MetaTrader 4 Expert Advisors

MetaQuotes Language 4

MetaQuotes Language 4 (MQL 4) is a new built-in language for programming trading strategies. This language allows the creation of your own Expert Advisors that render the trade process management automatic and are perfectly suitable for implementing your own trade strategies. You can create your own Custom Indicators, Scripts and Libraries of functions with the help of MQL 4, as well.

A large number of functions necessary for the analysis of the current and last quotations, the basic arithmetic and logic operations are included in MQL4 structure. There are also basic indicators built in and commands of order placement and the control over them.

The MetaEditor 4 IDE (Integrated Development Environment) that highlights different constructions of MQL 4 is used for writing the program code. It helps users to orient in the expert system text quite easily. As an information book for MQL 4 we use MetaQuotes Language Dictionary. A brief guide contains functions divided into categories, operations, reserved words and other language constructions and allows finding the description of every element we use.

Programs written in **MetaQuotes Language 4** have different features and purposes:

- Expert Advisors is a mechanical trade system (MTS) linked up to a certain plot. The Advisor can not only inform you about a possibility to strike bargains, but also can make deals on the trade account automatically and direct them right to the trade server. Like most trade systems, the MetaTrader 4 terminal supports testing strategies on historical data with displaying on the chart the spots where trades come in and out.
- Custom Indicators is an analogue of a technical indicator. In other words, Custom Indicators allow creating technical indicators in addition to those already integrated into MetaTrader 4 terminal. Like built-in indicators, they cannot make deals automatically and are aimed only for implementing analytical functions.
- Scripts are programs destined for single execution of some actions. Unlike Expert Advisors, Scripts aren't run tickwise and have no access to indicator functions.
- Libraries are user functions libraries where frequently used blocks of user programs are stored.

Syntax

Format

Spaces, tabs, line feed and form feed symbols are used as separators. You can use any number of such symbols instead of one. To enhance the readability of the text you should use tab symbols.

Comments

Multiline comments start with /* symbols and end with */ symbols. Such comments cannot be nested. Single comments start with // symbols, end with the symbol of a new line and can be nested into multiline comments.
Comments are allowed where blank spaces are possible and tolerate any number of spaces.

Examples:

```
 // single comment

/*  multi-
    line        // nested single comment
    comment
*/
```

Identifiers

Identifiers are used as names of variables, functions and data types. The length of an identifier cannot exceed 31 characters.
Symbols you can use: the numbers 0-9, Latin capital and small letters a-z, A-Z (recognized as different symbols), the symbol of underlining (_). The first symbol cannot be a number. The identifier mustn't coincide with any reserved word.

Examples:

```
NAME1 name1 Total_5 Paper
```

Reserved words

The identifiers listed below are fixed reserved words. A certain action is assigned to each of them, and they cannot be used for other purposes:

Datatypes	**Memory classes**	**Operators**	**Other**
bool	extern	break	false
color	static	case	true
datetime		continue	
double		default	
int		else	
string		for	
void		if	
		return	
		switch	
		while	

Data Types

The main data types are:

- Integer (int)
- Boolean (bool)
- String (string)
- Floating-point number (double)
- Color (color)
- Datetime (datetime)

We need the Color and Datetime types only to facilitate visualization and entering those parameters that we set from expert advisor property tab or custom indicator "Input parameters" tab. The data of Color and Datetime types are represented as integer values.

We use implicit type transformation. The priority of types at a transformation in growing order is the following:

```
int  (bool,color,datetime);
double;
string;
```

Before executing operations (except for the assignment operation) the data is transferred to a type of maximum precision, and before assignment operations - to the integer type.

Integer constants

Decimal: numbers from 0 to 9; Zero shouldn't be the first number.

Examples:

```
12, 111, -956 1007
```

Hexadecimal: numbers from 0 to 9, letters a-f or A-F to represent the values 10-15; they start with 0x or 0X.

Examples:

```
0x0A, 0x12, 0X12, 0x2f, 0xA3, 0Xa3, 0X7C7
```

Integer constants can assume values from -2147483648 to 2147483647. If a constant exceeds this range, the result isn't defined.

Literal constants

Any single character enclosed in single quotes or a hexadecimal ASCII-code of a character looking like '\x10' is a character constant of integer type. Some characters like a single quote ('), double quote (") a question mark (?), a reverse slash (\) and control characters can be represented as a combination of characters starting with a reverse slash (\) according to the table below:

line feed	NL (LF)	\n
horizontal tab	HT	\t
carriage return	CR	\r
reverse slash	\	\
single quote	'	\'
double quote	"	\"
hexadecimal ASCII-code	hh	\xhh

If a character different from those listed above follows the reverse slash, the result isn't defined.

Examples:

```
int a = 'A';
int b = '$';
int c = '©';     // code 0xA9
int d = '\xEA'; // symbol code ®
```

Boolean constants

Boolean constants may have the value of true or false, numeric representation of them is 1 or 0 respectively. We can also use synonyms True and TRUE, False and FALSE.

Examples:

```
bool a = true;
bool b = false;
bool c = 1;
```

Floating-point number constants

Floating-point constants consist of an integer part, a dot (.) and a fractional part. The integer and the fractional parts are a succession of decimal numbers. An unimportant fractional part with the dot can be absent.

Examples:

```
double a = 12.111;
double b = -956.1007;
double c = 0.0001;
double d = 16;
```

Floating-point constants can assume values from 2.2e-308 to 1.8e308. If a constant exceeds this range, the result isn't defined.

String constants

String constant is a succession of ASCII-code characters enclosed in double quotes: "Character constant".

A string constant is an array of characters enclosed in quotes. It is of the string type. Each string constant, even if it is identical to another string constant, is saved in a separate memory space. If you need to insert a double quote (") into the line, you must place a reverse slash (\) before it. You can insert any special character constants into the line if they have a reverse slash (\) before them. The length of a string constant is from 0 to 255 characters. If the string constant is longer, the superfluous characters on the right are rejected.

Examples:

```
"This is a character string"
"Copyright symbol \t\xA9"
"this line with LF symbol \n"
"A" "1234567890" "0" "$"
```

Color constants

Color constants can be represented in three ways: by character representation; by integer representation; by name (for concrete Web colors only).
Character representation consists of four parts representing numerical rate values of three main color components - red, green, blue. The constant starts with the symbol C and is enclosed in single quotes. Numerical rate values of a color component lie in the range from 0 to 255.
Integer-valued representation is written in a form of hexadecimal or a decimal number. A hexadecimal number looks like 0x00BBGGRR where RR is the rate of the red color component, GG - of the green one and BB - of the blue one. Decimal constants aren't directly reflected in RGB. They represent the decimal value of the hexadecimal integer representation.
Concrete colors reflect the so-called Web colors set.

Examples:

```
// symbol constants
C'128,128,128'      // gray
C'0x00,0xFF,0xFF'  // blue
// named color
Red
Yellow
Black
// integer-valued representation
0xFFFFFF               // white
16777215              // white
0x008000              // green
32768                 // green
```

Datetime constants

Datetime constants can be represented as a character line consisting of 6 parts for value of year, month, date, hour, minutes and seconds. The constant is enclosed in simple quotes and starts with a D character.
Datetime constant can vary from Jan 1, 1970 to Dec 31, 2037.

Examples:

```
D'2004.01.01 00:00'     // New Year
D'1980.07.19 12:30:27'
D'19.07.1980 12:30:27'
D'19.07.1980 12'   //equal to D'1980.07.19 12:00:00'
D'01.01.2004'      //equal to D'01.01.2004 00:00:00'
D'12:30:27'        //equal to D'[compilation date] 12:30:27'
D''                //equal to D'[compilation date] 00:00:00'
```

Operations & Expressions

Expressions

An expression consists of one or more operands and operation characters. An expression can be written in several lines.

Example:

```
a++; b = 10; x = (y*z)/w;
```

Note: An expression that ends with a semicolon is an operator.

Arithmetical operations

Sum of values	i = j + 2;
Difference of values	i = j - 3;
Changing the operation sign	x = — x;
Product of values	z = 3 * x;
Division quotient	i = j / 5;
Division remainder	minutes = time % 60;
Adding 1 to the variable value	i++;
Subtracting 1 from the variable value	k-—;

The operations of adding/subtracting 1 cannot be implemented in expressions.

Example:

```
int a=3;
a++;              // valid expression
int b=(a++)*3; // invalid expression
```

The operation of assignment

Note: The value of the expression that includes this operation is the value of the left operand following the bind character.

Assigning the y value to the x variable	y = x;
Adding x to the y variable	y += x;
Subtracting x from the y variable	y -= x;
Multiplying the y variable by x	y *= x;
Dividing the y variable by x	y /= x;
Module x value of y	y %= x;
Logical shift of y representation to the right by x bit	y >>= x;
Logical shift of y representation to the left by x bit	y <<= x;
Bitwise operation AND	y &= x;
Bitwise operation OR	y |= x;
Bitwise operation exclusive OR	y ^= x;

Note: There can be only one operation of assignment in the expression. You can implement bitwise operations with integer numbers only. The logical shift operation uses values of x less than 5 binary digits. The greater digits are rejected, so the shift is for the range of 0-31 bit. By %= operation a result sign is equal to the sign of divided number.

Operations of relation

The logical value FALSE is represented with an integer zero value, while the logical value TRUE is represented with any value different from zero.
The value of expressions containing operations of relation or logical operations is a 0 (FALSE) or a 1 (TRUE).

True if a equals b	a == b;
True if a doesn't equal b	a != b;
True if a is less than b	a < b;
True if a is greater than b	a > b;
True if a is less than or equals b	a <= b;
True if a is greater than or equals b	a >= b;

Two nonnormalized floating point number cannot be linked by == or != operations. For this purpose it is necessary to subtract one from another and the normalized outcome need to compare to null.

Boolean operations

The operand of negation NOT (!) must be of arithmetic type; the result equals true(1) if the operand value is false(0); the result equals to false(0) if the operand is different from false(0).

```
// True if a is false.
 if(!a)
    Print("not 'a'");
```

The logical operation OR (||) of values x and y. The value of expression is true(1) if x or y is true. Else the value of expression is false(0).

Example:

```
if(x<k || x>l)

   Print("out of range");
```

The logical operation AND (&&) of values x and y. The value of this expression is true(1) if x and y values are true. Else the value of expression is false(0).

Example:

```
if(p!=x && p>y)
   Print("TRUE");
n++;
```

Bitwise operations

One's complement of variables values. The value of the expression contains 1 in all digits where n contains 0; the value of the expression contains 0 in all digits where n contains 1.

```
b = ~n;
```

Binary-coded representation of x is shifted to the right by y digits. The right shift is logical shift, that is the freed left-side bits will be filled with zeros.

Example:

```
x = x >> y;
```

The binary-coded representation of x is shifted to the right by y digits; the freed right-side bits will be filled with zeroes.

Example:

```
x = x << y;
```

Bitwise operation AND of binary-coded x and y representations. The value of the expression contains 1 (TRUE) in all bits where both x and y aren't equal to zero; the value of the expression contains 0 (FALSE) in all other bits.

Example:

```
b = ((x & y) != 0);
```

Bitwise operation OR of binary-coded x and y representations. The value of expression contains 1 (TRUE) in all bits where one from x or y is not equal to zero; the value of the expression contains 0 (FALSE) in all other bits.

Example:

```
b = x | y;
```

Bitwise operation EXCLUSIVE OR of binary-coded x and y representations. The value of expression contains 1 in all bits where x and y have different binary values; the value of the expression contains 0 in all other bits.

Exapmle:

```
b = x ^ y;
```

Note: Bitwise operations are executed to integers only.

Other operations

Indexing. At addressing to i' element of array, the value of expression equals to the variable under number i.

Example:

```
array[i] = 3;
 //Assign the value of 3 to array element with index i.
//Mind that the first array element
//is described with the expression array [0].
```

The call of function with x1,x2,…,xn arguments. The expression accepts the value returned by the function. If the returned value is void type, you cannot place such function call on the right in the assignment operation. Mind that the expressions x1,x2,…,xn are sure to be executed is this order.

Example:

```
double SL=Ask-25*Point;
double TP=Ask+25*Point;
int ticket=OrderSend(Symbol(),OP_BUY,1,Ask,3,SL,TP,
                     "My comment",123,0,Red);
```

The "comma" operation is executed from left to right. A pair of expressions separated by a comma is calculated from left to right with a subsequent deletion of the left expression value. All side effects of left expression calculation can appear before we calculate the right expression. The result type and value coincide with the type and value of the right expression.

Precedence rules

Each group of operations in the table has the same priority. The higher the priority is, the higher is the group's position in the table.
The execution order determines the grouping of operations and operands.

()	Function call	From left to right
[]	Array element selection	
!	Negation	From left to right
~	Bitwise negation	
-	Sign changing operation	
*	Multiplication	From left to right
/	Division	
%	Module division	
+	Addition	From left to right
-	Subtraction	
<<	Left shift	From left to right
>>	Right shift	
<	Less than	From left to right
<=	Less than or equals	
>	Greater than	
>=	Greater than or equals	
==	Equals	From left to right
!=	Not equal	
&	Bitwise AND operation	From left to right
^	Bitwise exclusive OR	From left to right
|	Bitwise OR operation	From left to right
&&	Logical AND	From left to right
||	Logical OR	From left to right
=	Assignment	From right to left
+=	Assignment addition	
-=	Assignment subtraction	
*=	Assignment multiplication	
/=	Assignment division	
%=	Assignment module	
>>=	Assignment right shift	
<<=	Assignment left shift	
&=	Assignment bitwise AND	
|=	Assignment bitwise OR	
^=	Assignment exclusive OR	
,	Comma	From left to right

Use parentheses to change the execution order of the operations.

Operators

Format and nesting

Format. One operator can occupy one or several lines. Two or more operators can be situated on the same line.

Nesting. Execution order control operators (if, if-else, switch, while and for) can be nested into each other.

Compound operator

A compound operator (a block) consists of one or more operators of any type enclosed in braces {}. The closing brace shouldn't be followed by a semicolon (;).

Example:

```
if(x==0)
  {
   x=1; y=2; z=3;
  }
```

Expression operator

Any expression followed by a semicolon (;) is an operator. Here are some examples of expression operators:

Assignment operator.

Identifier=expression;

You can use an assignment operator in an expression only once.

Example:

```
x=3;
y=x=3; // error
```

Function call operator

Function_name (argument1,…, argumentN);

Example:

```
fclose(file);
```

Null operator

It consists of a semicolon (;) only. We use it to denote a null body of a control operator.

Break operator

A break; operator terminates the execution of the nearest nested outward operator switch, while or for. The control is given to the operator that follows the terminated one. One of the purposes of this operator is to finish the looping execution when a certain value is assigned to a variable.

Example:

```
for(i=0;i<n;i++)
  if((a[i]=b[i])==0)
    break;
```

Continue operator

A continue; operator gives control to the beginning of the nearest outward cycle operator while or for, calling the next iteration. The purpose of this operator is opposite to that of break.

Example:

```
for(int i=0,int k=9;i<10; i++, k--)
  {
   if(a[i]!=0) continue;
   a[i]=b[k];
   }
```

Return operator

A return; operator terminates the current function execution and returns the control to the calling program.
A return(expression); operator terminates the current function execution and returns the control to the calling program together with the expression value. The operator expression is enclosed in parentheses. The expression shouldn't contain an assignment operator.

Example:

```
return(x+y);
```

Void functions are necessary to complete by "return;"-operator without expression.

Example:

```
return;
```

The completing curly bracket of the function assumes implicit realization of an operator return without expression.

Conditional operator if

if (expression)
   operator;

If the expression is true, the operator is executed. If the expression is false, the control is given to the expression following the operator.

Example:

```
if(a==x)
  temp*=3;
 temp=MathAbs(temp);
```

Conditional operator if-else

```
if (expression)
    operator1
else
    operator2
```

If the expression is true, operator1 is executed and the control is given to the operator that follows operator2 (operator2 is not executed). If the expression is false, operator2 is executed.

The "else" part of the "if" operator can be omitted. Thus, an ambiguity may appear in nested "if" operators with an omitted "else" part. If it happens, "else" addresses to the nearest previous operator "if" in the block that has no "else" part.

Example:

```
//   The "else" part refers to the second "if" operator:
if(x>1)
   if(y==2)
      z=5;
else z=6;

 //   The "else" part refers to the first "if" operator:
if(x>l)
  {
   if(y==2)
      z=5;
  }
else z=6;

 //   Nested operators
 if(x=='a')
   y=1;
else if(x=='b')
  {
   y=2;
   z=3;
  }
else if(x=='c')
  y = 4;
else
  Print("ERROR");
```

Switch operator

```
switch (expression)
{
     case constant: operators
     case constant: operators
     ...
     default: operators
}
```

It compares the expression value with constants in all variants of "case" and gives control to the operator that resembles the expression value. Each variant of "case" can be marked with an integer or character constant or a constant expression. The constant expression mustn't include variables and function calls.

Example:

```
case 3+4: //valid
case X+Y: //invalid
```

Operators connected with "default" label are executed if none of constants in "case" operators equals the expression value. "Default" variant is not obligatory final. If none of the constants resembles the expression value and "default" variant is absent, no actions are executed. The "case" keyword and the constant are just labels and if operators are executed for some variant of "case" the program will further execute the operators of all following variants until it hits "break" operator. It allows linking one succession of operators with several variants. A constant expression is calculated during compilation. None of two constants in one switch operator can have the same values.

Example:

```
switch(x)

  {
   case 'A':
      Print("CASE A\n");
      break;
   case 'B':
   case 'C':
      Print("CASE B or C\n");
              break;
   default:
      Print("NOT A, B or C\n");
      break;
  }
```

Cycle operator while

while (expression) operator

If the expression is true, the operator is executed till the expression becomes false. If the expression is false, the control is given to the next operator.

Note: An expression value is defined before executing the operator. Therefore, if the expression is false from the very beginning, the operator is not executed at all.

Example:

```
while(k<n)
   {
    y=y*x;
    k++;
   }
```

Cycle operator for

for (expression1; expression2; expression3)
*   operator;*

Expression1 describes the initialization of the cycle. Expression2 is the cycle termination check. If it is true, the loop body operator is executed, Expression3 is executed. The cycle is repeated until Expression2 becomes false.
If it is false, the cycle is terminated and the control is given to the next operator. Expression3 is calculated after each iteration. The for operator is equivalent to the following succession of operators:

```
expression1;
while (expression2)
{
    operator;
```

```
   expression3;
};
```

Example:

```
  for(x=1;x<=7;x++)
    Print(MathPower(x,2));
```

Any of the three or all three expressions can be absent in the "for" operator, but you shouldn't omit the semicolons (;) that separate them.

If Expression2 is omitted it is considered constantly true. The "for(;;)" operator is a continuous cycle equivalent to the "while(l)" operator.
Each of the expressions 1-3 can consist of several expressions united by a comma operator ','.

Example:

```
  for(i=0,j=n-l;i<n;i++,j--)
     a[i]=a[j];
```

Functions

Function definition

A function is defined by return value type declaration, by formal parameters and a compound operator (block) that describes actions the function executes.

Example:

```
  double                       // type
   linfunc (double a, double b)  // function name and
                               // parameters list
    {                          // nested operators
     return (a + b);           // returned value
    }
```

The "return" operator can return the value of expression included into this operator. In case of necessity, the expression value assumes the type of function outcome. A function that doesn't return a value must be of "void" type.

Example:

```
  void errmesg(string s)
    {
     Print("error: "+s);
    }
```

Function call

function_name (x1,x2,...,xn)

Arguments (actual parameters) are transferred according to their value. Each expression x1,…,xn is calculated and the value is passed to the function. The order of expressions calculation and the order of values loading are guaranteed. During the execution, the system checks the number and type of arguments given to the function. Such way of addressing to the function is called a value call. There is also another way - call by link. A function call is an expression that assumes the value returned by the function. This function type must correspond to the type of the returned value. The function can be declared or described in any part of the program:

```
int somefunc()
  {
   double a=linfunc(0.3, 10.5, 8);
  }
double linfunc(double x, double a, double b)
  {
   return (a*x + b);
  }
```

Special functions *init, deinit* and *start*

Every program begins its work with the "init()" function. "Init()" function, attached to charts, is launched also after client terminal starting and in case of changing financial symbol and/or charts periodicity.

Every program finishes its work with the "deinit()" function. "Deinit()" function is launched also by client terminal shutdowning, chart window closing, changing financial symbol and/or charts periodicity.

By new quotations "start()" function of attached expert advisors and custom indicator programs is executed. If by new quotations "start()" function, triggered on the previous quotation, was fulfilled, the next calling "start()" function will be executed only after "return()" instruction. All new quotations that receiving during program execution, are ignored by program.

Detaching of the program from charts, changing financial symbol and/or charts periodicity, charts closing and also client terminal exiting interrupts execution of program.

Execution of scripts does not depend on coming quotations.

Variables

Definitions

Definitions are used to define variables and to declare types of variables and functions defined somewhere else. A definition is not a operator. Variables must be declared before they are used. Only constants can be used to initialize variables.

The basic types are:

- string - a string of characters;
- int - an interger;
- double - a floating-point number (double precision);
- bool - a boolean number "true" or "false".

Example:

```
string sMessageBox;
int    nOrders;
double dSymbolPrice;
bool   bLog;
```

The additional types are:

- datatime - date and time, unsigned integer, containing seconds since 0 o'clock on January, 1, 1970.
- color - integer reflecting a collection of three color components.

The additional data types make sense only at the declaration of input data for more convenient their representation in a property sheet.

Example:

```
extern datatime tBegin_Data   = D'2004.01.01 00:00';
extern color    cModify_Color = C'0x44,0xB9,0xE6';
```

Arrays

Array is the indexed sequence of the identical-type data.

```
int a [50];            //A one-dimensional array of 50 integers.
double m[7][50];       //Two-dimensional array of seven arrays,
                       //each of them consisting of 50 integers.
```

Only an integer can be an array index. No more than four-dimensional arrays can be declared.

Defining local variables

The variable declared inside any function, is local. The scope of a local variable is limited to limits of the function inside which one's declared. The local variable can be initialized by outcome of any expression. Every call of function execute the initialization of local variables. Local variables are stored in memory area of corresponding function.

Formal parameters

Examples:

```
void func(int x, double y, bool z)
{
...
}
```

Formal parameters are local. Scope is the block of the function. Formal parameters must have names different from those of external variables and local variables defined within one function. In the block of the function to the formal parameters some values can be assigned. Formal parameters can be initialized by constants. In this case the initializing value is considered as default value. The parameters following the initialized parameter, should be initialized too.

By calling this function the initialized parameters can be omitted, instead of them defaults will be substituted.

Example:

```
   func(123, 0.5);
```

Parameters are passed by value. That is modifications of an corresponding local variable inside the called function will not be reflected in any way in the calling function. As parameters it is possible to pass arrays. However, for an array passed as parameter, it is impossible to change the array elements.

There is a possibility to pass parameters by reference. In this case modification of such parameters will be reflected on corresponded variables in the called function. To point, that the parameter is passed by reference, after a data type it is necessary to put the modifier &.

Example:

```
void func(int& x, double& y, double& z[])
{
...
}
```

Arrays also can be passed by reference, all modifications will be reflected in the initial array. The parameters that passed by reference, cannot be initialized by default values.

Static variables

The memory class "static" defines a static variable. The specifier "static" is declared before a data type.

Example:

```
  {
   static int flag
  }
```

Static variables are constant ones since their values are not lost when the function is exited. Any variables in a block, except the formal parameters of the function, can be defined as static. The static variable can be initialized by corresponded type constant, as against a simple local variable which can be initialized by any expression. If there is no explicit initialization the static variable is initialized by null. Static variables are initialized single-valuedly before calling "init()" function. That is at an exit from the function inside which the static variable is declared, the value of this variable is not lost.

Defining global variables

They are defined on the same level as functions, i.e. they are not local in any block.

Example:

```
int Global_flag;

 int start()
{
...
}
```

Scope of global variables is all program. Global variables are accessible from all functions defined in the program. They are initialized with zero if no other initial value is explicitly defined. The global variable can be initialized only by corresponded type constant. Initialization of global variables is made single-valuedly before execution of "init()" function.

Note: it is not necessary to mix the variables declared at a global level, to global variables of Client Terminal, access to which is carried out by GlobalVariable...() function.

Defining extern variables

The memory class "extern" defines an extern variable. The specifier "extern" is declared before a data type.

Example:

```
extern double InputParameter1 = 1.0;
int init()
{
...
}
```

Extern variables define input data of the program, they are accessible from a property program sheet. It is not meaningful to define extern variables in scripts. Arrays cannot represent itself as extern variables.

Initializing variables

Any variable can be initialized during its definition. Any permanently located variable is initialized with zero (0) if no other initial value is explicitly defined. Global and static variables can be initialized only by constant of corresponded type. Local variables can be initialized by any expression, and not just a constant. Initialization of global and static variables is made single-valuedly. Initialization of local variables is made each time by call of corresponded functions.

Basic types

Examples:

```
  int mt = 1;              // integer initialization
  // initialization floating-point number (double precision)
  double p = MarketInfo(Symbol(),MODE_POINT);
  // string initialization
  string s = "hello";
```

Arrays

Example:

```
  mta [6] = {1,4,9,16,25,36};
```

The list of array elements must be enclosed by curly braces. If the array size is defined, the values that are not explicitly defined are equal to 0.

External function definition

The type of external functions defined in another component of a program must be explicitly defined. The absence of such a definition may result in errors during the compilation, assembling or execution of your program. While describing an external object, use the key word #import with the reference to the module.

Examples:

```
 #import "user32.dll"
   int      MessageBoxA(int hWnd ,string lpText,
                        string lpCaption,int uType);
   int      SendMessageA(int hWnd,int Msg,int wParam,int lParam);
 #import "lib.ex4"
   double   round(double value);
 #import
```

Preprocessor

If the first character in a program line is #, it means that this line is a compiler command. Such a compiler command ends with a carriage-return character.

Declaring of constant

#define identifier_value

The identifier of a constant obeys the same rules as variable names. The value can be of any type.

Example:

```
  #define ABC          100
  #define PI           0.314
  #define COMPANY_NAME "MyCompany Ltd."
```

The compiler will replace each occurrence of an identifier in your source code with the corresponding value.

Controlling compilation

#property identifier_value

The list of predefined constant identifiers.

Example:

```
  #property link          "www.mycompany.com"
  #property copyright     "My Company®"
  #property stacksize     1024
```

Constant	Type	Description
link	string	a link to the company's website
copyright	string	the company's name
stacksize	int	stack size
indicator_chart_window	void	show the indicator in the chart window
indicator_separate_window	void	show the indicator in a separate window
indicator_buffers	int	the number of buffers for calculation, up to 8
indicator_minimum	int	the bottom border for the chart
indicator_maximum	int	the top border for the chart
indicator_color X	color	the color for displaying line X, where X is from 1 to 8

The compiler will write the declared values to the settings of the executable module.

Including files

Note: The #include command line can be placed anywhere in the program, but usually all inclusions are placed at the beginning of the source code.
#include <file_name>

Example:

```
#include <win32.h>
```

The preprocessor replaces this line with the content of the file win32.h. Angle brackets mean that the file win32.h will be taken from the default directory (usually, terminal_direcroty\experts\include). The current directory is not searched.
#include "file_name"

Example:

```
#include "mylib.h"
```

The compiler replaces this line with the content of the file mylib.h. Since this name is enclosed by quotation marks, the search is performed in the current directory (where the main file of the source code is located). If the file is not found in the current directory, directories defined in the compiler settings are searched. If the find is not found there either, the default directory is searched.

Importing functions and other modules

```
#import "file_name"
    func1();
    func2();
#import
```

Example:

```
#import "user32.dll"
   int MessageBoxA(int hWnd,string lpText,string lpCaption,
                   int uType);
   int MessageBoxExA(int hWnd,string lpText,string lpCaption,
                     int uType,int wLanguageId);
#import "melib.ex4"
#import "gdi32.dll"
   int      GetDC(int hWnd);
   int      ReleaseDC(int hWnd,int hDC);
#import
```

Functions are imported from MQL4 compiled modules (*.ex4 files) and from operating system modules (*.dll files). In the latter case, the imported functions are also declared. A new #import command (it can be without parameters) finishes the description of imported functions.