, they are said to have different types: **int*** and **float*** respectively, depending on the type they point to.

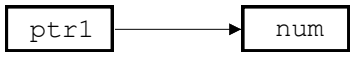
The **value** of a pointer variable is the **address** to which it points. For example, given the definitions

```
int *ptr1;      // pointer to an int
double *ptr2;   // pointer to a double
int num;        //int number
```

we can write:

```
ptr1 = &num;
```

The symbol **&** is the **address (reference)** operator; it takes a variable as argument and returns the memory address of that variable. The effect of the above assignment is that the address of integer variable num is assigned to ptr1. Therefore, we say that ***ptr1*** points to num.



The expression
*ptr1
dereferences ***ptr1*** to get to what it points to, and is therefore equivalent to ***num***. The symbol ***** is the **dereference** operator; it takes a pointer as argument and returns the contents of the location to which it points.

The asterisk sign (*) that we use when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the dereference, which is also written with an asterisk (*). They are simply two different things represented with the same sign.

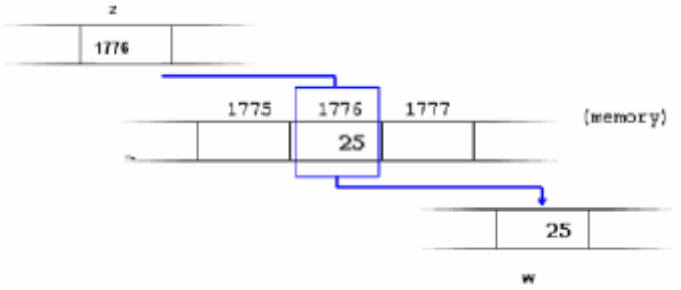
Example

```
int w;          //w is an integer variable
int * z=&w;     //z is a pointer to int; we assign to it an address of int variable w
```

If we write:

```
w = *z;
```

(that we could read as: "***w*** is equal to value, pointed by ***z***"), variable ***w*** would take the value 25, since ***z*** is 1776, and the value pointed by 1776 is 25.



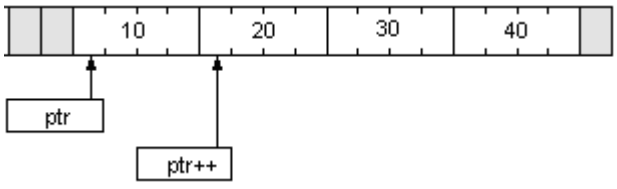
You must clearly differentiate that the expression **z** refers to the value 1776, while ***z** (with an asterisk * preceding the identifier) refers to the value stored at address 1776, which in this case is 25.

A pointer may be assigned the value 0 (called the **NULL** pointer). The NULL pointer is used for initializing pointers. NULL is a standard (predefined) C++ constant.

Pointer Arithmetic

In C++ one can **add** an integer quantity to or **subtract** an integer quantity from a pointer. This is frequently used by programmers and is called *pointer arithmetic*. Pointer arithmetic is **not** the same as integer arithmetic, because the outcome depends on the size of the object pointed to.

For example, ptr++ causes the pointer to be incremented, but not by 1. It is incremented by the size of type, which it points to. ptr-- causes the pointer to be decremented, but also by the size of type, which it points to.



Another form of pointer arithmetic allowed in C++ involves subtracting two pointers of the same type. For example:

```
int *ptr1 = &nums[1];
int *ptr2 = &nums[3];
int n = ptr2 - ptr1;            // n becomes 2
```

If we declare:

```
int a;          //variable a occupies in memory 4 bytes.
int *p = &a;    //p is a pointer to int, we can assign address of int variable a
```

Incrementation:

p++; *// the value of **p** increases by 4 bytes*

Decrementation:

p--; *// the value of **p** decreases by 4 bytes*

Addition:

p+=5; *// the value of **p** increases by 4*5=20 bytes*

On the whole, **p+=x** means that an address, which is the value of p, increases by **x*sizeof(x)** bytes. Standard function **sizeof** defines the size of variable or type, written in brackets.

Multiplication and division cannot be used in pointer arithmetic.

Pointers and arrays

In C++, arrays and pointers are essentially the same thing. Array name = pointer to first element (with index 0). By adding the statement

int hours[6];

we could then use the identifiers

hours[0] hours[1] hours[2] hours[3] hours[4] hours[5]

as though each referred to a separate variable. In fact, C++ implements arrays simply by regarding array identifiers such as "`hours`" as pointers. Thus if we add the integer pointer declaration

int *ptr;

to the same program, it is now perfectly legal to follow this by the assignment

ptr = hours;

After the execution of this statement, both "ptr" and "hours" point to the integer variable referred to as "hours[0]". Thus "hours[0]", "*hours", and "*ptr" are now all different names for the same variable. The variables "hours[1]", "hours[2]", etc. now also have alternative names. We can refer to them either as

*(hours + 1) *(hours + 2) ...

or as

*(ptr + 1) *(ptr + 2) ...

In this case, the "`+ 2`" is shorthand for "plus enough memory to store 2 integer values".

Difference between array and pointer:

– Declaring an array *creates storage* for the array

– Declaring a pointer *creates no storage*

Examples:

```
int a[10];      // creates storage for 10 ints, a is the address of the beginning of storage and address of
                //the first array element
int * p = a;    // p (and a) points to the first element (with index 0)
a[0]=7;         // set the first element to 7
(*a == 7)       // true (the same as (a[0]==7))
(*p == 7)       // also true
*a = 9          // change the first element to 9
*p = -3;        // change the first element to -3
*(a+1)=5;       // set the second element to 5, the same as a[1]=5;
a[2] = 8;       // change the third element to 8, the same is *(a+2)=8;
p[2] = 9;       // change the third element to 9, the same is *(p+2)=9;
```

Memory allocation

Automatic Allocation

Each variable has some “storage” associated with it – memory space that contains the contents (value) of the variable. Size of this memory depends on type of variable:

- ✓ pointers have 4 bytes
- ✓ ints have 4 bytes
- ✓ chars have 1 byte

Space is allocated **at compile time automatically** when variables are declared:

```
int a;          // allocates 4 bytes for the scope of a
int b[10];      // allocates 40 contiguous bytes, for the scope of b
```

Space exists as long as the variable is in scope

- ✓ space for variables in local scope goes away when the scope ends
- ✓ variables are deallocated at the end of blocks {} and functions
- ✓ global scope variables exist for the duration of the program

If an array is declared, enough space for the whole array is assigned, so:

int a[1000]

would use 2000 bytes (for 2 byte ints).

Wasteful if only a few of them are used in a particular run of the program. And problematic if more than 1000 elements are needed.

So there's a case for being able to dynamically assign memory as a program runs, based on the particular needs for this run.

Dynamic Memory

In addition to the program **stack** (which is used for storing global variables and stack frames for function calls), another memory area, called the **heap**, is provided. The heap is used for dynamically allocating memory blocks during program execution. As a result, it is also called **dynamic memory**. Similarly, the program stack is also called **static memory**.

The **new** operator can be used to allocate memory at *execution (run) time*. It takes the following general form:

pointer_variable = new data_type ;

Here, *pointer_variable* is a pointer of type *data_type*. The new operator allocates sufficient memory to hold a data object of type data_type and returns the address of this memory.

Memory for a single instance of a primitive type

```
int *ptr1 = new int ;
float *ptr2 = new float ;
```

Primitive types allocated from the heap via **new** can be *initialized* with some user-specified value by enclosing the value within parentheses immediately after the type name. For example,

```
int *ptr = new int ( 65 ) ; //allocates 4 bytes in dynamic memory and place in there number 65
```

Memory allocated from the heap does not obey the same scope rules as normal variables. For example,

```
void F ()
{
  int* a = new int;
  //...
 }
```

when F returns, the local variable a is destroyed, but the memory block pointed to by a is **not**. The latter remains allocated until explicitly released by the programmer.

The **delete** operator is used for releasing memory blocks allocated by new. It takes a pointer as argument and releases the memory block to which it points. For example:

```
delete a;
```

Dynamic Arrays

Often, we do not know the size of an array at compile time. If we try to write:

```
int N;
N=SrtToInt(Edit1->Text);
int a [N];
```

– compiler rejects this, because N is a variable. In this case we must use dynamic array.

Dynamic array is array, which:

- ✓ allocates in heap (dynamic memory) with new operator;
- ✓ may have a variable size (but its value must be obtained before array declaration).

To get memory for an array of some primitive type using new, write the keyword new followed by the number of array elements enclosed within square brackets.

int *a = new int [5] ;

After this declaration we can operate with dynamic array as with static one. For example, to input the values of array ***a*** from Memo1, we can write:

```
for ( int  i = 0 ; i < 5 ;  i++ )
        a [ i ] = StrToInt(Memo1->Lines->Strings[i]) ;
```

Note that it is not possible to initialize the individual elements of an array created when using new.

To destroy the dynamic array, we write

```
delete [] a;
```

The "[]" brackets are important. They signal the program to destroy all 5 variables, not just the first.

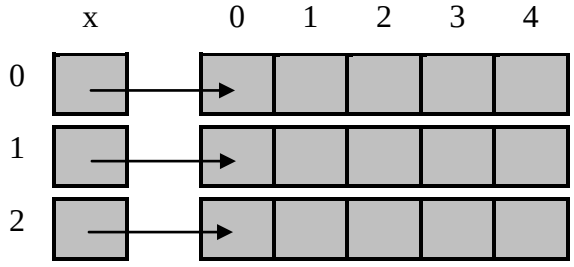
Dynamic arrays can be passed as function parameters just like ordinary arrays.

Dynamic Matrixes

We can create 2-d arrays of variable size based on pointers, using dynamic data. To create n x m array ***x***:

```
int** x;
x = new int* [n];
for(i = 0; i<n; i++)
        x[i] = new int[m];
```

This code allocates memory for ***n*** pointers to pointers to integers (pointers to matrix rows – vertical one-dimensional array ***x*** on the figure below). Variable ***x*** now points to start of this block (array) of pointers (x[0] is a pointer to the first matrix row, x[1] is a pointer to the second matrix row, etc.). For each of these pointers, allocates memory for ***m*** integers (for each row separately) and assign the address of the beginning of this memory to x[i]. The array elements **x[i]** now point to matrix rows. We can write x[i][j] to access an element in row i and column j.



To free memory, allocated this way for dynamic matrix, we must at first free memory allocated for each row separately and then free memory allocated for array of pointers to rows:

```
for(int i=0;i<n;i++)
        delete[] x[i];
delete[] x;
```

Example 1 of program with dynamic matrix:

Enter float matrix nxm and calculate the product of positive elements.

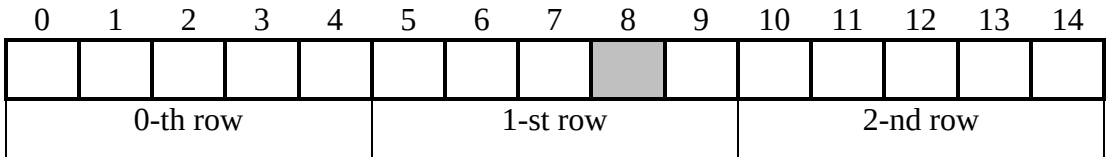
```
double p=1;
float **a = new float* [n];      //Allocate memory for array a of pointers to matrix rows

for (int i=0; i<n;i++)
  a[i]=new float[m];             //Allocate memory for each row (it contains m elements) separately
for (int i=0; i<n;i++)
  for (int j=0; j<m;j++)
     {a[i][j]=StrToFloat(StringGrid1->Cells[j][i]);
    if(a[i][j]>0)
      p*=a[i][j];
    }
Edit3->Text=FloatToStr(p);
for (int i=0; i<n;i++)
    delete []a[i];               //Free memory allocated for matrix rows
delete []a;                      //Free memory allocated for array a
}
```

There is one more approach to work with dynamic matrixes. You remember, that matrix elements are stored in memory in one line. We can allocate memory for the matrix in standard way. It will be the same as one-dimensional array of n*m elements:

```
float *a=new float [n*m];
```

If n and m are constants, we can write two dimensions separately. For example, float matrix 3x5:

```
float *a=new float [3][5];
```



To access matrix elements we have to calculate its index: a[i*m+j] (we cannot use two indexes separately as in static matrix). For example, highlighted element (in figure above) is a[1*5+3]=a[8].

To free memory, allocated in this way, we must write:

delete []a;

Example 1 of program with dynamic matrix (II approach of memory allocation):

Enter float matrix nxm and calculate the product of positive elements.

```
double p=1;
float *a = new [n*m];          //Allocate memory for all matrix elements, placed in one line
  for (int i=0; i<n;i++)
    for (int j=0; j<m;j++)
          {a[i*m+j]=StrToFloat(StringGrid1->Cells[j][i]);
     if(a[i*m+j]>0)
        p*=a[i*m+j]; }

Edit3->Text=FloatToStr(p);
delete [] a;
```

We can access matrix elements with pointers: a[i*m+j]. The same program with pointers:

```
double p=1;
float *a = new [n*m];          //Allocate memory for all matrix elements, placed in one line
  for (int i=0; i<n;i++)
    for (int j=0; j<m;j++)
          {*(a+i*m+j)=StrToFloat(StringGrid1->Cells[j][i]);
     if(*(a+i*m+j)>0)
        p*=*(a+i*m+j); }

Edit3->Text=FloatToStr(p);
delete [] a;
```

The pointer has following advantages:

- ✓ It allows to pass variables, arrays, functions, strings and structures as function arguments.
- ✓ A pointer allows to return structured variable from a function.
- ✓ It provides function which modify their calling arguments.
- ✓ It supports dynamic allocation and deallocation of memory segments.
- ✓ With the help of pointer, variable can be swapped without physically moving them.

Argument passing

There are three ways to pass arguments

- ✓ **Value** - give a copy of the argument to the function. Changes to the passed variable made by the function are not visible by the caller.
- ✓ **Reference** - allows the function to change the value. Changes made by the function are seen by the caller.
- ✓ **Address/pointer** - allows the function to change the value of passed variable and change the "pointer" itself.

The most common use of references is for function parameters. Reference parameters facilitates the **pass-by-reference** style of arguments, as opposed to the **pass-by-value** style which we have used so far. To observe the differences, consider the three swap functions in following program:

```
void Swap1 (int x, int y)                    // pass-by-value (objects)
{
        int temp = x;
        x = y;
        y = temp;
}
//-------------------------------------------------------------------------------
```

```
void Swap2 (int *x, int *y)          // pass-by-value (pointers)
{
        int temp = *x;
        *x = *y;
        *y = temp;
}
//---------------------------------------------------------------------------
void Swap3 (int &x, int &y)          // pass-by-reference
{
        int temp = x;
        x = y;
        y = temp;
        }
```

Although **Swap1** swaps x and y, this has no effect on the arguments passed to the function, because **Swap1** receives a copy of the arguments. What happens to the copy, does not affect the original.

Swap2 overcomes the problem of **Swap1** by using pointer parameters instead. By dereferencing the pointers, **Swap2** gets to the original values and swaps them.

Swap3 overcomes the problem of **Swap1** by using reference parameters instead. The parameters become aliases for the arguments passed to the function and therefore swap them as intended.

Swap3 has the added advantage that its call syntax is the same as Swap1 and involves no addressing or dereferencing. The following calls illustrate the differences:

```
int i = 10, j = 20;
Swap1(i, j);                                  //pass copies of i and j
Edit1->Text=IntToStr(i)+", "+IntToStr(j));  // i and j  do not swap (i=10, j=20)

Swap2(&i, &j);                                //pass addresses of i and j
Edit2->Text=IntToStr(i)+", "+IntToStr(j));  // i and j swap (i=20, j=10)

Swap3(i, j);
Edit3->Text=IntToStr(i)+", "+IntToStr(j));  // i and j swap again (i=10, j=20)
```

When run, it will produce the following output:

```
Edit1:  10, 20
Edit2:  20, 10
Edit3:  10, 20
```

References as function parameters offer three advantages:

1) They eliminate the overhead associated with passing large data structures as parameters and with returning large data structures from functions.
2) They eliminate the pointer dereferencing notation used in functions to which you pass references as arguments.
3) Like pointers, they can allow the called function to operate on and possibly modify the caller's copy of the data.

Lecture 3.2 User's libraries. Characters

Large C++ projects are usually elaborated by different programmers or even groups of programmers. On the other hand, a small simple project may have different (though also simple) parts of different nature (e.g. functions dealing with numbers and functions dealing with text strings). In both these cases it is very useful to divide a project into separate parts that can be compiled and tested separately.

A unit **U** is an independently compileable code module arranged so that the result of its compiling can be "attached" to any C++-project **P**. The developers of this project **P** need not to know the details of a unit **U** implementation. The only thing they need to know is: how to call functions included in unit **U**. It means: what are the names of unit's functions and of what types are the arguments of these functions.

In C++-Builder each unit consisting of 2 files:

– a header file, and

– a .CPP file.

A header file contains so-called functions' prototypes, which are namely functions' identifiers and lists of arguments preceded by corresponding types.

A .CPP file contains source codes of unit functions.

If a user intends to call in his project **P** some functions implemented in unit **U**, he must include prototypes of desired function to **P** (in order to supply **P** with the information about correct calls for each function). It is also necessary to have the translated bodies (binary codes) of desired functions, i.e. the result of .CPP-file compiling. At the same time it is not necessary to have the source C++-code of .CPP-file.

Standard C++ libraries, like "stdio" or "math", are organized as units. As you already know, to get access to functions contained in these libraries one must use the instruction #include (e.g. #include "math.h") before the first call to a function from corresponding library. You see that this instruction, also called "directive", contains a reference to the header file of a library – math.h in the above example.

In fact the directive #include causes the following actions of C++ compiling system. Let **P** be a C++ program with some occurrence of #include Z...Z (e.g. #include "math.h"). Then before a compiling of **P** this occurrence of #include Z...Z will be substituted for the text-contents of file Z…Z (e.g. of the header file math.h). It is obvious that after this the compiler will "know" all prototypes declared in Z…Z and hence can make correct calls for desired functions.

Every form in C++ Builder has an associated unit. E.g. the main form Form1 of Project1 has an associated unit Unit1 that is stored in the file pair Unit1.h and Unit1.cpp. The header file Unit1.h contains declarations of form's elements (labels, buttons, etc.) and prototypes of functions (like Button1Click). This file is usually built automatically during the user's construction of a form. The .CPP file Unit1.cpp contains "bodies" of all functions declared in Unit1.h and also any necessary auxiliary declarations (of constants, functions, etc.). This file you have already composed (modified) many times.

A programmer can easily compose his "own" units (other than units associated with forms) and use them as libraries of functions similarly to the usage of "math" or "stdio" libraries.

In the next lab work we will compose a simple C++ unit separate to the unit Unit1. Our unit will not depend on any C++-project, just the opposite, any C++ project will can call any desired function from it.

Consider the following task:

a) Create C++ header file and unit which include:

1) a function that finds the index of the last zero in its parameter: array of 7 integers;

2) a function that takes an array of 5 integers as input parameter and builds the output array including only even elements of the input.

b) Create a C++ project that enters input data and calls each of the above two functions from the developed unit.

We can start our work as usually do: launch C++ Builder ad get a "blank" Form1 (usually, with Unit1.h and Unit1.cpp corresponding to it). In order to start elaborating the new unit do the following:

1) In the main menu line of the Builder, choose:

File ⇒ New

and in a window that will open find "Unit", click twice.

2) You will see the following "layout" of a new unit's .CPP-file (it automatically gets name Unit2.cpp):

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

```
#pragma hdrstop

#include "Unit2.h"

#pragma package(smart_init)
```

You see here two directives #include, the first refers to header file of standard C++-Builder's library "vcl", and the second refers to the header file of currently elaborating unit. You see also another directive #pragma that I will not explain now (yet note that it is not a good idea to remove directives not already explained).

3) In order to see the layout of a header file (Unit2.h) of our new unit do the following:

a) Click RIGHT button of the mouse somewhere in Unit2.cpp-window;

b) In the appeared menu choose Open Source/Header File.

You will see the following text:

```
#ifndef Unit2H
#define Unit2H
//---------------------------------------------------------------------------
#endif
```

You see new directives #ifndef, #define and #endif here. I will explain them later, and now only note that all your modifications of the new header file must be placed AFTER #define and BEFORE #endif .

Though in previous lab works we have make no modifications to automatically built header file Unit1.h, now we must include prototypes of our task's functions to the new unit (otherwise these functions will be not accessible from another projects).

But at first let us change the "default" name of our new unit:

4) Choose

File ⇒ Save as

and type some new unit name, e.g. MyLib. You will see that both Unit2.h and Unit2.cpp will be changed for MyLib.h and MyLib.cpp respectively.

5) In connection to our task's items, I will form the following header file MyLib.h :

```
#ifndef Unit2H
#define Unit2H
//-------------------------------------------------
const int N=7;
typedef int AType[N];
int IndOfLastZero(AType A); //1-st function prototype

const int N2=5;
typedef int BType[N2];
void GetEvenElems(BType B, BType &Res, int &LenRes);
                    //2-nd function prototype
#endif
```

6) The next step is to fill the source file MyLib.cpp. Let me propose the following implementations of the above functions' prototypes:

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

#include "Unit2.h"

//-------------------------------------------------

#pragma package(smart_init)
```

```
int IndOfLastZero(AType A)
{ int j;
  for(j=N-1; j>=0; j--)
     if(A[j]==0) return j;
  return -1;
}

bool AnEvenNumber(int n)
{ if((n==0)||(n%2==1))
       return false;
  else return true;
}

void GetEvenElems(BType B, BType &Res, int &LenRes)
{ int j,
      k=0;
  for(j=0; j<N2; j++)
     if( AnEvenNumber(B[j]) )
       {Res[k]=B[j]; k++;}
  LenRes=k;
}
```

7) Finally we must compose a project that will call the above two task's function and check their correctness. Let me propose the following form and corresponding calls in Button1Click and Button2Click:

```
#include "Unit2.h"

void __fastcall TForm1::Button1Click(TObject *Sender)
{AType A;
 int j, ind0;
 for(j=0; j<N; j++)
    A[j]=StrToInt(Memo1->Lines->Strings[j]);
 ind0=IndOfLastZero(A);
 Edit1->Text=IntToStr(ind0);
}
//---------------------------------------------------

void __fastcall TForm1::Button2Click(TObject *Sender)
{BType B,
       C;
 int j,
     LenC;
 Memo2->Clear();
 for(j=0; j<N2; j++)
    B[j]=StrToInt(Memo1->Lines->Strings[j]);
 GetEvenElems(B, C, LenC);
 if(LenC==0)ShowMessage("No even items");
 else
    for(j=0; j<LenC; j++)
       Memo2->Lines->Add(IntToStr(C[j]));
}
```

Characters (symbols)

I think it is obvious to all you that a number and a symbol (or symbols) that denotes this number are different things. For example, the number 5 can be denoted by either the digit 5, or the Rome V, or simply the word “five”.

So symbols form a special type of data that differs from numbers. In fact there are many symbolic types in C++, but one type – namely

char

– is the basic one. The constants of this type are plain symbols like A, я, + and so on. The variables having this type are declared in the following form:

```
char c, c1, plus, blank;
```

Being declared, the char-variables can get concrete values in a usual manner, for instance:

```
c='A';   c1='я';   plus='+';   blank=' ' ; etc.
```

So you see that we must use apostrophes in typing the char-type constants. Due to these apostrophes the compiler can distinguish symbols from variables (for example, digits from numbers denoted by them).

In the computer memory each char-type constant or variable occupies exactly one byte. So it is obvious that each symbol has its corresponding 8-bits code (or binary number). The smallest possible code is 00000000, and the biggest is 11111111. So it is clear that there exist exactly 256 different constants of the type char.

The decimal number being equal to the binary code (number) of a given symbol is usually called the decimal number of this symbol. Obviously it is the matter of a convention to assign concrete binary (respectively decimal) numbers to concrete symbols. In Windows operational system there accepted a convention that called ANSI coding table. It looks like follows:

Visual image of a symbol	Decimal number	Binary number
…		
the blank	32	00100000
…		
0	48	00110000
1	49	00110001
…		
A	65	01000001
B	66	01000010
…		
я	255	11111111

In operating systems different from Windows different code tables may be used. For instance, the Ms DOS system uses another code table named ASCII.

Symbols with decimal numbers from 0 to 31 are usually used for special purposes of operating system. Some of them (like symbol number 0) have no visual image. They all also have no corresponding keys on the keyboard. So if a programmer needs to put any of them in any source code he must use special representations. For instance:

'\0' – the symbol with zero number;

'\n' – the symbol with decimal number 10;

(both '\0' and '\n' are widely used in C++, as we will see soon.)

You see that back slash sign marks the usage of special representation of symbols. So it must be clear that this sign cannot be represented by his own icon, namely '\'. In order to compel the compiler to accept back slash as such, one need to use the representation '\\'.

In any case when you know the number of a given symbol, you can represent it through the hexadecimal code of this number. For instance:

'\x0A' – the symbol with hexadecimal number A (the same to decimal 10, so it is '\n');

'\x5C' – the symbol with hexadecimal number 5C (the same to decimal 92, it is back slash \);

Because of the fact that the decimal number of a symbol in ANSI table equals to binary number contained in corresponding byte, there exists a simple way to get this decimal number. All we need is to tell the compiler to treat this byte as being not of char-type, but of type integer. To do it, one can used so-called type cast operation:

```
char c;   c='?' ;  // or   c='\x5C';   etc.
int n;
n = int(c);    //type cast operation
```

But there appear two problems. The first is that the integer type values are by default signed ones, that means that the highest bit of their binary representations is treated as sign (0 as plus and 1 as minus). At the same time the number of a symbol is always positive (e.g., the number of 'я' is 255, while its binary code is 11111111).

To avoid the treatment of a character number as a signed integer we must use the type specifier unsigned:

```
n = (unsigned int)(c);      //we need parenthesis rounded the type name
                              because of the blank in it.
```

The second problem is that the base type int of C++ reserves for its values not 1, but 4 bytes. To avoid the treatment of a symbol as 4 bytes value we must use another integer type, namely __int8 (it occupies 8 bits, i.e. exactly 1 byte). So the correct type cast for getting the number of a symbol will be:

```
n = (unsigned __int8)(c);
```

Analogously we can, having the decimal number of any symbol, get its representation as a character by using the following type cast:

```
unsigned __int8 n;   n=33;   // or   n=233;   etc.
char c;
c = char(n);          //type cast operation
```

Note that the declaration of n as “unsigned __int8”, not as “int”, is important here.

The ANSI code table is arranged so that the order of Latin letters in it is the following:

ABCD…Zabcd…z

Likewise the order of Cyrillic letters is

АБВГ…Яабвг…я

and the order of symbols that are digits are

01234567890

The arrangement of code table induces a certain order between characters, namely the order “before” or “after”. Symbol S_1 stands before symbol S_2 just in the case when the number of S_1 is less than the number of S_2. So there is introduced a sign < for a relation “before” in the set of symbols, and we write:

```
'0' < '1' ;
'A' < 'B' ;
'Я' < 'я' ;
```

etc. The relations >, <=, and >= are introduced analogously. There exist also obvious relations S_1== S_2 and S_1!= S_2 between two symbols.

These features of the ANSI table give possibility to check whether a given character variable c contains a capital Latin letter, a small Cyrillic letter, an arbitrary Latin letter, a digit, etc.

Lecture 3.3 Strings

There exist many different ways to store and manage strings of characters (like words, sentences, texts and so on) in processes of program functioning. Previously you got acquainted with the simplest mode of keeping strings: treating them as the following pair:

1) character array that declared as having some constant length *L* treating as "maximum possible length of a string"

2) integer number *n* that belongs to the segment [0, *L*] and is treating as the "current length of a string".

This mode of managing strings requires from programmers to allocate much superfluous memory. Indeed, if you write a program that deals with words and sentences (e.g. <exempli gratia>, of natural language) you need to foresee the possibility of long sentences and allocate appropriate long memory for your character array, in spite of the fact that most (yet not all) natural sentences you meet are of short or average length.

So it had been invented other – more efficient – modes for strings' managing. You must note that these modes, though all being effective in saving memory for strings, use *different representations* of strings (so called string *types*). Due to differences of these types you may meet collisions if you will mix them in operations applies to strings in your program.

That does not mean that the mixing of string types is quite impossible. But you must take certain precautions to avoid mistakes and/or unexpected results in program output.

The original, "native" mode of saving character strings in the language C is using special *delimiter* character that marks *the end* of a string. This character is '\0', i.e. "zero-character" – the symbol with zero number in ANSI code table.

A string constant must be written in C or C++ text using the form:

"*sequence of characters that forms a string*"

So, unlike single symbols (enclosed in apostrophes), strings are enclosed in quotation marks. You need not to manifestly ending a string constant with the symbol '\0'. A C++ compiler do it itself whenever it translates a sequence of the form ".....".

On the other hand, if you evidently end the "traditional" C++ string with zero-character, e.g. "ABC\0", this string will be treated by a compiler as equal to "ABC". Note that when you including a special symbol (like '\0' or '\n') in a string constant, you should *not* use apostrophes: the constant "ABC'\n'" will be interpreted as: 1) ABC' in the first line, 2) the line-break, and 3) the apostrophe ' in the second line (you can check it by using the string "ABC'\n'" as a parameter in ShowMessage function).

It is important to note that

'A' and "A"

are constants of different types. Namely, the first occupies exactly 1 byte, but the memory allocated to the second is 2 bytes: the symbol 'A' and the symbol '\0' that follows it.

The order of symbols in ANSI table entails the lexicographical (alphabetical) order in the set of string constants. String constant S_1 is treated as being less than S_2 ($S_1 < S_2$) if S_1 stands before S_2 in an imaginary vocabulary that includes all possible "words" that can be constructed on the basis of ANSI table. So "ABC" < "ABD" < "aBC", and "ABC" < "ABCD". The relations <=, >, >=, ==, != are introduced by analogy.

It is allowed to assign string constant as an *initial* value to an array of characters declared as `char A[n]` (where `n` is some integer constant). For instance:

```
char A[10] = "Honey";
```

In this case the array A will contain 6 characters: the first 5 form the word Honey, and the sixth character will be '\0'.

It is also allowed to declare character array with a string initial value without fixing the length of this array:

```
char B[] = "Honey";
```

In this case memory allocated for the array B will be exactly 6 bytes length. Yet it is still impossible to change this length in run time. Surely, you can change some symbols in B. E.g., you can assign zero-character to its first element:

```
B[0] = '\0';
```

that causes treating B as *empty string* in spite of the fact that its "tail" elements are oney (these "tail" unused elements are usually called "garbage"). It is also possible to use using special C++ functions of strings' managing in order to change the content of B. For instance, the following function

```
...; strcpy(B,"Beer"); ...
```

will substitute Honey for Beer in B. But if you try to write a string of more than 5 characters (not counting the end '\0') to B, you will have the run-time error "Access violation".

In order to avoid errors of this type "in general" we need to allocate and free memory dynamically. It means that if some string containing in a C++ variable S changes its length, then there must be changed the amount of memory allocated to variable S. There are many ways of do it in C++, and different ways are usually require different *types* for variable S.

We will study below one of these variable types, namely AnsiString type. You must remember that this type *differs* from char A[n] or char A[] types.

You already know that to declare AnsiString variable, e.g. S, one must use the declaration:

```
AnsiString S;
```

After this declaration we may assign different string values to S, for instance:

```
S = "Honey";
...
S = "Beer";
```

etc. If such an assignment needs the *change of the length* of string value in S, it is accompanied by corresponding *changes of memory allocated to* S. These changes are done in run time, when the assignment operator gets the control. So the memory to S is allocated dynamically.

The type AnsiString is declared in vcl.h header file.

We will not explain here in details how AnsiString variables are kept in memory (to make correct explanation we need to learn the notion of *pointer* before). It will be enough now to say that together with the "body" of an AnsiString S the integer number equal to current length of S is saved too. Due to that even *empty* AnsiString S occupies 4 bytes of memory.

Note that the rights sides of the following operators:

```
S = "Honey";
S = "";
```

are *not* AnsiStrings but the "traditional" C++ strings, i.e. zero-ended arrays of characters. In the second operator right side "" denotes an array with only *one* element that is zero-character '\0'. This array occupies exactly 1 byte. During the assignment to S it is *converted* to 4-bytes representation of empty AnsiString. Analogous conversion takes place in the case of first assignment above.

Unlike zero-ended arrays of characters (and other string types of C++ that do not mentioned here) *elements of AnsiStrings are numerated starting from index* 1. So if you have the following code:

```
char B[] = "Honey";
AnsiString C = "Honey";
```

then the following Boolean expressions will be `true`:

```
B[0]==C[1]
C[1]=='H'
```

but the next are `false`:

```
C[1]==B[1]
C[0]==B[0]   // here you will have a run-time error
             // because C[0] does not exist at all
```

"Traditional" C++ string-managing functions, like `strcpy` or `strcmp`, will not work correctly with AnsiStrings. It means that you should not directly use *variables* of AnsiString type as parameters of these functions.

Vice versa, functions elaborated to manage AnsiStrings do not good for other string types, like zero-ended arrays of characters. In order to transfer zero-ended strings to functions with AnsiString-type parameters you should use type cast, like in below example:

```
char B[] = "Honey";
Edit1->Text=AnsiString(B)+"costs money";
```

Note that our previously composed function `CTypeToAnsiS` *cannot* be applied in this example because it deals with arrays of characters being *not zero-ended* ones (so it isn't applicable to `B`).

Standard functions applying to `AnsiString`-type arguments.

The list of the most important functions is following:

`+` – concatenation. Operator `S1=S2+S3;` says that string from `S3` must be appended to those from `S2` and the result saved in `S1`.

`==, !=, <, <=, >, >=` – logical operations. Expression `S1==S2` returns `bool`-type value; as well as expressions with other 5 operations.

Now let `S` be an AnsiString-type variable. Any following C++-Builder function `F` applying to `S` must be called in the form:

`S.F(`<additional parameters (if any) of `F`>`);`

(You will understand the motives of choosing such a form after you learn the notion of `class` of C++-Builder.)

`S.c_str()` – the result of this function is a *zero-ended array* contains a copy of string `S`. So if you have

```
AnsiString S = "Honey";
```

then it will be `true` that

```
S.c_str()[0] == 'H'
```

`S.Length()` – returns an `int` number – the current length of `S`.

`S.IsEmpty()` – returns a `bool` value (`true` if `S` is empty, `false` otherwise.)

`S.Pos(sub_S)` – returns the *index* of a *first occurrence* of a *substring* `sub_S` in the string `S`. If there are no occurrences of `sub_S`, it returns 0. Parameter can be a character, a constant string, or an AnsiString variable. So if `S` contains the string `"twins"`, the value of `S.Pos("win")` equals to 2.

`S.SubString(iBeg, sLen)` – returns the substring of `S`, which begins at index `iBeg` and has `sLen` characters in length. So if `S` contains the string `"twins"`, the value of `S.SubString(2,3)` equals to `"win"`.

`S.Delete(iBeg, sLen)` – deletes `sLen` characters from starting from index `iBeg` and returns the result of deletion. So if `S` contains the string `"twins"`, the value of `S.Delete(2,3)` equals to `"ts"`.

`S.Insert(S1, iIns)` – inserts the string `S1` into the string `S` in the position marked by index `iIns`. So if `S` contains the string `"twins"`, and `S1` contains `"o_tw"`, the value of `S.Insert(S1,3)` equals to `"two_twins"`. Just the same result will be obtained, if `S1` contains `"two_"` and we use function calls `S.Insert(S1, 1)` or `S1.Insert(S, S1.Length()+1)`.

If you look the "Help" to the AnsiString-type you can see many other useful functions which help to manipulate this kind of data.

Let us now compose some programs dealing with AnsiStrings.

Example 1. Write a function that erases all ending blanks from a given AnsiString. One of possible solutions is:

```
AnsiString DelEndBlanks(AnsiString S)
{
const char Blank=' ';
int Len=S.Length();
if(Len==0)return S;
while((Len>0)&&
      (S[Len]==Blank))
   {S.Delete(Len,1);
    Len--;
   }
return S;
}
```

You see from this example that it is possible to call the AnsiString-function `Delete` in a mode normally used for functions having `void` returned values. But in fact `Delete` returns `AnsiString` value (more precisely, a *pointer* to an `AnsiString` value); and the above operator `S.Delete(Len,1)` is equivalent to `S=S.Delete(Len,1)`. You will understand the cause of this equivalence after you will learn the notion of `class` of C++-Builder.

Example 2. Write a function that erases all multiple blanks from a given AnsiString (it means that any sequence of blanks must be substituted for a single blank). One of possible solutions is:

```
AnsiString DelMltplBlanks(AnsiString S)
{
const AnsiString twoBlanks="  ";
int Len=S.Length();
if(Len==0)return S;
int j=S.Pos(twoBlanks);
while((Len>0)&&
      (j>0))
   {S.Delete(j,1);
```

```
        Len--;
        j=S.Pos(twoBlanks);
       }
    return S;
    }
```

Example 3. Write a function that returns the first word in a given AnsiString. It is assumed that blank symbol is the only possible delimiter of words in the string. One of possible solutions is:

```
AnsiString Get1stWordIn(AnsiString S)
{
const char Blank=' ';
int Len=S.Length();
if(Len==0)return S;

S=DelMltplBlanks(S);
S=DelEndBlanks(S);
S=DelBegBlanks(S); // write this function yourself
Len=S.Length();
if(Len==0)return S;

int j=S.Pos(Blank);
if(j==0)return S;

return S.SubString(1,j-1);
}
```

Converting numbers to `AnsiString`s and vice versa

You have already used in most your projects the following functions dealing with `AnsiString`s as arguments or as return values:

```
StrToInt
StrToFloat
IntToStr
FloatToStr
```

In cases when the *formatted* conversion of real numbers to text strings is desired, you can substitute `FloatToStr` for more refined function `FloatToStrF`. It has the following parameters:

```
FloatToStrF(aReal, aFormat, aPrecision, aFraction)
```

The parameter `aReal` can be a number of an arbitrary type. It will be converted to a text string using `aFormat` and `aFraction` parameters. The parameter `aPrecision` tells what precision (in digits after decimal point) must be used while `aReal` number converting.

Here is one example of `FloatToStr` call:

```
Edit1->Text=FloatToStrF(R, ffFixed, 7, 2);
```

Here a number contained in the variable `R` is converted to the text form *X.YY* where *X* is an integer part of `R` and *YY* is its fraction rounded to hundredth. The format constant `ffFixed` defines the *X.YY…Y* representation; the precision 7 means that seven digits of `R`'s fraction must be considered while `R` is rounding; and the last parameter causes rounding to hundredth. You can get more information about `FloatToStrF` parameters using HELP of C++-Builder.

Lecture 3.4 Structures

Structures are aggregate **data types** built using elements of other types, including other structs. Consider the structure definition of:

```
struct Calendar //keyword = struct, identifier = Calendar (used later to declare variables of the struct type
{
    int day; //structure member 1 – with a name unique for that structure, values 0-31
    int month; //struct member 2 – cannot be instance of Calendar, value 0-12
    int year; //0-3333
}; //each structure definition must end with ;
```

Properties of structures:
-They do not reserve any space in memory,
-They are used to declare data types.

Declaring structures' variables:

```
Calendar CalendarObject; //declares CalendarObject variable;
Calendar CalendarArray[10]; //declares Calendar Array object;
Calendar *CalendarPointer; //declares CalendarPointer to a Calendar object;
Calendar &CalendarRef; //declares &CalendarRef as a reference to a Calendar object.
```

Accessing Members of structures:

We use the member access operator – the dot operator(.), and the arrow operator (->).

The dot operator accesses a structure or a class member via the variable name for the object. For example, this prints member day of structure CalendarObject:

```
cout << CalendarObject.day;
```

The arrow operator (->) accesses a structure member or class member via a pointer to the object. For example:

```
CalendarPointer = &CalendarObject
```

(this assigns address of structure CalendarObject to CalendarPointer)
To write member day of the structure CalendarObject referenced by CalendarPointer we need to write:

```
cout << CalendarPointer->day;
```

Note that:
CalendarPointer->day is equivalent to (*CalendarPointer).day.