Open Basic Interpreter

User Guide

Version 1.94

Table of contents

1 Abstract

Open Basic (OB) is a Basic programming language interpreter.
Open Basic is developed for embedding into user application as a scripting language.

Open Basic is distributed as a source code under MIT License.

Open Basic system commands can be expanded by attaching user-defined functions to Open Basic execution system.

User-defined functions can be written in C/C++ programming language, assembler or other languages.
User-defined functions can be activated from a Basic-program, they can receive parameters of various types from a Basic-program and return their operation results to a Basic-program.

The interface specially developed to activate user-defined functions allows to determine the type and the sequence order of parameters in execute phase.

Open Basic implements Basic programming language subset of commands. Open Basic is written entirely in C++ and implemented as a class named ob_obasic.

Open Basic supports the following types of data:

1. Floating-point numbers.
2. Fixed-point numbers.
3. Strings.
4. Arrays of floating-point numbers.
5. Arrays of fixed-point numbers.
6. Arrays of strings.

Open Basic has a multithread-safe code.

This document describes:

1. Operating instructions to Open Basic interpreter.
2. Syntax of Open Basic operators.
3. User-defined functions interface to access Open Basic data.
4. Rules to attach user-defined functions.
5. Open Basic data types.
6. Debugging options.

Open Basic does not use any graphic libraries.
Open Basic does not use any operating system calls.

Vendor:

http://sourceforge.net/projects/obasic/
http://www.mktmk.narod.ru.

e-mail: mktmk<at>yandex.ru
e-mail: openbasicsoft<at>gmail.com

MKTMK software company. Moscow, Russia

Feedback

If you would like to use the interpreter and to receive information on new versions release or our e-mail/web address possible changes, please contact us via e-mail.

MKTMK software company is interested in getting information on your experience of this interpreter use:

1. What do you apply the interpreter for?
2. Your compiler and its version number.
3. Bugs found in the program.
4. Documentation completeness and mistakes found in it.
5. Have you attached any user-defined functions to Open Basic execution system?
6. Your commentary on a user-defined functions interface.
7. What operating system do you use?
8. Where have you got information on the interpreter from?

This information will help us when maintaining the program.

Disclaimer

The interpreter and its documentation are provided as is.
MKTMK company is not responsible for possible damage or loss caused by use of the interpreter.
We will appreciate any information on bugs and mistakes found in the program provided by you.

References

All trademarks mentioned in this document are the property of their owners.

List of files:

name	content
.\bas	example files that can be executed in ob194.exe console application.
.\bat	example files that run ob194.exe console application.
.\source	source code files.
.\comline	functions for command line parsing.
.\example	main() of ob194.exe test example, user-defined functions and examples of their attaching.
.\exe	ob194.exe console application.
.\documentation	files containing user guides in Russian and English.
.\include	headers file

2 History of changes and bugs fixes

1. 27.may.2018 – version 1.94 for Windows

    1. The version was compiled and tested in Visual Studio 2017 and wxDev-C++ (GCC).

    2. The Borland C ++ Builder 6.0 compiler is no longer supported. Version 1.94 was not compiled and was not tested on Borland C ++ Builder 6.0.

    3. Fixed compilation error on VS2017 (uninitialized pointer).

    4. Fixed compilation error on VS2017 (message about the use of the insecure function localtime).

    5. Added a new type of user-defined functions that returns the ob_type_standartstring type. For a version Open Basic without wchar_t support, ob_type_standartstring is equivalent to the standard C ++ string type. For a version Open Basic with wchar_t support, ob_type_standartstring is equivalent to the standard C ++ wstring type. Version 1.94 does not have support for wchar_t.

    6. Removed saving of the debug line in the load_project method. To organize the debugger, it is recommended to use the ob_type_istreamcommon * i stream obtained by the get_current_input_stream () method. At the same time, we must not forget that:

    - stream ob_type_istreamcommon * i is opened in ios::binary mode

    - after using the ob_type_istreamcommon *i stream, it is necessary to return its pointer to the initial state and reset the error flags (for example, the error eof may appear when reading the last character of the stream).

    - if the user still needs to store strings or other information, then the user himself must organize storing and retrieving strings (or other data) using the value ob_type_istreamcommon * i as the storage key

    7. To extend the capabilities of the interpreter, it is recommended to create a new class from the ob_obasic class and implement all the new functionality as methods of this derived class.

Example:

```
class user_ob_obasic: public ob_obasic{

//user data

public:

//user methods

};
```

    8. In the example ob.cpp, the key "-m" is added. With this key, the text of the *.bas-program is stored not in the disk, but in memory. The location of the text *.bas-program in memory allows to increase the speed of execution of *.bas-program in 10 times. For store the *.bas-program in memory the istringstream class is used.

    9. The obmain.h file is removed from the project. Now the project has one *.h file named ob.h. If the user wants to redefine the macros in the ob.h file, then he must create his user_ob.h file in which to override the macros.

    10. The ability to override the locality sign for variable names is introduced. To do this, use the methods get_local_name_detect and set_local_name_detect.

    11. Added several user-defined functions.

2. 25.oct.2015 – version 1.93 for Windows

12. Fixed a bug with the return of the version number in the version 1.92.

13. Fixed the e-mail address for contacts.

2. 18.oct.2015 – version 1.92 for Windows

   1. Added support for macro conditional compilation OB_USER_FILE_OPERATION_OFF. Now you can disable disk operations in the interpreter and use the interpreter as a server script.
   2. Fixed error in declaration using namespase std; in file comline.h. The error occurs only when you use the compiler Borland C ++ Builder 6.0.

3. 16.feb.2013 – version 1.91 for Windows

   1. We added the support for local (in the file) labels, variables and arrays. Local labels, variables and arrays have a name beginning with "_" undeline character. The support for local (in the file) labels, variables and arrays reduces the naming conflict when developing multi-file projects.
   2. We developed an option to continue a Basic operator on the next line with "\" backslash character.
   3. To access local variables and arrays, the methods of access to the variables got a new parameter – a pointer to the stream (file), which contains a variable. If you specify a local variable name and a pointer to the stream, the search is performed in the table of the specified stream local variables.
   4. readvar, writevar methods were modified. These methods enable access to array elements from user-defined functions. Now the access to array elements by indicating the index of a variable name is not supported. The access to array elements is possible by explicit indicating the index of array elements in «as» parameter of readvar, writevar methods.
   5. The input-output base class has been renamed. The previous name of the input-output base class was ob_functor_break_base. The new one became ob_base_class_for_IO_service.
   6. Input-output via ob_base_class_for_IO_service base class became the only way to implement input-output. The previous input-output mechanism was removed.
   7. We removed ob_basic::restoreinputpointer method. When running ob_base_class_for_IO_service::run_before and ob_base_class_for_IO_service::run_after methods, the next lexeme is available by ob_basic::gettypenextlex method.
   8. The bug is fixed in REM comment operator. Now an error does not occur with false lexems in the comments.
   9. Load_project method now returns its operation result.
   10. We added CHECKLOAD operator, which prevents stream re-downloading.
   11. We added obmain.h header file and the keys of conditional compilation.
   12. The bug is fixed in DATA operator.
   13. Version 1.91 was tested under the following translators: MSVC2010, Borland C++ Builder, vxDev-C++ (GCC).

4. 03.jan.2010 – version 1.90 for Windows

   1. The bug is fixed in REM operator processing at file loading. When commenting on a string label with a colon, the bug caused program infinite looping during load_project method execution.
   2. We developed an option to create projects from multiple files. The option is possible due to two new added methods — clear_project and load_project, which allow multiple *.bas files loading. GOTO operators can be used for jumping between the project files and GOSUB operators can be used for calling between the project files. All labels and variables of the project are avaiable to any module.
   3. Now GOTO and GOSUB operators can jump to a label placed in another file.
   4. We added restriction on the maximum number of GOSUB nested calls. The maximum number of nested calls by default is 4096. The maximum number can be changed by set_max_nested_gosub or get_max_nested_gosub methods.
   5. We added a new mechanism of running a user code before and after the specified operator. The previous mechanism of running a user code was based on interruption of run method execution. The new mechanism of running a user code is based on input-output classes and enables to reduce the number of the code required to arrange input-output. The previous mechanism of running a user code is now deprecated and is used for compatibility with the existing code. This mechanism will be removed in future versions of the interpreter and therefore is not recommended for use in new development projects. For further information please refer to setoperatorbreak and getoperatorbreak methods description in this document.

5. 18.feb.2007 – version 1.80 for Windows and Linux

   1. We fixed the bug in array indexing by an array element.
   2. Now a file name in OPEN operator can be not only by a string constant but also by a string variable.

3. Starting from version 1.80 in *.bas program string labels with a final colon are supported in addition to labels in the form of a number (line numbers).
4. Getstarttime() and getstoptime() methods are excluded from ob_obasic class as not relating to the interpreter primary function . A user is able to control the run time of any interpreter method by him/herself.

6. 28.aug.2005 – version 1.71 for Windows and Linux.

   1. The bug is fixed in the function of version number return.

7. 21.aug.2005 – version 1.70 for Windows and Linux.

   1. The bug is fixed in hexadecimal numbers operating.
   2. Multithread libraries for MSVC7 are created.

8. 23.feb.2005 – version 1.50 for Windows and Linux.

9. 1998 – the first version for DOS.

3 Program task and purpose

When working in a user application, it often becomes necessary to use a simple scripting language.
If an application operates several primitives that are called in a variety of sequence orders in various operation modes, each primitive call could be in the form of a seperate spripting language function. This approach enables to make flexible changes in the application algorithm without code recompile.

For example:

1. Database queries under various conditions of data sampling.
2. Hardware and/or software testing
3. Communication using nonstandard hardware and different communication channels.
4. Graphical User Interfaces and manipulation with GUI objects
5. The input language in program-terminals for command line parsing.
6. The input language for use by a technical operator in automated industrial control systems.
7. The input language of complex configuration files.
8. Many others.

A graphical user interface makes the process of task automatization more difficult. Therefore it is often necessary to complement a graphical user interface with features provided by a scripting language.

Open Basic interpreter is developed for embedding into user applications as a scripting language.

Open Basic command system can be expanded by attaching user-defined functions to Open Basic execution system.

User-defined functions can be written in C/C++ programming languages, assembler or other languages.
User-defined functions can be activated from a Basic-program, they can receive parameters of various types from a Basic-program and return their operation results to a Basic-program.

User-defined functions have access to all variables and arrays of a Basic program and not only to those transferred by parameters.
This access to variables and arrays of a Basic program is implemented by using the names of these variables and arrays.

3.1 Commentary on interpreter speed

On average, the interpreter executes the code 100 times slower than an equivalent compiling code.
However, the interpreter has the advantage that it does not need a compiler and makes it much easier to debug.
If the application is provided with an interpreter, the algorithm can make changes without recompiling the application.

In general we can say that linear algorithms with conditional jumps and loops without nesting are executed in the interpreter rather quickly.
Applying nested loops on large amounts of data could require to move the inner loop into a user-defined function (due to the loss of speed).
The use of nested loops with a nesting level of three or more makes sense only for very small amounts of data (about ten).

When identifying an algorithm part that is critical in speed, it is necessary to move this part of the algorithm into a user-defined function with the appropriate parameters. You have to properly distribute the task between user-defined functions and the interpreter. At the level of the interpreter it makes sense to keep control over high-level parts of the algorithm.

Since Open Basic is an interpreter, the length of label names, variables, arrays and other internal interpreter objects affect its speed.

In general the longer the object names are, the more time is spent searching these objects in interpreter tables.

To increase the speed of execution of *.bas-programs in the interpreter, it is recommended to place the text of *.bas-programs in memory, rather than on the disk. The location of the text of *.bas-programs in memory increases the speed of *.bas-programs execution by 10 times. For the location of *.bas-programs in memory, it is recommended to use C ++ string streams, which are implemented in the istringstream classes. An example of the location of *.bas-programs in memory can be found in the ob.cpp file (the -m switch on the command line and the flm field in the input structure for analyzing the command line keys).

4 Terms of use

Open Basic interpreter is written entirely in C++ and does not use system calls of any operating system.
Open Basic interpreter can be compatible with any graphic library.

5 Input and output data

5.1 General information on Open Basic interpreter. How to use the interpreter? How to load and run *.bas program?

The Open Basic interpreter is implemented as a class with ob_obasic name.
The interpreter executes a program written in BASIC programming language.
In this user guide this program is called *.bas-program. *.bas-program can be either a text file or a buffer in memory opened as istream data stream.

The following definitions are further used in the user guide:

"operator"	denotes an operator of a BASIC language (for example, PRINT, FOR etc.).
"user-defined function"	denotes a user-defined function attached to Open Basic execution system.
"function"	denotes C++ function.
"method"	denotes a method of ob_obasic class or other class included into the interpreter.
"*.bas-program"	denotes a program written in BASIC programming language.

To use the interpreter you should:

1. Download from http://sourceforge.net/projects/obasic/ the archive containing the interpreter.
2. Include the interpreter source code into the project (.\source folder).
3. Indicate the path in the project to attach ob.h interpreter header files (.\include folder).
4. Indicate the path in the project to attach mstore.h,mvect.h,mlist.h,mstack.h,mhash.h template containers header files (.\include folder).
5. Include ob.h interpreter header file into your C++ files using #include directive.
6. Declare the use of interpreter namespace using namespace ob_charspace.
7. Create an instance of ob_obasic class in your C++ program.
8. If using PRINT and INPUT operators in *.bas-program for input-output, you create user-defined I/O classes derived from ob_base_class_for_IO_service class and attach them to the interpreter.
9. Clear the project by clear_project method.
10. Load the program to be executed by load_project method.
11. Run the program to be executed by run method.

Example:

```
using ob_charspace; namespace

// load and run of test1.bas program
ifstream inp1("test1.bas",ios::binary);//input stream opening

ob_obasic basic_interpreter;//interpretator instance creating

basic_interpreter.clear_project();//project cleaning
basic_interpreter.load_project(&inp1,0);//test1.bas loading by load_project method
```

```
basic_interpreter.run();//test1.bas program runnng
```

NOTE: The input stream should be opened in ios::binary mode.

COMMENTARY:

Despite *.bas program is a text file it should be opened as a stream in ios::binary mode.
This has to be done due to some features of processing carriage return characters and line feed in ios::text mode.

Example:

```
using ob_charspace; namespace

ob_obasic basic_interpreter;//interpretator instance creating

//after test1.bas program complition load and run of test2.bas and test3.bas programs
ifstream inp2("test2.bas",ios::binary);//input stream opening
ifstream inp3("test3.bas",ios::binary);//input stream opening

basic_interpreter.clear_project();//project cleaning
basic_interpreter.load_project(&inp2,0);//*.bas-program loading
basic_interpreter.load_project(&inp3,0);//*.bas-program loading
basic_interpreter.set_current_input_stream(&inp2);//placing inp2 stream the first to be run

basic_interpreter.run();//test2.bas program running
```

NOTE: Input stream should be opened in ios::binary mode.

NOTE: If project consists of multiple files (streams), the last loaded file (stream) will be the first to be run. In order to set a file (stream) that should be run first, set_current_input_stream method should be used.

5.2 Synchronizing OS threads, internal tables of the interpreter, and using ob_obasic class methods in a multi-threaded environment

During the operation of methods of the ob_obasic class, the internal structures of the interpreter are modified, such as variable tables, array tables, and label tables. The interpreter does not contain any means of synchronizing access to these structures. This means that the user must take care of himself so that calls to methods of the ob_obasic class from different threads of the operating system are not performed simultaneously. That is, the user must ensure that at each point in time only one method of the ob_obasic class is called from different operating system threads.

This can be done using the following algorithm:

1. An instance of the ob_obasic class is created in static memory.
2. Running methods of the ob_obasic class is done in a separate thread of the operating system.
3. An «end flag» is created for the method of the ob_obasic class.
4. The «end flag» is reset before the method starts.
5. After the method is finished, the «end flag» is set.
6. Before starting a method from another thread, the «end flag» flag is checked. If the «end flag» is cleared, the method can not be started, since the previously executed method execution was not completed.

All methods of the ob_obasic class on accessing variables, connecting user-defined functions, and so on are performed in a limited time. The exception is the ob_obasic::run method, which may not end at all due to the presence in the *.bas-program of infinite loops.

To abort the ob_obasic::run method, use the ob_obasic::setstep method with the STEP parameter. The ob_obasic::setstep method can be used at any time during the interpreter's operation without synchronization with other methods, since the ob_obasic::setstep method only sets the internal flag in interpreter structures.

Suppose that the ob_obasic::run method is started. The algorithm for starting, stopping, and possibly resuming execution of the ob_obasic::run method can be as follows:

1. From another (higher-priority) OS thread, use the ob_obasic::setstep method with the STEP parameter

2. Wait for the completion of the ob_obasic::run method with the appropriate stop code
3. After the completion of ob_obasic::run, you can use all methods of the ob_obasic class (access to variables and arrays, connection of user-defined functions, etc.).

The described algorithm guarantees that all modifications of the internal tables of the interpreter will be completed correctly.

After stopping and accessing the interpreter variables, you can continue the ob_obasic::run method. The following algorithm is used for this:

1. Run the ob_obasic::setstep method with the NOSTEP parameter
2. Run the ob_obasic::run method

Connections of user-defined functions to the executing system of the interpreter must be performed before the first use of these functions in the * .bas-program.

5.3 Technical solutions review

The following main problems were coped during the development of the interpreter:

1. A *.bas-program can be used by extracting it from a file on a disc or a memory buffer. To provide this option, the interpreter receives an input program as istream* input stream pointer. The particular stream type (file or buffer) is determined by a user. The same way the input stream for INPUT operator and output stream for PRINT operator can be either a buffer or a file. It should be noted that the location of the *.bas-program in the memory buffer using the istringstream class accelerates the interpreter 10 times.

2. One copy of the interpreter can execute one project at a moment of time. The program can contain multiple instances of the interpreter. The interpreter does not have static data members (except for a few read-only constants available). Therefore, multiple instances of the interpreter can be executed simultaneously in different threads of the program.

3. User-defined functions in Open Basic are of a unified interface. This interface allows you to analyze the parameters types and sequence orders when calling a user-defined function at the program execution stage. This approach requires a programmer to be accurate, but it does not limit the option to create user-defined functions with different parameters.

4. Since the Open Basic interpreter is a C ++ class with the name ob_obasic, the user can create a derivative of the ob_obasic class. It is recommended to implement additional functionality for the interpreter in this way.

Example:

```
class user_ob_obasic: public ob_obasic{

//user data

public:

//user methods

};
```

To provide operation flexibility, *.bas-program execution is divided into three stages:

1. The first stage is project cleaning by clear_project method. During this stage the previous project data are cleaned from the interpeter internal tables (cleaning tables of jump labels, tables of variables and array etc).

2. The second stage is program loading by load_project method. During this stage a table of jump labels and subprograms is created and internal variables are configured. The set of files (streams) simultaneously loaded in an instance of the interpreter is called a project. The concept of common namespace of global variables, names of arrays and labels is applied in the project. The global labels in all files (streams) of the project have not to be repeated. The global variables created in one project file should be accessible in all project files. The names of global labels, variables and arrays begin with any letter except "_" underline character. The digital labels are always global. Starting from version 1.91 you can create local (in a file) labels, variables and arrays. The names of local labels, variables and arrays begin with "_" underline character. The local (in a file) labels, variables, and arrays can be repeated in different

files. The local variables and arrays in each file have their own value and local labels can jump only within the file to which they are attached.

3. Since version 1.94, it is possible to override the locality sign for variable names. To do this, use the methods get_local_name_detect and set_local_name_detect.

4. The third stage is program execution by run method. During this stage Open Basic reads a program text and performs different actions (variables and arrays creation, variables calculation and assignment, operators and functions execution).

run method is exited due to several reasons:

1. The input stream was expired during operation.
2. A syntax error was found in the program during operation.
3. A breakpoint character was found during operation.
4. A character of line end was found during operation when single-step work mode was enabled.

run method returns an exit code. Analyzing run method exit code, a user is able to know the reason of run method exit. If run method has been exited without mistakes, it can be continued without a program reload. For example, if run method has been exited due to a breakpoint character or a character of line end at enabled single-step work mode run method can be continued without a program reload.

If run method has been exited due to a syntax or other mistake, you should clear the project by clear_project method and reload by load_project method before the next run method running.

5.4 Conditional compilation macros review

1. OB_USER_WCHAR macro

OB_USER_WCHAR macro is reserved for future support of Unicode.
If OB_USER_WCHAR macro is defined during compilation, the Open Basic is compiled with Unicode support.
In version 1.94 OB_USER_WCHAR macro is not supported, but there are parts of the algorithm in the code, which will subsequently be active when the program is compiled with Unicode support.

2. OB_USER_VERSION_CHARSPACE macro

OB_USER_VERSION_CHARSPACE macro enables to control the name of the namespace, which contains the interpreter code.
The interpreter code is located in namespace namespace.
The namespace names are different to avoid any name conflict when creating multiple ob_obasic class objects of different versions in a single program.

The table shows the interpreter namespace in accordance with OB_USER_WCHAR and OB_USER_VERSION_CHARSPACE macros for interpreter version 1.94.

	OB_USER_VERSION_CHARSPACE macro is not defined	OB_USER_VERSION_CHARSPACE macro is defined
OB_USER_WCHAR macro is not defined	namespace ob_charspace	namespace ob_charspace194
OB_USER_WCHAR macro is defined	namespace ob_wcharspace	namespace ob_wcharspace194

Example: If OB_USER_VERSION_CHARSPACE macro is defined during compilation and the interpreter is compiled without Unicode support, version 1.94 interpreter code will be in namespace ob_charspace194 namespace.

3. OB_USER_HEADERS macro

OB_USER_HEADER macro allows to specify the library header files to the compiler unlike MSVC2017.

If OB_USER_ HEADERS macro is not defined during compilation, the library header files are of the form used in MSVC2017 compilers and CGG.

If OB_USER_ HEADERS macro is defined during compilation, the library header files should be defined by the user.

To redefine the library header files, a mechanism of replacing the main interpreter header file is used.

user_ob.h user-defined file to have the following content is created:

```
#ifndef OB_THIS
#define OB_THIS

#define OB_USER_HEADERS

#include <sstream>
#include <iostream>
#include <fstream>
#include <math>
#include <stdlib>
#include <ctype>
#include <limits>
#include <time>

#include <ob.h>

#endif
```

user_ob.h user-defined file is used in the project and the library header files will have the names required by a compiler user.

4. OB_USER_INT_AND_FLOAT and OB_USER_POW macros

OB_USER_INT_AND_FLOAT macro enables to redefine the variable types of fixed-point and floating point variables interpreter is working with.

If OB_USER_INT_AND_FLOAT macro is not defined during compilation the fixed-point variables in the interpreter correspond to int type and floating point variables in the interpreter correspond to float type used in C + + compiler .

If OB_USER_INT_AND_FLOAT macro is defined during compilation the fixed point variables and floating point variables in the interpreter should be defined by a user by way of typedef directive.

If you change variables types, you have to consider that the interpreter operation of exponentiation uses pow (double x, double y) function of exponentiation from math.h. Therefore, along with the change of variables types in the interpreter, you should change the function type that supports the operation of exponentiation. You have to redefine ob_type_pow type, which is used as arguments for exponentiation.

OB_USER_POW macro allows to redefine a function used in an operator of exponentiation and function arguments types .

If OB_USER_POW macro is not defined during compilation, the interpreter uses the function to raise to a power:
double pow (double x, double y) of math.h.

To define all these parameters, the mechanism of replacing the main interpreter header file is used.

user_ob.h user-defined file to have the following content is created:

```
#ifndef OB_THIS
#define OB_THIS

#define OB_USER_INT_AND_FLOAT

typedef long double            ob_type_flo;
typedef long int               ob_type_int;

#define OB_USER_POW powl
```

```
typedef long double            ob_type_pow;

#include <ob.h>

#endif
```

user_ob.h is used in the project and the interpreter operation of exponentiation uses powl (long double x, long double y) function.

user_ob.h user-defined file is used in the project and integer variables in the interpreter will correspond to long int type and floating point variables will correspond to long double type.

5. OB_USER_MAXLENGTHSTRING, OB_USER_MAXNUMPAR, OB_USER_MAXARRAYDIMENTION and OB_USER_MAXNUMPARSUBROUTINE macros

OB_USER_MAXLENGTHSTRING enables to redefine the maximum length of the string the interpreter is working with (by default it is 4096).
OB_USER_MAXNUMPAR allows to redefine the maximum number of parameters in the user-defined function (by default it is 64).
OB_USER_MAXARRAYDIMENTION can redefine the maximum number of array dementions in the interpreter (by default it is 20).
OB_USER_MAXNUMPARSUBROUTINE enables to redefine the maximum number of parameters in a parameters subprogram (in version 1.94 it is not supported).

To redefine all of these values, the mechanism of replacing the main interpreter header file is used.

For example, for the interpreter to work with strings of 8192 characters maximum length, you have to create user_ob.h file with the following content:

```
#ifndef OB_THIS
#define OB_THIS

#define OB_USER_MAXLENGTHSTRING (8192)

#include <ob.h>

#endif
```

6. OB_USER_HASHLENGTH macro

Many of the interpreter objects such as labels, variables, arrays are stored in hash-tables. Hash-tables in Open Basic present a vector of lists. The length of a hash-table in Open Basic is this vector lengh.

OB_USER_HASHLENGTH allows to redefine the length of hash-tables in the interpreter.

This macro will affect the interpreter speed.

The general rule is that the length of the hash table to be 10-100 times less than the number of objects contained therein.

Now OB_USER_HASHLENGTH equals by default to 131. This corresponds to *. bas-programs that contains about thousands of labels, variables and arrays.

If the interpreter will run rather large *.bas-programs that contain tens of thousands of labels, variables and arrays to maintain the speed, you should increase the length of hash-tables in the interpreter up to about thousands values.

To redefine this value, the mechanism of replacing the main interpreter header file is used.

To increase the length of hash-tables in the interpreter up to 1031, you should create user_ob.h file to have the following content:

```
#ifndef OB_THIS
```

```
#define OB_THIS

#define OB_USER_HASHLENGTH (1031)

#include <ob.h>

#endif
```

When accessing data in a hash-table, the vector index in Open Basic is calculated as the remainder of dividing the stored object hash-value by the length of the hash-table.

As the length of OB_USER_HASHLENGTH hash-table, it is recommended to use a prime number, since the remainder of integer dividing by a prime better approximates uniform distribution.

8. OB_USER_FILE_OPERATION_OFF macro

With this macro, you can disable operators OPEN and CLOSE. When operators OPEN and CLOSE are disabled you can be used interpreter as a server-side script. When operators OPEN and CLOSE are disabled the client can not get uncontrolled access to the disk server. If you want to provide the client controlled access to the disk server you need to use user-defined functions.

5.5 Input-output system review

There are two alternative ways to arrange data input-output process from/to the interpreter:

1. Using PRINT and INPUT operators along with auxiliary input-output classes.
2. Using user-defined functions of input-output.

5.5.1 The first way to arrange input-output process (using input-output classes).

Input-output operation in Open Basic is organized by PRINT and INPUT operators. PRINT operator formats an output string. INPUT operator formats an input string.

During execution of *.bas program the interpreter works with internal variables, arrays and also with user-defined functions. All operations with these objects are performed by means of the interpretator without calling any operating system services. However, the input-output type and means can differ considerably in different operating systems and applications. Therefore, PRINT and INPUT input-output operators require calling operating system services. To couple interpreter input-output with operating system input-output, setoperatorbreak, setin and setout methods are used.

PRINT operator in the interpretator sends data to the output data stream. This data stream is attached to the interpreter by setout method. ostrstream class with a static buffer is usually used as an output stream. Then the data prepared by PRINT operator should be moved into the stream specified by a user. In the console example, as a final output stream of a formatted string cout stream is used.

INPUT operator in the interpretator receives data from the input data stream. This input stream is attached to the interpreter by setin method. istrstream class with a static buffer is usually used as an input stream. In the console example, as an initial input stream to input characters from a console cin stream is used.

To couple PRINT and INPUT operators, a user should create two classes derived from ob_base_class_for_IO_service class. The basic ob_base_class_for_IO_service class is of the form:

```
class ob_base_class_for_IO_service{

public:

ob_base_class_for_IO_service(){}

virtual ~ob_base_class_for_IO_service(){}
```

```
virtual void run_before(ob_obasic* basicptr)=0;
virtual void run_after(ob_obasic* basicptr)=0;
};
```

The classes derived from ob_base_class_for_IO_service class are called input-output classes for the corresponding operator.

In the classes derived from ob_base_class_for_IO_service class, a user redefines run_before and run_after methods. After that an instance of input-output user-defined classes is created and a user creates reference to the instance of input-output user-defined classes by setoperatorbreak method in the interpretator.

Run_before and run_after methods will be run before and after execution of input-output operators. In run_before methods, a user prepares input-output streams for the interpreter. In run_after methods, a user processes input-output results.

Input-output classes instances should be in the same scope as ob_obasic interpreter instance. If the program contains several ob_obasic interpreter instances, each of them should have its own input-output classes instances to provide operation in multithread environment).

Example:

```
// creating input-output class for PRINT operator

class ob_class_print_service_def:public ob_base_class_for_IO_service{

char printbuffer[ob_maxlengthstring];

ostrstream pr;

public:

ob_class_print_service_def():pr(printbuffer,ob_maxlengthstring){}

virtual ~ob_class_print_service_def(){}

void run_before(ob_obasic* basicptr){
printbuffer[0]=0;
pr.clear();
pr.seekp(0);
basicptr->setout(&pr);
}

void run_after(ob_obasic* basicptr){
if(printbuffer[0]!=0){
pr<<ends;

//--------------------------------------------------
cout<<printbuffer;//your output code place here
//--------------------------------------------------

}//if
}
};



//creating input-output class for INPUT operator

class ob_class_input_service_def:public ob_base_class_for_IO_service{

char inputbuffer[ob_maxlengthstring];
char inputbuffer1[ob_maxlengthstring];

strstream pr;

public:

ob_class_input_service_def():pr(inputbuffer1,ob_maxlengthstring){}

virtual ~ob_class_input_service_def(){}

void run_before(ob_obasic* basicptr){

ob_lex::typelex tl1=basicptr->gettypenextlex();
```

```
if(tll!=ob_lex::SHARP){

inputbuffer[0]=0;

ob_type_char ch;
ob_type_stringsize i;

for(i=0;;++i){

if(i==ob_maxlengthstring){
inputbuffer[ob_maxlengthstring-1]=0;
break;
}//if

//--------------------------------------------------
cin.get(ch);//your input code place here
//--------------------------------------------------
inputbuffer[i]=ch;

if((ch==ob_const_cr)||(ch==ob_const_lf)){
inputbuffer[i]=0;
break;
}//if

}//for

pr.clear();
pr.seekp(0);

pr<<inputbuffer;

pr.seekg(0);
basicptr->setin(&pr);
}//if
}

void run_after(ob_obasic* basicptr){}
};




// creation input-output classes instances

ob_class_print_service_def ob_class_print_service;
ob_class_input_service_def ob_class_input_service;




//placing references on input-output classes in an interpreter instance

example_basic_interpreter.setoperatorbreak("PRINT",&ob_class_print_service);
example_basic_interpreter.setoperatorbreak("INPUT",&ob_class_input_service);
```

In the example, input-output is performed from/to standard cin and cout streams. The given examples of input-output classes can be used when replacing the following lines:

```
//--------------------------------------------------
cout<<printbuffer;//your output code place here
//--------------------------------------------------
//--------------------------------------------------
cin.get(ch);//your input code place here
//--------------------------------------------------
```

by the user code of input-output from/to a corresponding Windows window.

For PRINT operator output is performed line by line. For INPUT operator input is performed symbol-by-symbol. The differences result from the fact that cin.get (ch); operation is blocking the application performance.

For further information on input-output by IO classes see "Methods and enumerations ob_obasic class to couple Open Basic interpreter input-output with operating system input-output ."

5.5.2 The second way to arrange input-output process (using input-output user-defined functions).

Input-output can be arranged without PRINT and INPUT operators by way of input-output user-defined functions. A user writes his/her own input-output user-defined functions and registers them in the interpreter so that these input-output user-defined functions are used in *.bas program text instead PRINT and INPUT operators.

*5.6 ob_obasic class methods and enumerations to load and run *.bas-programs*

5.6.1 ob_obasic::ob_obasic constructor

The constructor:

```
ob_obasic::ob_obasic()
```

creates an instance of Open Basic interpreter.

5.6.2 ob_obasic::clear_project method

The method:

```
void ob_obasic::clear_project()
```

cleans all interpreter tables except a user-defined functions table.

During operation the interpreter creates the following tables:

1. Tables of variables (local and global).
2. Tables of arrays(local and global).
3. Tables of labels (local and global).
4. Tables of FOR operators cycles (cycles tables are always local).
5. A user-defined functions table (one global table per each instance of the interpreter).
6. A table of active files (one global table per each instance of the interpreter).
7. CHECKLOAD string operator table (one global table per each instance of the interpreter).
8. DATA operator data table (stream).
9. A stack of GOSUB subprograms.

A user-defined functions table can be erased only by calling void clrtablfun(); method or ob_obasic::~ob_obasic class destructor.

clear_project method should be always used before the interpreter operation to clean interpreter tables from previous data.

5.6.3 ob_obasic::load_project method and ob_obasic::loadresult enumeration

The method:

```
loadresult ob_obasic::load_project(ob_type_istreamcommon* i,ob_type_char* id)
```

loads one or several *.bas-programs into the interpreter. During loading tables of global and local labels are created.

The parameters:

i	this is a pointer to an input stream, which contains *.bas-program
id	this is a pointer to an identification line of the input stream. The name of a file is usually used as an identification line. An identification line is used to debug when accessing local variables. If you do not plan to use an identifacation line use a null pointer as id parameter

NOTE: i stream should be opened in ios::binary mode.

NOTE: i stream data pointer should point to the beginning of the program.

After the interpreter finishes its work the stream data pointer points to the end of the program.

load_project method is able to load several streams simultaneously. A set of simultaneously loaded streams is called a project.

The exit code of load_project method is the values of the enumeration:

```
enum ob_obasic::loadresult {LOAD_OK, CHECKLOAD_DUPLICATE, STREAM_DUPLICATE, STREAM_NULL};
```

1. LOAD_OK - normal load exit.

2. CHECKLOAD_DUPLICATE — download is interrupted due to detection of the reload by CHECKLOAD operator. When you load *.bas-program the interpreter makes a table of CHECKLOAD operators rows. When the the existing row is detected in this table the load is interrupted with returning CHECKLOAD_DUPLICATE code. For further information refer to the description of CHECKLOAD operator.

3. STREAM_DUPLICATE — download is interrupted due to the detection of the same stream re-download. When you load *.bas-programs for each stream the interpreter creates tables for local variables, arrays, and labels. In addition to local variables, arrays, and labels these tables contain a pointer to the input stream. When detecting the existing table for a stream the load is interrupted with returning STREAM_DUPLICATE code.

4. STREAM_NULL - download is interrupted because a null pointer is passed as i parameter.

5.6.4 ob_obasic::run method ob_obasic::typeend enumeration

The method:

```
typeend ob_obasic::run()
```

runs a loaded program or continues program execution after program stop with exit code returning . If the project consists of more than one stream, the last loaded stream is run. In order to set stream to be run, one should use set_current_input_stream method.

run method returns values out of enumeration:

```
enum ob_obasic::typeend{DUMMYEND,ENDFILEEND,ENDOPERATORDETECT,EOLEND,BREAKPOINTEND,NOLOADEND};
```

The exit code:

DUMMYEND	during normal operation it is never returned by run method. It is used for internal jumps.
ENDFILEEND	run method finished because of the input stream being exhausted. This is normal *.bas- program completion if *.bas- program does not have END operator.
ENDOPERATORDETECT	run method finished when END operator was detected. This is normal *.bas-program completion
EOLEND	run method finished after the next line execution, because a flag was set step=STEP by setstep method
BREAKPOINTEND	run method finished after reading '@' symbol, which is a symbol of breakpoints for Open Basic
NOLOADEND	run method finished because load_project method was not executed and the input stream was not opened

run method is usually run in a separate lower priority stream.

5.6.5 ob_obasic::set_current_input_stream method

The method:

```
void ob_obasic::set_current_input_stream(ob_type_istreamcommon* I)
```

sets up i stream in the interpreter to be executed first when run() method is called. i stream should be loaded beforehand by load_project method.

5.6.6 ob_obasic::get_current_input_stream method

The method:

```
ob_type_istreamcommon* ob_obasic::get_current_input_stream();
```

returns the currently executed stream. ob_type_istreamcommon* ob_obasic::get_current_input_stream() method is used to access local variables of Open Basic interpreter.

5.7 ob_obasic class methods and enumerations used for debugging in Open Basic interpreter

Open Basic has two modes of debugging: step-by-step execution of *.bas-programs and setting breakpoints.

Step-by-step execution of *.bas-programs is carried out by means of setstep method.

'@' character in a separate line in *.bas-program text implements a breakpoint. There can be several breakpoints in *.bas-program. After introducing at least one breakpoint in *.bas-program the whole project should be cleaned by clear_project method and reloaded by load_project method.

When transferring control to a line containing '@' chracter the program will halt. The exit code of run method will be BREAKPOINTEND. To continue the program after the breakpoint you should run again run() method.

5.7.1 ob_obasic::step enumeration

The enumeration:

```
enum step{NOSTEP,STEP}
```

determines step-by-step program execution mode. This enumeration is used in setstep and getstep methods.

If step=NOSTEP, the program is executed without interruptions.
If step=STEP, the program is terminated with typeend=EOLEND code after each program line execution.

The default value is step=NOSTEP.

5.7.2 ob_obasic::setstep method

The method:

```
void ob_obasic::setstep(step ts);
```

sets a step-by-step program execution mode on or off. This method can be used at any moment of the interpreter operation.
If step-by-step program execution mode is set on, run(); method will be terminated after execution of each *.bas-program line.
In order to continue program execution run(); method should be run again.

5.7.3 ob_obasic::getstep method

The method:

```
step ob_obasic::getstep();
```

returns the current mode of step-by-step program execution.

5.7.4 ob_loadbreakstr function

The function:

```
bool ob_loadbreakstr(ob_type_istreamcommon* in,ob_type_stringsize lengthbuffers,ob_type_char*
ptrbefore,ob_type_char* ptrcurrent,ob_type_char* ptrafter);
```

ob_loadbreakstr function seeks the current, previous and following lines in a file and loads corresponding executable program lines into the specified buffers
ob_loadbreakstr function is usually called after *.bas-program halts.

The parameters:

in	a current input executable line of *.bas-program
lengthbuffers	the length of ptrbefore, ptrcurrent, ptrafter buffers
ptrbefore	a buffer for a line preceding the current executable one
ptrcurrent	a buffer for the current executable line
ptrafter	a buffer for the line following the current executable line

ob_loadbreakstr function is not a method of ob_obasic class.

Example:

```
//Creating an interpreter and running a program in a step-by-step mode with printing executable strings

ob_type_char str_before[ob_maxlengthstring];//buffer
ob_type_char str_current[ob_maxlengthstring];//buffer
ob_type_char str_after[ob_maxlengthstring];//buffer

ob_obasic basic_interpreter;//interpreter creating
//test1.bas-program loading and running
//with step-by-step program execution
//and with printing executable strings

ifstream inp1("test1.bas",ios::binary);//input stream opening

basic_interpreter.clear_project();//project cleaning

basic_interpreter.load_project(&inp1,0);//test1.bas program loading

basic_interpreter.setstep(ob_obasic::STEP);//step-by-step mode setting

while(1){

if(basic_interpreter.run()!=ob_obasic::EOLEND) break;//*.bas-program running

ob_loadbreakstr(&inp1,ob_maxlengthstring,str_before,str_current,str_after);

cout<<endl<<"String current: "<<str_current;

}//while
```

5.8 *ob_obasic class methods and enumerations to couple Open Basic interpreter input-output and operating system input-output*

5.8.1 General information

During the execution of *.bas program the interpreter operates internal variables, arrays and user-defined functions. All actions with these objects are performed only by means of the interpreter system without calling operating system services. However, the input-output type and means for multiple operating systems and applications can differ considerably. Therefore, PRINT and INPUT input-output operators require addressing to operating system services. To couple interpreter input-output with operating input-output, setoperatorbreak, setin and setout methods are used.

5.8.2 setoperatorbreak and getoperatorbreak methods

For each of the operators in Open Basic (PRINT, INPUT, FOR etc.), it is possible to execute a user code before and after operator execution.
This option is usually used only for operators like PRINT, INPUT.

The method:

```
bool ob_obasic::setoperatorbreak(ob_type_char* name,ob_base_class_for_IO_service* b);
```

sets a pointer to IO class for “name” operator.

The parameters:

name	an operator name
b	a pointer to the instance of IO class (derived from ob_base_class_for_IO_service class)

The return value:

true	a successful termination
false	an operator name is not found possibly because of wrong operator name setting

The method:

```
bool ob_obasic::getoperatorbreak(ob_type_char* name,ob_base_class_for_IO_service** b);
```

allows to get a pointer to iput-output class. If iput-output class has not been set for a stated operator, b=0. By default for all operators iput-output classes are not set.

The parameters:

name	an operator name
b	a pointer to the instance of IO class (derived from ob_base_class_for_IO_service class)

The return value:

true	successful termination
false	an operator name is not found possibly because of wrong operator name setting

5.8.3 ob_obasic::setin и ob_obasic::setout methods

The methods:

```
void setin(ob_istreamcommon* i);
void setout(ob_ostreamcommon* o);
```

allow to redefine input and output streams.

NOTE: The input stream should be opened in ios::binary mode.
NOTE: The output stream should be opened in ios::text mode.

setin и setout methods are used to arrange input-output.

5.8.4 ob_lex::typelex enumeration

The enumeration:

```
enum typelex{
PLUS,MINUS,MUL,DIV,POWER,ASSIGN,LP,RP,STRING,SEMICOLON,COMMA,ENDPROGRAMM,EOL,
LESS,GREAT,SHARP,BREAKPOINT,
DECINTNUMBER,HEXINTNUMBER,FLOATNUMBER,
```

```
FUNC,OPERATOR,
OLDVARIABLE,OLDARRAY,NEWNAME,
BAD
};
```

defines a lexeme type.

ob_lex::typelex is used in ob_obasic::gettypenextlex(); method.

5.8.5 ob_obasic::gettypenextlex method

The method:

```
ob_lex::typelex ob_obasic::gettypenextlex();
```

is used to simplify input and output operators implementation.

Before the redefined iput-output class methods are run, the execution system selects and identifies the next lexeme. To access to the next lexeme type, gettypenextlex method can be used from the redefined IO class methods. When run method is completed, the access to the next lexeme by gettypenextlex method is not supported.

This method returns the next lexeme. Some operators (for example,INPUT operator) have the same syntax for input out of a file and a console.
The current direction of input is determined by “#” lexeme following the operator name. Gettypenextlex() method enables to determine the next lexeme type. And thereby determine where the input is expected from.

In the example given in section 3.3 “Input-output system review” gettypenextlex method is used to determine the input direction in INPUT operator. If SHARP lexeme is detected the console input is not be performed. The appropriate code to check the next lexeme for PRINT operator is not needed sinec the output does not block the operation.

5.9 Open Basic language syntax. General information.

Open Basic interpreter supports a subset of Basic language operators. These operators are described below. It is possible to operate three types of data:

1. Floating-point variables.
2. Fixed-point variables (integer variables).
3. String variables.

It is possible to operate arrays consisting of these three data types.

All data for variables and arrays are located in free memory by new operator.

Variables in Open Basic do not need to be described before their first use. The first appearance of a variable in a program should be to the left of the assignment operator.

Arrays in Open Basic should be described before their first use by DIM operator.

Arrays can be multidimensional. The maximum size of array depends on new operator restrictions in each specific operating system.

When an array is created by DIM operator, numeric arrays elements are initialized with zeros, string arrays elements are initialized with empty strings.

In Open Basic, the first array index begins with 1.

Unlike standard Basic language, DIM operator in Open Basic is executable. It is executed each time the control is transferred to it.

It is possible to repeat the array description with the same name in DIM operator. A new array can have dimensions that differ from old array dimensions. During this process the old data is lost.

The repeated array description can be used for memory release. For example, after a huge array is created by DIM operator it is possible to release memory by describing the array with the same name and dimension in one element.

When describing the array in DIM operator, not only constants but also integer variables can be used as dimensions.

The range of floating-point and integer data depends on C++ compiler. Usually integer data is int and floating-point data is float. In file ob.h types of variables are determined using typedef:

```
typedef char                    ob_type_char;
typedef float                   ob_type_flo;
typedef int                     ob_type_int;
```

It is possible to redefine type_flo и ob_type_int types by means of conditional translation macros.

The maximal length of string variables is determined by ob_maxlengthstring constant in ob.h file. For Open Basic v 1.94, it is ob_maxlengthstring=4096.

NOTE: The maximum length of constant strings and names in Open Basic can not exceed ob_maxlengthstring. However, this limit is not applied to the operation of adding lines. Therefore, if we add two string variables of ob_maxlengthstring length, the total length of the line will be (2*ob_maxlengthstring). Thus, the operation of adding lines provides lines of any length.

*.bas-program can contain empty strings.

Open Basic interpreter supports the following operations with fixed-point types and floating-point:

+ - addition
- - substraction
* - multiplication
/ - division
^ - exponentiation
- - unary minus
+ - unary plus
= - assignment

The operations priority is standard for C++. The operations priority can be changed by means of brackets.

Open Basic interpreter supports the following relational expressions:

< - "less than"
> - "greater than"
= - "equal"
<> - "not equal"
<= - "less or equal"
>= - "equal or greater"

The relational expressions are used only in IF operator.

In addition, comparison and line assignment operations are supported for string data. The operations of line comparison is performed in accordance with the rules of array comparison in mvect ([mktmk]vect[or]) template class. mvect class is described in mvect.h file.

5.10 Open Basic data types and names.

Any sequences of letters and numbers can be used as variables, arrays and user-defined functions names. A variable or array name should begin with a letter.

'_' underline character, '$' currency character and '%' procentage character are considered to be letters and can be involved in names.

The type of a variable or an array in Open Basic is determined by its name. The rules to determine a variable type in Open Basic.

1. If a variable or array name ends with '%' character, this variable or array is of an integer type.
2. If a variable or array name ends with '$' character, this variable or array is of a string type.
3. If a variable or array name ends with any other character, this variable or array is of a floating-point type.

Each variable and an array is assigned with globality/locality property.
Locality or globality of a variable or array in Open Basic is defined by name as well as the type. The rules to determine a local/global variable in Open Basic:

1. If the first character is '_' underline character in a variable or an array name , it is a local variable or local array.
2. If the first character is not '_' underline character in a variable or an array name , it is a global variable or global array.

Global variables and global arrays are accessible out of any project file.

Local variables and local arrays are accessible out of any file, which they are attached to.

Most Open Basic types are described in obmain.h file by typedef, for example:

1. ob_type_flo type is reflected into float type by typedef.
2. ob_type_int type is reflected into int type by typedef.

In user-defined functions it is recommended to use ob_type_flo and ob_type_int types.
When types are changed, for example, ob_type_flo is changed from float into double types compatibility will be supported.

Letters in a name can be capital or small. Reduction of letters to one register is not performed, so ARRAY, array and ArRay names are different names.

The maximum name length is determined by ob_maxlengthstring constant in ob.hfile. For Open Basic v 1.94, it is ob_maxlengthstring=4096.

Example:

Declaration of array global array and initialization of two global variables — counter% integer variable and a variable of data floating-point type. Note that the last dimension of array array is specified by a variable. array array elements type is a floating-point type. The counter% variable is an integer type. data variable is a floating-point type. All arrays and variables in this example are global as their names do not begin with an underline character character.

```
counter%=7
DIM array(2,3,10,counter%)
data=counter%+37.77
```

Example:

Declaration of _array local array and initialization of two global variables — _counter% integer variable and a variable of _data floating-point type. Note that the last dimension of _array array is specified by a variable. _array array elements type is a floating-point type. The _counter% variable is an integer type. _data variable is a floating-point type. All arrays and variables in this example are local as their names begin with underline character character.

```
_counter%=7
DIM _array(2,3,10,_counter%)
_data=_counter%+37.77
```

Example:

Declaration of array global array and its use. Redeclaration of the array with same name but of other dimension and its use.

```
DIM array(10)

FOR i%=1 TO 10 STEP 1

array(i%)=i%+37.77

NEXT i%

counter%=7

DIM array(counter%,3)

FOR i%=1 TO counter% STEP 1

array(i%,2)=i%+37.77

NEXT i%
```

If different types of data are used in the expression, reduction of types is used. If data of integer type and data of floating-point type are used in the expression the resultant type will be floating-point. During assignment, the type of right expression is reduced to the type of expression on the left of = assignment sign.
If there is an integer variable on the left of the assignment sign and a floating-point variable on the right the fractional part is discarded (according to the rules of rounding in C++).

5.11 Open Basic labels.

Open Basic interpreter supports two types of labels.

1. Labels in the form of sting numbers.
2. String labels with a final colon.

Each label in Open Basic interpreter gets globality/locality property.

Global labels are accessible out of any project file.
Global labels enable to trasmit control into the other project file.
You can trasmit control into the other project file on a global label string by GOTO or GOSUB operators.

Global labels are available from the file which they are defined to.

Jumping by GOTO or GOSUB operators is only possible for a string with a label. You can also place a label on an empty string.

5.11.1 Labels in the form of string numbers

Labels in the form of numbers are always global.

Labels in the form of numbers should be placed at the beginning of a line and separated by minimum one space from the other lexemes. Line numbers do not have to be in order.

Example:

```
    ii%=6
    ik%=8

    IF ii%<>6 THEN GOTO 10

    PRINT "ii%=";ii%;
10  PRINT " ii%!=6"

    IF ik%=7 THEN GOSUB 11
```

```
    PRINT "ik%=";ik%;
    GOTO 12
11
    PRINT " ik%!=7"
    RETURN
12
    STOP
    END
```

Labels in the form of numbers are deprecated. In the following interpreter versions they are not going to be supported. Users should not use labels in the form of line numbers in new products.

5.11.2 String labels with a final colon

Starting from version 1.80 besides labels in the form of numbers Open Basic supports string labels with a colon. A colon is a label identificator part and it should follow the identificator without a space. The string label identificator is written without a final colon in GOTO and GOSUB operators. The program can simaltaneously use labels in the form of string numbers and string labels. A string label with a final colon should begin with a letter.

String labels with a final colon can be local and global.

Locality and globality of a string label with a final colon in Open Basic is defined by a name as well as variables and arrays locality. To be exact:

1. If the first character in a label name is '_' underline character, it is a local label.
2. If the first character in a label name is not '_' underline character, it is a global label.

Example:

The way to create and use global string labels with a final colon.

```
  REM Test global string label

  PRINT
  PRINT "Start--------------------------------"
  PRINT

  GOSUB gosub_label_1

  STOP
  END

gosub_label_1:

  PRINT "Work local label"

  RETURN
```

Example:

The way to create and use local string labels with a final colon.

```
  REM Test local string label

  PRINT
  PRINT "Start--------------------------------"
  PRINT

  GOSUB _gosub_label_1

  STOP
  END

_gosub_label_1:
```

```
PRINT "Work local label"

RETURN
```

Global labels should be unique in all project files.

Local labels should be unique in the file they are attached to.

When repeated global labels are detected in different project files or in a single project file load_project method generates exception.

When repeated local labels are detected in a single project file, load_project methods generate exception.

5.12 User-defined functions in Open Basic.

Open Basic provides system commands expansion by attaching user-defined functions to Open Basic execution system.

Parameters of user-defined functions allow to determine a call parameters type and their sequence during the program execution. User-defined functions are attached to Open Basic execution system under Basic-names. User-defined functions Basic-names do not have to coinside with user-defined functions names in C++. However, it is recommended that the Basic-name of the user-defined function be the same as the names of the user-defined functions in C++. In future versions of Open Basic, it is planned to generate C++ code from the code of *.bas-programs. If the Basic-name of the user-defined functions in the project coincides with the names of the user-defined functions in C++, this will automate the process of generating C++ code.

Each user-defined function is attached under its own Basic-name. After functions are attached they can be called in *.bas-program text using their Basic-names. It is possible to pass parameters to user-defined functions and get results from user-defined functions. User-defined functions can be called without parameters.

User-defined functions can be written on C/C++, assembler or other languages.

All user-defined functions have 2 names:

1. C++ function name. You can call a function from the C++ program by this name.
2. A Basic-name function name. You can call a function from a Basic program by this name.

The first name is obtained by user-defined functions upon their writing in C++.
The second name is obtained by user-defined functions when they are attached to Open Basic execution system using setfun method.

A user-defined function can have one of three prototypes:

1. A function that returns ob_type_flo floating-point value.
2. A function that returns ob_type_int fixed-point value.
3. A function that returns ob_type_char* value.
4. A function that returns ob_type_standartstring value.

Since version 1.94, a new type of user-defined function has been introduced. This is a user-defined function that returns the string ob_type_standartstring. For a version Open Basic without wchar_t support, ob_type_standartstring is equivalent to the standard C ++ string type. For a version Open Basic with wchar_t support, ob_type_standartstring is equivalent to the standard C ++ wstring type. Version 1.94 does not have support for wchar_t.

Starting with version 1.94, the user-defined function returning ob_type_char * is considered obsolete and is not recommended for use. Instead, it is recommended that you use a user-defined function that returns ob_type_standartstring.

To create a user-defined function and attach it to Open Basic execution system, you perform the following actions:

1. Choose the type of returned value for a user-defined function.
2. Choose C++ user-defined function name.
3. Create a user-defined function using one of three prototypes listed above.

4. Choose a Basic-name function name for a user-defined function.
5. Attach a user-defined function to Open Basic execution system by setfun method and a Basic-name function name and C++ function name as parameters.

After that a user-defined function Basic-name is possible to be used in *.bas-program.

User-defined functions are always global.

User-defined functions names should not be repeated.

If several user-defined functions will be attached to Open Basic execution system under one and the same names, the last attached one will be called out of *.bas-program.

It is possible to delete a user-defined function from Open Basoc execution system by delfun method.

5.12.1 Prototypes for a user-defined function

A user-defined function can be one of the following three types:

1. The function returning floating-point value is of the following prototype:

```
ob_type_flo name_flo(
ob_obasic* basic_ptr,
const ob_type_char* parstring,
const ob_type_parnum *descrf,
const ob_type_parnum *descri,
const ob_type_parnum *descrc,
const ob_type_flo* parf,
const ob_type_int* pari,
const ob_type_char* parc
);
```

2. The function returning fixed-point value is of the following prototype:

```
ob_type_int name_int(
ob_obasic* basic_ptr,
const ob_type_char* parstring,
const ob_type_parnum *descrf,
const ob_type_parnum *descri,
const ob_type_parnum *descrc,
const ob_type_flo* parf,
const ob_type_int* pari,
const ob_type_char* parc
);
```

3. The function returning a ob_type_char* value of the following prototype:

```
ob_type_char* name_char(
ob_obasic* basic_ptr,
const ob_type_char* parstring,
const ob_type_parnum *descrf,
const ob_type_parnum *descri,
const ob_type_parnum *descrc,
const ob_type_flo* parf,
const ob_type_int* pari,
const ob_type_char* parc
);
```

4. The function returning a ob_type_standartstring value of the following prototype:

```
ob_type_standartstring name_char(
ob_obasic* basic_ptr,
```

```
const ob_type_char* parstring,
const ob_type_parnum *descrf,
const ob_type_parnum *descri,
const ob_type_parnum *descrc,
const ob_type_flo* parf,
const ob_type_int* pari,
const ob_type_char* parc
);
```

These prototypes differ only in a type of returned value.

Starting with version 1.94, the user-defined function returning ob_type_char * is considered obsolete and is not recommended for use. Instead, it is recommended that you use a user-defined function that returns ob_type_standartstring.

ob.h file contains OB_DECLARE_USER_FUNCTION(name,typeret) macro to be used for declaring user-defined functions. name parameter is a function name in C++, typeret parameter is a returned value type.

Example:

Declaration of a user-defined function by OB_DECLARE_USER_FUNCTION macro. C++ function name is myfun1. The type of returned value is ob_type_flo.

```
OB_DECLARE_USER_FUNCTION(myfun1,ob_type_flo);
```

Example:

Determination of a user-defined function by OB_DECLARE_USER_FUNCTION macro. C++ function name is myfun2. The type of returned value is ob_type_int.

```
OB_DECLARE_USER_FUNCTION(myfun2,ob_type_int){
return 365;
}
```

Example:

Determination of a user-defined function by OB_DECLARE_USER_FUNCTION macro. C++ function name is myfun3. The type of returned value is ob_type_char*.

```
OB_DECLARE_USER_FUNCTION(myfun3,ob_type_char*){
return "Hello world";
}
```

5.12.2 User-defined functions parameters

A user-defined functions interface in Open Basic is specially developed to provide ability to control actual parameters in calling during program execution.

The set of user-defined functions parameters contains two sets of arrays.

The first array set contains three arrays of descrf, descri, descrc parameters descriptions. descrf, descri, descrc arrays contain information about the type, quantity and sequence of actual parameters in user-defined function calling.

The second array set contains three arrays of parf, pari, parc parameters values. parf, pari, parc arrays contain values of actual parameters in user-defined function calling.

The purpose and structure of descrf, descri, descrc и parf, pari, parc arrays are described in detail in the next section.

5.12.3 Detailed description of user-defined functions parameters

There are three types of user-defined functions. All these types have the same parameters and different types of a returned result.
We illustrate user-defined function parameters by an example of a function that returns floating-point value.

```
ob_type_flo name_flo(
ob_obasic* basic_ptr,
const ob_type_char* parstring,
const ob_type_parnum *descrf,
const ob_type_parnum *descri,
const ob_type_parnum *descrc,
const ob_type_flo* parf,
const ob_type_int* pari,
const ob_type_char* parc
);
```

The user-defined functions parameters:

basic_ptr	a pointer to ob_obasic interpereter instance, which the function is attached to. It is used to access program variables by ob_obasic methods
parstring	string representations of parameters
descrf, descri, descrc	three arrays of parameters description
parf, pari, parc	three arrays of parameters values

parstring array name is derived from par[ameters] string words.

descrf array name is derived from descr[iption] f[loat] words.
descry array name is derived from descr[iption] i[nt] words.
descrc array name is derived from descr[iption] c[har] words.

parf array name is derived from par[ameters] f[loat] words.
pari array name is derived from par[ameters] i[nt] words.
parc array name is derived from par[ameters] c[har] words.

5.12.3.1 parstring parameter structure

parstring parameter contains all parameters of the current call of user-defined functions in the form of strings.

Example:

If USER_FUN1 function is called with USER_FUN1(1.123,a_a%,1+2) parameters, a parstring array will contain the following strings:

"1.123","a_a%","1+2".

These strings will be consecutive, each line will end in a null byte.

5.12.3.2 descrf parameter structure (descri and descrc have similar structures)

The null element of descrf array contains the whole number of floating-point parameters in current calling. The rest descrf array elements are ordinal numbers of floating-point parameters in current calling. The ordinal numbers of actual parameters in descrf array are counted starting from 1.

Example:

a_a%	integer variable
1.123	floating-point constant
1+2	integer experession

If USER_FUN1 function is called with USER_FUN1(1.123,a_a%,1+2) parameters,

1. descrf array will contain the numbers: 1,1.
2. descri array will contain the numbers: 2,2,3.
3. descrc array will contain the numbers: 0.

Example:

"str1"	string variable
"str2"	string variable
"str3"	string variable
1.1	floating-point constant
1.2	floating-point constant
1.3	floating-point constant
1.4	floating-point constant

If USER_FUN1 function is called with USER_FUN1("str1","srt2"+"str3",1.1,1.2,1.3,1.4) parameters,

1. descrf array will contain the numbers: 4,3,4,5,6.
2. descri array will contain the numbers: 0.
3. descrc array will contain the numbers: 2,1,2.

5.12.3.3 parf parameter structure (pari and parc have similar structures)

There are three types of the parameters value:

const ob_type_flo* parf	values array of floating-point parameters
const ob_type_int* pari	values array of integer parameters
const ob_type_char* parc	values array of string parameters

parf, pari, parc arrays are the arrays describing the values of all corresponding types parameters transferred to a user-defined function in current calling from *.bas-program.

The strings in parc array as in parstring array are separated by a null byte.

Example:

a_a%	integer variable
[a_a%]	variable value named a_a%
1.123	floating-point constant
1+2	integer expression

If USER_FUN1 function is called with USER_FUN1(1.123,a_a%,1+2) parameters,

1. parf array will contain the numbers: 1.123.
2. pari array will contain the numbers:[a_a%],3.
3. parc array will not contain any values.

Example:

"str1"	string variable
"str2"	string variable
"str3"	string variable
1.1	floating-point constant
1.2	floating-point constant
1.3	floating-point constant
1.4	floating-point constant

If USER_FUN1 function is called with USER_FUN1("str1","str2"+"str3",1.1,1.2,1.3,1.4) parameters,

1. parf array will contain the numbers: 1.1,1.2,1.3,1.4.
2. pari array will not contain any values.
3. parc array will contain the strings: "str1","srt2str3".

The maximum number of parameters in a user-defined function is determined by ob_maxnumpar constant in ob.h file. For Open Basic v 1.94 it is ob_maxnumpar=64;.

5.12.3.4 Use of user-defined functions parameters

Using descrf, descri, descrc arrays and parf, pari, parc arrays a user can obtain control of types and control of parameters sequence order in current user-defined function calling.
The analysis descrf, descri, descrc arrays gives entire information about the types, quantity and sequence order of parameters in current user-defined function calling.

Example:

A string variable is transferred as a parameter into a user-defined function that calculates sine. At the same time descrf, descri and descrc arrays will contain the following values:

```
descrf[0]=0
descri[0]=0
descrc[0]=1
```

Analyzing this data in descrf, descri and descrc arrays a user can perform some actions, for example:

1. Generate user exception.
2. Substitute in a default argument, for example 0.
3. Perform a program halt with printing alert.
4. Try to transform a string into a number by atof.

To access the string data Open Basic provides ob_getstringparam function:

const ob_type_char* ob_getstringparam(const ob_type_parnum *descrc,const ob_type_char* parc,const ob_type_parnum i);

ob_getstringparam function parameters:

descrc	an array of string parameters description
parc	an array of string parameters values
i	a number of a string transferred to a user-defined function (numeration starts with zero)

Using ob_getstringparam function you can access the i-th string, which is recorded in string parameters. If i is greater or equal to descrc[0], the function returns a null pointer. If descrc[0] is equal to zero, the function returns a null pointer.

ob_getstringparam function is not ob_obasic class method.

5.12.4 Choosing a Basic-name for a user-defined function

A Basic-name for a user-defined function should differ from the names of operators, variables, arrays or other user-defined functions of *.bas-program.

5.13 Attaching a user-defined function to Open Basic execution system (general information)

Attaching a user-defined function to Open Basic execution system is performed by four ob_obasic::setfun methods.

setfun methods prototypes:

```
bool ob_obasic::setfun(ob_type_char* name,ob_type_flofun f);.
bool ob_obasic::setfun(ob_type_char* name,ob_type_intfun f);.
bool ob_obasic::setfun(ob_type_char* name,ob_type_charfun f,ob_type_del dt);.
bool ob_obasic::setfun(ob_type_char* name,ob_type_stringfun f);.
```

Starting with version 1.94, the user-defined function ob_type_charfun returning ob_type_char * is considered obsolete and is not recommended for use. Instead, it is recommended that you use a user-defined function ob_type_stringfun that returns ob_type_standartstring.

setfun methods parameters:

name	a Basic-name of a user-defined function
f	a pointer to a user-defined function
dt	a type of returned value allocation in memory (only for functions that return a string)

Example:

userfun1 user-defined function is attached to an interpreter instance.
userfun1 user-defined function return floating-point value.
userfun1 user-defined function prints a message "Hello world" into the standard input stream and returns 3.1416 value.
When userfun1 user-defined function is attached to an interpreter instance it gets TEST_FUN1 Basic-name.

Creating userfun1 user-defined function by OB_DECLARE_USER_FUNCTION macro.

```
using ob_charspace; namespace

OB_DECLARE_USER_FUNCTION(userfun1,ob_type_flo){
cout<<endl<<"Hello world";
return (ob_type_flo)3.1416;
}
```

Creating an interpreter instance and attaching userfun1 user-defined function by setfun method.

```
void main(){

ob_obasic onebasic;//an interpreter instance named onebasic creating

onebasic.setfun("TEST_FUN1",userfun1);//userfun1 user-defined function attaching to the interpreter
instance under TEST_FUN1 Basic-name

ifstream inonebasicf("EXAMPLE.BAS",ios::binary);//EXAMPLE.BAS file opening

onebasic.clear_project();//the project cleaning
```

```
onebasic.load_project(&inonebasicf,0);//EXAMPLE.BAS file loading into the interpreter

onebasic.run();//*.bas-program running

}
```

The function returning a string has difference in order of attaching. The function returning a string has two types of memory allocation of returned string value.

1. Allocation of returned string value in the heap by new[] operator.
2. Allocation of returned string value in static memory.

In the first case, a user-defined function is attached with dt=OB_DELETE parameter. After this function finishes its operation Open Basic will free the memory by delete[] operator. new[] operator in a user-defined function should be compatible with delete[] operator in Open Basic. It is new[] operator, not delete[] operator, that should be used in a user-defined function, since the memory is freed in Open Basic by delete[] operator, not delete operator.

In the second case, a user-defined function is attached with dt=OB_NODELETE parameter. After this function finishes its operation, Open Basic will not free memory by delete[] operator.

Example:

In the example, an interpreter instance is created, a file named EXAMPLE.BAS is opened and loaded into the interpreter. After that four TEST_FUN1, TEST_FUN2, TEST_FUN3, TEST_FUN4 functions are attached to the execution system.

1. TEST_FUN1 function prints a list of its floating-point parameters and returns floating-point value that is equal to its sum of floating-point parameters.

2. TEST_FUN2 function prints a list of its integer parameters and returns integer value that equals to its sum of fixed-point parameters.

3. TEST_FUN3 function prints its first string parameter and returns "This is string" string. TEST_FUN3 function allocates its returned value in the heap by new[]operator.

4. TEST_FUN4 function prints its first string parameter and returns "This is string" string. TEST_FUN4 function allocates its returned value in static memory.

All these functions control the types of their parameters. If parameters of the required type are missing, the functions print the messages.

```
#include <ob.h>

char* s="This is string";

OB_DECLARE_USER_FUNCTION(userfun1,ob_type_flo){

ob_type_flo summ=0;
ob_type_parnum i;

if(descrf[0]==0) cout<<endl<<"Not float parameters";

for(i=0;i!=descrf[0];i++){
summ+=parf[i];
cout<<endl<<"parameters["<<i<<"]="<<parf[i];
}

return summ;
}

OB_DECLARE_USER_FUNCTION(userfun2,ob_type_int){

ob_type_int summ=0;
ob_type_parnum i;

if(descri[0]==0) cout<<endl<<"Not int parameters";

for(i=0;i!=descri[0];i++){
summ+=pari[i];
```

```
cout<<endl<<"parameters["<<i<<"]="<<pari[i];
}

return summ;
}

OB_DECLARE_USER_FUNCTION(userfun3,char*){

if(descrc[0]==0) cout<<endl<<"Not string parameters";

cout<<endl<<parc;
char* p=new char[strlen(s)+1];
strcpy(p,s);

return p;
}

OB_DECLARE_USER_FUNCTION(userfun4,char*){

if(descrc[0]==0) cout<<endl<<"Not string parameters";
cout<<endl<<parc;

return s;
}

void main(){

ob_obasic onebasic;//instance interpreter named onebasic creating

onebasic.setfun("TEST_FUN1",userfun1);// TEST_FUN1 function attaching
onebasic.setfun("TEST_FUN2",userfun2);// TEST_FUN2 function is attaching
onebasic.setfun("TEST_FUN3",userfun3,OB_DELETE);// TEST_FUN3 function attaching
onebasic.setfun("TEST_FUN4",userfun4,OB_NODELETE);// TEST_FUN4 function attaching

ifstream inonebasicf("EXAMPLE.BAS",ios::binary);// EXAMPLE.BAS file opening

onebasic.clear_project();//project is cleaned
onebasic.load_project(&inonebasicf,0);//EXAMPLE.BAS file loading into the interpreter

onebasic.run();// *.bas-program running
}
```

When the attached function name is detected in *.bas-program, Open Basic performs the following actions:

1. It calculates all parameters in current calling of *.bas-program.
2. It fills descrf, descri and descrc arrays.
3. It fills parf, pari, parc arrays.
4. It fills parstring parameter.
5. It transfers control to a user-defined function.

A user-defined function should control types, quantity and sequence order of parameters in current calling.

5.14 ob_obasic class methods and enumerations to attach and detach user-defined functions

5.14.1 ob_type_del enumeration

The enumeration:

```
enum ob_type_del{OB_DELETE,OB_NODELETE};
```

determines the type of allocation in memory of value returned by a used-defined function.

enum ob_type_del enumeration is used in setfun method to attach user-defined functions returning ob_type_char* type.

If dt=OB_DELETE, returned value is put to free memory by new operator. After user-defined function is terminated, this value will be deleted by delete[] operator. new[] operator in a user-defined function should be compatible with delete[] operator in Open Basic. It is new[] operator, not new operator, that should be used in a user-defined function since memory is freed in Open Basic by delete[] operator, not delete operator.

If dt=OB_NODELETE, returned value will not be deleted by delete[] operator.

ob_type_del enumeration is not a member of ob_obasic class.

NOTE: Misuse of ob_type_del parameter results in the crash of the program.

5.14.2 ob_obasic::setfun methods

The methods

```
bool ob_obasic::setfun(ob_type_char* name,ob_type_flofun f);
bool ob_obasic::setfun(ob_type_char* name,ob_type_intfun f);
bool ob_obasic::setfun(ob_type_char* name,ob_type_charfun f,ob_type_del dt);
bool ob_obasic::setfun(ob_type_char* name,ob_type_stringfun f);.
```

attach a user-defined function to Open Basic execution system.

Starting with version 1.94, the user-defined function ob_type_charfun returning ob_type_char * is considered obsolete and is not recommended for use. Instead, it is recommended that you use a user-defined function ob_type_stringfun that returns ob_type_standartstring.

The parameters:

name	a Basic-name of a user-defined function
f	a pointer to a user-defined function
dt	a type of returned value allocation in memory (only for functions that return a string)

If the function named name has not been attached to Open Basic execution system yet, the method attaches the function named name and returns false.
If the function named name has already been attached to Open Basic execution system, the method replaces the function named name for a new one and returns true.

Types ob_type_flofun, ob_type_intfun and ob_type_charfun, ob_type_stringfun are pointers to functions described in ob.h. The user-defined functions differ in the type of returned value.

5.14.3 ob_obasic::checkfun method

The method

```
bool ob_obasic::checkfun(ob_type_char* name);
```

checks availability of a user-defined function in Open Basic.

The parameters:

name	a Basic-name of a user-defined function

Checkfun method returns true value, if the function named name has already been attached to Open Basic interpreter instance.

Checkfun method returns false value, if the function named name has not been attached to Open Basic interpreter instance yet.

5.14.4 ob_obasic::delfun method

The method

```
bool delfun(ob_type_char* name);
```

This method detaches user-defined function from Open Basic execution system.

The parameters:

name	a Basic-name of a user-defined function

delfun method returns true if the function has been successfully detached from Open Basic interpreter instance.
elfun method returns false if such function does not exist in Open Basic interpreter instance.

*5.15 Access to *.bas program data from a user-defined function. General information*

A user-defined function can receive parameters from *.bas-program.
A user-defined function can send output value to *.bas-program.
A user-defined function can have read/write access to local and global variables and arrays of *.bas-program.

If a variable or an array is used in *.bas-program, a user-defined function is able to read and write this variable and array elements.

A user-defined function has means to check availability of a variable with a specified name in the program.
A user-defined function has means to create variables with a specified name in the variables table of *.bas-program.

All options to access *.bas-program data are implemented by ob_obasic class methods.

ob_obasic class methods to access local and global variables and local and global arrays can be divided into 2 groups:

The first group includes proper methods to access data. There are 12 methds in the first group:

```
ob_type_ident typevar(ob_type_char* name,ob_type_istreamcommon* i=OB_NULL).

ob_type_ident typearray(ob_type_char* name,ob_type_istreamcommon* i=OB_NULL).

void createvar(ob_type_char* name,ob_type_istreamcommon* i=OB_NULL).

void createarray(ob_type_char* name,ob_type_arraydimension kr,ob_type_arraysize* as,ob_type_istreamcommon*
i=OB_NULL).

ob_type_ident strlenvar(ob_type_char* name,ob_type_stringsize* len,ob_type_arraysize*
as=OB_NULL,ob_type_istreamcommon* i=OB_NULL).

bool writevar(ob_type_char* name,ob_type_flo val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon*
i=OB_NULL).

bool writevar(ob_type_char* name,ob_type_int val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon*
i=OB_NULL).

bool writevar(ob_type_char* name,ob_type_char* val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon*
i=OB_NULL).

bool readvar(ob_type_char* name,ob_type_flo* val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon*
i=OB_NULL).
```

```
bool readvar(ob_type_char* name,ob_type_int* val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon*
i=OB_NULL).

bool readvar(ob_type_char* name,ob_type_char* val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon*
i=OB_NULL).

const ob_type_arraysize* getarraysize(ob_type_char* name,ob_type_arraydimension* kr,ob_type_istreamcommon*
i=OB_NULL).
```

The second group includes helper methods, which provide the first group methods with information.
There are 2 methods in the second group:

```
ob_type_istreamcommon* get_current_input_stream().

bool local_name_detect(ob_type_char* name).
```

name parameter in all first group methods is a name of Open Basic variable or array.
ob_type_istreamcommon parameter in all first group methods is a pointer to a stream when accessing a local variable.

The rules of maintaining local and global variables in the first group methods:

1. At first name name is analysed on locality or globality. If name name is global, the search is performed in the tables of global variables and arrays independent of the value of ob_type_istreamcommon parameter.

2. If name name is local, the search is performed in the tables of local variables and arrays, which belong to the stream (file) specified in ob_type_istreamcommon parameter.

3. If name name is local and ob_type_istreamcommon parameter equals to zero, the search fails.

To apply access methods correctly you should implement the following algorithm:

1. You get a pointer to the current stream by get_current_input_stream() methods.
2. You call a method of access with the received stream as i parameter.

Such algorithm guarantees that the search of local and global name will be performed correctly.

local_name_detect(ob_type_char* name) method enables to determine locality or globality of a name.

If you seek a local name in a stream, which is not current, the pointer to a stream should be got from external (regarding the interpreter) procedure.

The parameters of methods to access variables and arrays are described in detail in the next sections.

5.16 ob_obasic class methods and enumerations to access Open Basic data from user-defined functions

The methods described in this Section are usually run in user-defined functions so in the given examples these methods are run by ob_obasic* basic_ptr pointer.

basic_ptr name has a pointer to Open Basic interpreter instance in all user-defined functions if you determine user-defined functions by OB_DECLARE_USER_FUNCTION macro.

Generally the methods described in this section can be run in any part of C++ program, which the interpreter is used. For example, after *.bas-program halts by a breakpoint you can print values of variables and arrays.

You can create and initialize variables and arrays before the program starts by methods and enumerations described in this Section.

5.16.1 ob_obasic class methods and enumerations determine a type of Open Basic variables and arrays

5.16.1.1 ob_type_ident enumeration

The enumeration:

```
enum ob_type_ident{OB_IDENTFLO,OB_IDENTINT,OB_IDENTSTR,OB_NOIDENT};
```

determines a type of a variable or array.

ob_type_ident enumeration is used in typevar and typearray methods to determine a type of a variable or array.

OB_IDENTFLO	means that a variable is of a floating-point type
OB_IDENTINT	means that a variable is of an integer type
OB_IDENTSTR	means that a variable is of a string type
OB_NOIDENT	means that a variable is not found

ob_type_ident enumeration is not a member of ob_obasic class.

5.16.1.2 ob_obasic::typevar method

The method:

```
ob_type_ident ob_obasic::typevar(ob_type_char* name,ob_type_istreamcommon* i=OB_NULL);
```

returns a type of a variable named name.

The parameters:

name	a Basic-variable name
i	a pointer to a stream where a variable is sought (for global variables a stream pointer can equal to zero)

The returned value:

OB_IDENTFLO	means that a variable is of a floating-point type
OB_IDENTINT	means that a variable is of an integer type
OB_IDENTSTR	means that a variable is of a string type
OB_NOIDENT	means that a variable is not found

5.16.1.3 ob_obasic::typearray method

The method

```
ob_type_ident ob_obasic::typearray(ob_type_char* name,ob_type_istreamcommon* i=OB_NULL);
```

returns the type of an array named name.

The parameters:

name	a Basic-array name
i	a pointer to a stream where a array is sought (for global arrays a stream pointer can equal to zero)

The returned value:

OB_IDENTFLO	means that a array is of a floating-point type
OB_IDENTINT	means that a array is of an integer type
OB_IDENTSTR	means that a array is of a string type
OB_NOIDENT	means that a array is not found

Example:

```
    DIM array1%(10,10), array2$(10,10), array3(10,10)

    a%=10
    b$="string example"
    c=101.4
```

In the example, global variables and global arrays are determined.
The access to them is performed in the following way.

```
ob_type_ident ident;

ident=basic_ptr->typevar("a%");//after this line is executed, ident has OB_IDENTINT value
ident=basic_ptr->typevar("b$");//after this line is executed, ident has OB_IDENTSTR value
ident=basic_ptr->typevar("c"); //after this line is executed, ident has OB_IDENTFLO value

ident=basic_ptr->typearray("array1%");//after this line is executed, ident has OB_IDENTINT value
ident=basic_ptr->typearray("array2$");//after this line is executed, ident has OB_IDENTSTR value
ident=basic_ptr->typearray("array3"); //after this line is executed, ident has OB_IDENTFLO value
```

Where basic_ptr is the first parameter in user-defined functions, the pointer to the interpreter instance.

The example of *.bas-program does not contain a variable named VAR and an array named ARR.

So, the use of typevar and typearray methods will give the following results:

```
ident=basic_ptr->typevar("VAR");//after this line is executed, ident has OB_NOIDENT value
ident=basic_ptr->typearray("ARR");//after this line is executed, ident has OB_NOIDENT value
```

To access local variables and local arrays, you should have a pointer to a stream where a local variable and a local array are determined.

Example:

```
    DIM _array1%(10,10), _array2$(10,10), _array3(10,10)

    _a%=10
    _b$="string example"
    _c=101.4
```

In the example, local variables and local arrays are determined. The access to them is performed in the following way.

```
ob_type_istreamcommon* stream_ptr=basic_ptr->get_current_input_stream();
```

```
ob_type_ident ident;

ident=basic_ptr->typevar("_a%",stream_ptr);//after this line is executed, ident has OB_IDENTINT value
ident=basic_ptr->typevar("_b$",stream_ptr);//after this line is executed, ident has OB_IDENTSTR value
ident=basic_ptr->typevar("_c",stream_ptr); //пafter this line is executed, ident has OB_IDENTFLO value

ident=basic_ptr->typearray("_array1%",stream_ptr);//after this line is executed, ident has OB_IDENTINT
value
ident=basic_ptr->typearray("_array2$",stream_ptr);//after this line is executed, ident has OB_IDENTSTR
value
ident=basic_ptr->typearray("_array3",stream_ptr); //after this line is executed, ident has OB_IDENTFLO
value
```

If variables are determined not in the current stream executed by the interpreter, you can not get a stream pointer by get_current_input_stream method. In this case a stream pointer should be passed through external (regarding the interpreter) procedures.

5.16.2 ob_obasic class methods to determine the length of Open Basic string variables and string arrays elements

The method

```
ob_type_ident strlenvar(ob_type_char* name,ob_type_stringsize* len,ob_type_arraysize*
as=OB_NULL,ob_type_istreamcommon* i=OB_NULL).
```

determines a variable type and the length of a variable or array element named name.

The parameters:

name	a name of a Basic-variable or a Basic-array
*len	an output parameter. For string variables, this parameter contains string length without a terminating null. For floating-point or integer variables, this parameter contains sizeof(ob_type_float) or sizeof(ob_type_int) correspondently
*as	current array element indexes if name is the name of an array)
i	a pointer to a stream where a variable is sought (for global variables and global arrays, a pointer can equal to zero)

For floating-point and integer variables, strlenvar method operation result depends on the platform where Open Basic interpreter is compiled and variables type in Open Basic. For example, for a platform x86-32 sizeof(float) will equal to 4 and sizeof(int) will equal to 4, as well.

strlenvar method can be called for Basic-variables or elements of Basic-arrays. If the parameter as equals to zero, name name is sought in the variable table.

strlenvar method returns OB_NOIDENT if there is name assignment error (for example if a name does not begin with a letter or a name is a name of array not created yet at the moment of method calling or there is no such variable.

In case of normal termination strlenvar method returns a variable type.

*len parameter for string variables contains string length without a terminating null.
*len parameter for floating-point variables contains sizeof(ob_type_float).
*len parameter for fixed-point variables contains sizeof(ob_type_int).

Example:

Calling strlenvar method for an array element with specifying current indexes in *as parameter.

If there is *.bas-program:

```
    DIM array2$(10,10)
    array2$(5,5)="array string example"
```

after this part of the code is executed in user-defined functions attached to the interpreter, use of strlenvar method will give the following results:

```
ob_type_stringsize len;

ob_type_ident ident;

ob_type_arraysize as[]={5,5};

ident=basic_ptr->strlenvar("array2$",&len,as);//ident==OB_IDENTSTR len==20
```

Example:

Calling strlenvar method for local variables.

If there is *.bas-program:

```
    _a%=10
    _b$="string example"
    _c=101.4
```

after this part of the code is executed in user-defined functions attached to the interpreter, using strlenvar method will give the following results:

```
ob_type_istreamcommon* stream_ptr=basic_ptr->get_current_input_stream();

ob_type_ident ident;
ob_type_stringsize len;

ident=basic_ptr->strlenvar("_a%",&len,0,stream_ptr);// ident==OB_IDENTINT len==4
ident=basic_ptr->strlenvar("_b$",&len,0,stream_ptr);// ident==OB_IDENTSTR len==14
ident=basic_ptr->strlenvar("_c",&len,0,stream_ptr); // ident==OB_IDENTFLO len==4
```

5.16.3 ob_obasic class methods to determine Open Basic arrays sizes

The method:

```
const ob_type_arraysize* getarraysize(ob_type_char* name,ob_type_arraydimension* kr,ob_type_istreamcommon* i=OB_NULL).
```

returns the number of array dimensions and the size of each dimension.

The parameters:

name	an array name
*kr	an output parameter of the number of array dimensions (it is filled by getarraysize method)
i	a stream pointer where a variable is sought (for global arrays a stream pointer can equal to zero)

The returned parameter is an array of sizes for each dimension. It is created during describing the array by DIM operator. This parameter is Open Basic internal buffer and should be read-only.

NOTE: Change of the returned array will result in the crash of the program.

Example:

Use of getarraysize method for a global array.

```
DIM Array_1(10,11,12).
```

```
const ob_type_arraysize *sr;
ob_type_arraydimension kr;

sr=basic_ptr->getarraysize("Array_1",&kr);
```

After this C++ code is executed, getarraysize kr=3 and sr pointer points to {10,11,12} array.

Example:

Use of getarraysize method for a local array.

```
DIM _Array_1(10,11,12).
```

```
ob_type_istreamcommon* stream_ptr=basic_ptr->get_current_input_stream();

const ob_type_arraysize *sr;
ob_type_arraydimension kr;

sr=basic_ptr->getarraysize("_Array_1",&kr,stream_ptr);
```

After this C++ code is executed, getarraysize kr=3 and sr pointer points to {10,11,12} array.

5.16.4 ob_obasic class methods to write Open Basic variables

The methods:

```
bool writevar(ob_type_char* name,ob_type_flo val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon*
i=OB_NULL).

bool writevar(ob_type_char* name,ob_type_int val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon*
i=OB_NULL).

bool writevar(ob_type_char* name,ob_type_char* val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon*
i=OB_NULL).
```

writes a value into a variable with the specified name or into an array element with the specified name.

readvar methods are redefined for multiple types of the second parameter.

The parameters:

name	a name of a Basic-variable or Basic-array
val	values written into variables or array elements
*as	arrays indexes if name is the name of an array
i	a stream pointer where a variable is sought (for global variables and global arrays a stream pointer can equal to zero)

writevar methods can be called for Basic-variables or Basic-arrays elements. If as parameter equals to zero, name name is sought in the variables tables. If as parameter does not equal to zero, name name is sought in the arrays tables.

writevar methods write a variable named name into a table of variables and assign val value to it. If a variable named name does not exist, it is created. When a variable is created, the name type is checked, i.e. a type variable is created that corresponds to the name.

If the type of assigned value does not correspond to the name type (for example, when trying to create an integer type variable with the second ob_type_char* val argument), the method generates an exception.

If writevar method calls an array that does not exist, an array is not created. You should create an array either by DIM operator or createarray method.

writevar methods return false if there is a name assignment error, for example, if it does not begin with a letter or is a name of an array not described yet at the moment of array use. The methods return true in case of normal termination.

Example:

The following code should be written in a user-defined function to assign value 10.4 to _VAR local variable:

```
ob_type_istreamcommon* stream_ptr=basic_ptr->get_current_input_stream();

if(basic_ptr->writevar("_VAR",10.4,0,stream_ptr)==false) cout<<endl<<"error in writevar";
```

Example:

The following code should be written in user-defined function to assign string value "Hello world" to the third element of array_string$(10) local string array:

```
ob_type_istreamcommon* stream_ptr=basic_ptr->get_current_input_stream();

ob_type_arraysize as[]={3};

if(basic_ptr->writevar("_array_string$","Hello world",as,stream_ptr)==false) cout<<endl<<"error in 
writevar";
```

In this example, as parameter does not equal to zero since the element index is indicated not in name name but in as parameter. Note that _array_string$(10) local string array is redefined in a current input stream and a pointer to a current input stream is called beforehand by get_current_input_stream method.

5.16.5 ob_obasic class methods to read Open Basic variables

The methods

```
bool readvar(ob_type_char* name,ob_type_flo* val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon* 
i=OB_NULL).

bool readvar(ob_type_char* name,ob_type_int* val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon* 
i=OB_NULL).

bool readvar(ob_type_char* name,ob_type_char* val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon* 
i=OB_NULL).
```

read value from a variable with the specified name or from an array element with the specified name.

readvar methods are redefined for multiple types of the second parameter.

The parameters:

name	a name of a Basic-variable or a Basic-array
*val	value read from a variable or an array element
*as	array indexes if name is the name of an array
i	a stream pointer where a variable is sought (for global variables and global arrays a stream pointer can equal to zero)

readvar methods can be called for Basic-variables or Basic-arrays elements. If as parameter equals to zero, name name is sought in the variables tables. If as parameter does not equal to zero, name name is sought in the arrays tables.

readvar methods reads up a variable named name from a table of variables into val pointer. If a variable named name does not exist, it is not created.

readvar methods return false if there is a name assignment error, for example, if it does not begin with a letter or is a name of an array not described yet at the moment of array use or the variable does not exist. The methods return true in case of normal termination.

If a variable type and val parameter do not coinside (for example, when trying to assign integer value to a string variable), the method generates an exception.

Example:

The following code should be written in a user-defined function to read the global variable named VAR1 and assign value to a variable named val1:

```
ob_type_flo val1;
if(basic_ptr->readvar("VAR1",&val1)==false) cout<<endl<<"error name var in readvar";
```

Example:

The following code should be written to read up the value of the forth element of array_string$(10) global array:

```
ob_type_char buf[ob_maxlengthstring];
ob_type_char* pchar=buf;

ob_type_arraysize as[]={4};

if(basic_ptr->readvar("array_string$",pchar,as)==false) cout<<endl<<"error name var in readvar";
```

pchar buffer should have a size sufficient to copy array_string$(4) array element. The static buffer from the example got the maximum string size in Open Basic that equals to ob_maxlengthstring. However, in an actual program, the length of a string variable can exceed ob_maxlengthstring due to the fact that the length of the strings sum can be larger than ob_maxlengthstring. You should use strlenvar method to determine the length of a string variable and a dynamic buffer to store the read up string value.

5.16.6 ob_obasic class methods to create Open Basic variables and arrays

5.16.6.1 ob_obasic class method to create Open Basic variables

The method:

```
void createvar(ob_type_char* name,ob_type_istreamcommon* i=OB_NULL)
```

creates a variable named name in Open Basic tables.

The parameters:

name	a name of a Basic-variable
i	a stream pointer where a variable is sought (for global variables a stream pointer can equal to zero)

createvar method creates a variable named name in the table of the interpreter variables. A variable type is determined according to the availability of '%' and '$' characters in the array name. If a local variable is created, it is created in the table of local variables for the specified input stream.

5.16.6.2 ob_obasic class methods to create Open Basic arrays

The method:

```
void createarray(ob_type_char* name,ob_type_arraydimension kr,ob_type_arraysize* as,ob_type_istreamcommon* i=OB_NULL)
```

creates an array named name in the table Open Basic interpreter.

The parameters:

name	a name of a Basic-array
kr	number of array dimensions
*as	sizes of an array of each dimension
i	a stream pointer where a variable is sought (for global arrays a stream pointer can equal to zero)

createarray method creates an array named name in a table of the interpreter arrays. If an array with such name already exists, it is deleted and created again with the specified parameters.

The elements type of the created array that is determined according to the availability of '%' and '$' characters in the array name. The method is equivalent to DIM operator. If a local array is created, it is created in the table of local variables for the specified input stream.

Example:

The following code should be written in a user-defined function to create an array of ARRAY1%(5,10,20) integers :

```
ob_type_arraysize as[]={5,10,20};
basic_ptr->createarray("ARRAY1%",3,as);
```

5.16.6.3 Methods of the ob_obasic class to override the OpenBasic name locality feature.

The Starting with the version of Open Basic 1.94, you can override the Open Basic name locality feature. For this, you should use methods:

```
ob_local_name_detect_def ob_obasic::get_local_name_detect();
void ob_obasic::set_local_name_detect(ob_local_name_detect_def ptr);
```

These methods allow you to redefine the internal function of the interpreter, which is called to determine the sign of locality of the name. This function has the following prototype:

```
typedef bool(*ob_local_name_detect_def)(ob_type_char* name);
```

By default, the function with the following implementation is used to determine the name locality feature:

```
bool ob_local_name_detect_default(ob_type_char* name) {
if (name[0] == ob_const_underscore) return true; else return false;
}
```

The user can define his function with such a prototype and thus redefine the sign of locality of names in Open Basic.

Example:

```
bool user_local_name_detect(ob_type_char* name) {
bool ret=false;

if (name[0] == ob_const_underscore)){
if (name[1] == ob_const_underscore)){
ret=true;
}
}

return ret;
}
```

If you use the ob_obasic::set_local_name_detect method to set the user_local_name_detect function in Open Basic as a locality feature, then the names with two leading underscores will be considered local. This applies to names and variables and arrays and labels.

5.17 Description of other ob_obasic class methods

~ob_obasic();	a destructor
static const ob_type_char* about();	it returns a string of information about the program. The string can contain characters of a new string.
static const ob_type_serialnum serial_number();	it returns the serial number of the program
static const ob_type_int version();	it returns the number of the current version of the program multiplied by 100. For version 1.94 this method returns 194.
void set_max_nested_gosub(ob_type_countlist m);	determines the maximum number of nested calls for GOSUB operator. The default number of nested calls for GOSUB operator is 4096. If the number of nested calls exceeds 4096, an exception is generated. If the maximum number of nested calls for GOSUB operator is set to zero, the maximum number of nested calls is not checked.
ob_type_countlist get_max_nested_gosub();	it returns the set maximum number of nested calls for GOSUB operator

Example:

```
cout<<ob_obasic::about();//information about the program is printed

cout<<ob_obasic::serial_number();//serial number of the rpogram is printed

cout<<ob_obasic::version();//version of the program is printed
```

5.18 *Open Basic interpereter operators*

Operators can be placed in the same line and be seperated from each other by spaces.
Some operators like DATA or PRINT should be the only operators in the line since the list of these operators arguments are limited to a character of the line end.

When operators are described, characters in square brackets mean optional characters.

Open Basic interpreter supports the following operators:

5.18.1 PRINT operator

PRINT operator sends the list of variables into the output stream. The output stream is attached to the interpreter by setout method.

PRINT operator format:

```
[N] PRINT [#EXP,][LIST]
```

where:

N	an optional number of a line (label)
LIST	a list of elements for print (may be constants, variables, string or numerical expressions)
EXP	an expression determining a number of the output channel opened by OPEN operator

PRINT operator outputs a character of a new line without a list of elements. If a list element is an expression, Open Basic calculates this expression and prints the result.

Example:

```
10 TT%=40
   TTT=101.3
20 PRINT "TT%=";TT;" TTT=";TTT+0.3
```

The result of this example:

```
TT%=40 TTT=101.6
```

List elements are separated from each other by commas or semicolons. If list elements are separated by commas, the tab character is output between list elements. If list elements are separated by semicommas, no separators are output between list elements.

If there is a comma or a semicolon at the end of the elements list, a line is not passed after PRINT operator.

PRINT operator with an elements list should be the only operator in a line.

5.18.2 INPUT operator

This operator inputs data from the input stream. The input stream is attached to the interpreter by setin method.

INPUT operator format:

```
[N] INPUT [#EXP,]VAR1,[VAR2,VAR3,...]
```

where:

N	an optional number of a line (label)
VAR1,[VAR2,VAR3,...]	a list of variables
EXP	an expression determining a number of an input channel opened by OPEN operator

When Open Basic meets INPUT operator, it inputs data represented in a numerical form from the stream and assigns it to variables from the list. The strings in the stream should be quoted.

Example:

```
10 TT%=40
   TTT=101.3
20 INPUT TT%,TTT
```

5.18.3 FOR and NEXT operators

FOR and NEXT operators enable to create a cycle.

FOR operator format:

```
[N] FOR VAR=EXP1 TO EXP2 [STEP EXP3]
```

where:

N	an optional number of a line (label)
VAR	a controling variable (cycle index)
EXP1	initial index value (any numerical expression)
EXP2	final index value (any numerical expression)
EXP3	index value increment (any numerical expression, it can be positive or negative, by default it is 1)

A controling variable (cycle index) can be an array element.

FOR and NEXT operators are used only together. FOR operator determines cycle start and NEXT operator determines cycle end.

NEXT operator format:

```
[N] NEXT VAR1
```

where:

N	an optional number of a line (label)
VAR1	a controling variable used in FOR operator

If initial value of a cycle variable is greater than its final value, cycle is not performed.
Transferring control inside the cycle is not allowed.

To avoid rounding errors it is recommended to use integer variables as cycle variables.

Cycles can be nested into each other, the inner cycle should end before the outer cycle.

Example:

```
    FOR i%=1 TO 3
    FOR ii%=4 TO 1 STEP -2

    PRINT "Work FOR operator";" ii%=";ii%;" i%=";i%

    NEXT ii%
    NEXT i%
```

The result of this example:

```
Work FOR operator ii%=4 i%=1
Work FOR operator ii%=2 i%=1
Work FOR operator ii%=4 i%=2
Work FOR operator ii%=2 i%=2
Work FOR operator ii%=4 i%=3
Work FOR operator ii%=2 i%=3
```

As Open Basic is an interpreter, FOR operator has some implementation features.

They take place because Open Basic itself is an interpreter. That is, in case of entering the cycle, cycle conditions are checked. If a cycle condition is not met, the cycle will not be performed at all. In this case corresponding NEXT operator is sought and the control is transfered to it. During these actions any possible branching of the algorithm by GOTO operators are not taken into account.

Example:

```
    GOTO 30

10

    NEXT i%

    GOTO 20

30 FOR i%=10 TO k% STEP 1

    GOTO 10


20
    STOP
    END
```

This example will operate correctly if k% is greater than 10. If k% is less or equals to 10, emergency halt will happen since NEXT operator is sought only in the direction from FOR operator to the file end. In the given example NEXT operator will not be found.

FOR and NEXT operators are recommended to be used in accordance with their purpose, the way NEXT operator would always follow FOR operator corresponding to it.

5.18.4 GOTO operator

GOTO operator calls direct jump to the specified line in the abnormal order of the program operators operation.

GOTO operator format:

```
[N] GOTO line_number
```

or

```
[N] GOTO label_string
```

where:

N	an optional number of a line (label)
line_number	a number of a line which jump is performed to
label_string	a label

Example:

```
20 GOTO 50
   PRINT "This operator do not work in example"
50 PRINT "Work GOTO operator"
```

The result of this example:

```
Work GOTO operator
```

Example:

```
20       GOTO label_1
         PRINT "This operator do not work in example"
label_1: PRINT "Work GOTO operator"
```

The result of this example:

```
Work GOTO operator
```

5.18.5 GOSUB and RETURN operators

GOSUB and RETURN operators bind the program with its subprogram.

GOSUB operator format:

```
[N] GOSUB line_number
```

or

```
[N] GOSUB label_string
```

where:

N	an optional number of a line (label)
line_number	a number of a line which jump is performed to
label_string	a label which jump is performed to

When Open Basic meets GOSUB operator, it transfers control to the line specified in GOSUBoperator. The program continues to run from this line. When Open Basic meets RETURN operator, the control is transferred to the line following the line containing GOSUB operator.

GOSUB and RETURN operators should be used only together.

RETURN operator format:

```
[N] RETURN
```

where:

N	the number of the line

Subgrams can be nested into each other.

Example:

```
   GOSUB 10
   GOTO 100

10
   PRINT "Work GOSUB operator 1"
   GOSUB 20
   PRINT "Work GOSUB operator 2"
   RETURN
20
   PRINT "Work GOSUB operator 3"
   PRINT "Work GOSUB operator 4"
   RETURN

100

   STOP
   END
```

The result of this example:

```
Work GOSUB operator 1
Work GOSUB operator 3
Work GOSUB operator 4
Work GOSUB operator 2
```

Example:

```
   GOSUB label_gosub_1
   GOTO  label_goto_1

label_gosub_1:

   PRINT "Work GOSUB operator 1"
   GOSUB label_gosub_2
   PRINT "Work GOSUB operator 2"
   RETURN

label_gosub_2:
   PRINT "Work GOSUB operator 3"
```

```
   PRINT "Work GOSUB operator 4"
   RETURN

label_goto_1:

   STOP
   END
```

The result of this example:

```
Work GOSUB operator 1
Work GOSUB operator 3
Work GOSUB operator 4
Work GOSUB operator 2
```

5.18.6 LET operator

This operator assigns value to a variable.

LET operator format:

```
[N] [LET] VAR=EXP
```

where:

N	an optional number of a line (label)
LET	an optional name of the operator
VAR	a variable receiving new value
EXP	an expression determining new value

Example:

```
 20 LET a=100.1

    PRINT "Work LET operator","a=";a
```

The result of this example:

```
Work LET operator a=100.1
```

5.18.7 DIM operator

DIM operator reserves a place for a numerical or string array.

DIM operator format:

```
[N] DIM LIST
```

where:

N	an optional number of a line (label)
LIST	a list of arrays names separated by commas

In Open Basic, arrays are located in free memory. Open Basic supports arrays of any size and dimension in accordance with OS restrictions. Due to the fact that the place for these arrays in Open Basic is reserved by C++ new[] operator, the restrictions on an array size or dimension are caused by new[] operator implementation features on a specific platform. For example, in MS DOS new[] operator does not request more than 64K bytes. Therefore, the overall size of an array in MS DOS Open Basic cannot exceed 64K bytes and the number of array dimensions also cannot exceed 64K bytes.

The arrays in Open Basic should be described before the first use by DIM operator.

Arrays can be multidimentional. The maximum size of an array depends on the restrictions of new[] operator implementation in the applied operating system.

When arrays are created by DIM operator, numerical arrays elements are initialized by zero, string arrays elements are initialized by empty strings.

In Open Basic an array index starts with 1.

Unlike standard Basic language, DIM operator is executable in Open Basic. It executes every time the control is passed to it. It is possible to repeat array description with the same name in DIM operation. A new array can be of other dimentions. The old data is lost.

The array repeated description can be applied to free memmory. For example, after a large array is created, DIM operator can free memory by describing an array with the same name and one element size.

When an array is described in DIM operator, not only constants but also integer variables can be used as dimentions.

The type of the array being created by DIM operator is determined (as well as a variable type) in accordance with availability of % and $ characters in the array name end.

Example:

```
DIM a%(2,2,3), b(3,2,4)
DIM a$(2,2,3,2)

a$="variable string"
a$(1,1,3,2)="array string"
a%(2,2,3)=10
b(1,1,1)=101.1


PRINT a$(1,1,3,2)
PRINT a$
PRINT "a%=";a%(2,2,3)
PRINT "b=";b(1,1,1)
```

The result of this example:

```
array string
variable string
a%=10
b=101.1
```

5.18.8 STOP and END operators

STOP and END operators are used to terminate program operation.

STOP operator calls cleaning all Open Basic tables - variables, arrays, cycles etc.

END operator also calls cleaning all Open Basic tables and also sets the internal halt flag. The internal halt flag is removed in ob_obasic::load_project method.

STOP and END operators do not call cleaning a user-defined functions table.

In a normal state, all *.bas-programs should end in STOP and END operators execution.

In version 1.94. of Open Basic, STOP operator is redundant since END operator executes all STOP operator functions. However, STOP and END operators are recommended to be used only together in *.bas-programs, since debugging functions are planned to be added in further versions of Open Basic.

STOP and END operators format:

```
[N] STOP
[N] END
```

where:

N	an optional number of a line (label)

Example:

```
    PRINT "example STOP and REM operator"

10  STOP
20  END
```

5.18.9 REM operator

REM operator sets comments into the program.

REM operator format:

```
[N] REM COMMENT
```

where:

N	an optional number of a line (label)
COMMENT	a commentary text

Example:

```
    a=1

10  REM this text is comment

    PRINT "a=";a

    STOP
    END
```

Example:

```
    a=1
    b=2
    c=3

    GOSUB label1
    GOSUB label2

    STOP
```

```
    END


label1:
        a=b+c
        RETURN

 REM this text is comment
 REM this text is comment
 REM this text is comment
 REM this text is comment
 REM this text is comment

label2:
        b=a+c
        RETURN
```

REM operators and empty lines in *.bas-program increase load time and program execution time and occupy some amount of memory or a disk. If control is not transferred to REM operator, REM does not increase program execution time (but increases program load time).

5.18.10 OPEN and CLOSE operators

OPEN operator opens file. CLOSE operator closes file.

OPEN operator format:

```
[N] OPEN filename FOR INPUT  AS FILE #EXP
[N] OPEN filename FOR OUTPUT AS FILE #EXP
```

where:

N	an optional number of a line (label)
filename	a file name
EXP	an expression that is calculated and reduced to an integer type (constants are usually used)
AS FILE	key words

OPEN FOR INPUT operator opens a file for reading.
OPEN FOR OUTPUT operator opens a file for writing.

CLOSE operator format:

```
[N] CLOSE #EXP1,[#EXP1,#EXP2,...]
```

where:

N	an optional number of a line (label)
EXP	expression that is calculated and reduced to an integer type (constants are usually used)

Example:

```
   k%=10

   OPEN "F000.TXT" FOR OUTPUT AS FILE #k%+1
   OPEN "F001.TXT" FOR OUTPUT AS FILE #k%+2

   FOR i%=1 TO KOL% STEP 1
     PRINT "i%=",i%
```

```
    PRINT #k%+1,i%
    f=i%+0.1
    PRINT #k%+2,f
  NEXT i%

  CLOSE #k%+1
  CLOSE #k%+2

  OPEN "F000.TXT" FOR INPUT AS FILE #k%+2
  OPEN "F001.TXT" FOR INPUT AS FILE #k%+3

  FOR i%=1 TO KOL% STEP 1
    INPUT #k%+2,ii%
    INPUT #k%+3,ff
    PRINT "ii%=",ii%
    f=i%+0.1
    IF ii%<>i% THEN PRINT " Error test OPEN-CLOSE command" GOTO 10
    IF ff<>f THEN PRINT " Error test OPEN-CLOSE command" GOTO 10
  NEXT i%

  CLOSE #k%+2
  CLOSE #k%+3

10 STOP
  END
```

5.18.11 KILL operator

KILL operator deletes a file.

KILL operator format:

```
[N] KILL STR1[,STR2,STR3,...]
```

where:

N	an optional number of a line (label)
STR1, STR2 ,STR3	file names

Example:

```
10 KILL "F000.TXT","F001.TXT"
```

5.18.12 READ, DATA and RESTORE operators

READ and DATA operators are used to arrange a data block that is read by Open Basic interpreter during program execution.

READ operator format:

```
[N] READ VAR1[,VAR2,VAR3,...]
```

where:

N	an optional number of a line (label)
VAR1,VAR2,VAR3	variables, which are assigned values to from DATA operator list

DATA operator format:

```
[N] DATA EXP1,[EXP1,EXP2,...]
```

where:

N	an optional number of a line (label)
EXP1,EXP2	expressions, which are calculated and assigned to a variable from READ list. The xepressions can be numerical or string. Constants are usually used.

RESTORE operator format:

```
[N] RESTORE
```

where:

N	an optional number of a line (label)

Before executing a program Open Basic browses all DATA operators in order of their appearance and creates a data block. Each time READ operator is encountered, a data block outputs sequentially matching value for variables of this operator in the same order, in which they are specified in the data block.

After READ operator is executed, the position of the last read data is remembered. The next READ operator starts to select data from the position set by the previous READ operator.

Open Basic rereads just the same data by RESTORE operator. RESTORE operator sets a pointer of DATA block reading for the initial position.

DATA operator should be the only operator in a line.

Example:

```
    DATA 1.1,2,3,4,"1 string for data"
    DATA 5.1,6,7,4+4,"2 string for data"

    DIM z%(3)

    READ a2,z%(1),z%(2),z%(3),e2$

    READ a,b%,c%,d%,e$
    PRINT "a=";a;" b%=";b%;" c%=";c%;" d%=";d%;" e$=";e$

    READ a1,b1%,c1%,d1%,e1$
    PRINT "a1=";a1;" b1%=";b1%;" c1%=";c1%;" d1%=";d1%;" e1$=";e1$

    RESTORE

    READ a1,z%(1),z%(2),z%(3),e1$
    PRINT "a1=";a1;" z%(1)=";z%(1);" z%(2)=";z%(2);" z%(3)=";z%(3);

    PRINT " e1$=";e1$
```

The result of this example:

```
a=1.1 b%=2 c%=3 d%=4 e$=1 string for data
a1=5.1 b1%=6 c1%=7 d1%=8 e1$=2 string for data
a1=1.1 z%(1)=2 z%(2)=3 z%(3)=4 e1$=1 string for data
```

5.18.13 RANDOMIZE operator

RANDOMIZE operator initializes a random number generator with new value. The current system time in seconds is used as new value. Therefore at least 1 second should pass before RANDOMIZE operator can be used again after its previous use.

RANDOMIZE operator format:

```
[N] RANDOMIZE
```

where:

N	an optional number of a line (label)

RANDOMIZE operator is placed before the first use of a random number generation function(RND function). When RND function is performed, RANDOMIZE operator changes the initial value of a random number in such way that RND function gives different numbers on its next pass.

Example:

```
10 RANDOMIZE
```

5.18.14 IF operator

This operator is used for conditional jumps performing. It has three formats: string format, block format and short block format.

5.18.14.1 String format of IF operator

A string format of IF operator:

```
[N] IF REL-EXP THEN operators
```

where:

N	an optional number of a line (label)
REL-EXP	a test condition. The relational expression can be either arithmetic or string
THEN	a key word
operators	an operator or a group of operators

If REL-EXP condition is true, operators are performed in a string after THEN key word. If REL-EXP condition is false, an operator is performed in a line following IF operator.

Operators are placed in a string format of IF operator just after THEN key word, operators should be executed in case of condition validity. If Open Basic execution system does not meet the line end character after THEN key word, this is considered to be a sign of a string format of IF operator.

A string format of IF operator is used if one or several operators to fit one line should be executed in case of condition validity.

Example:

```
    TT%=40
20  IF TT%=40 THEN GOTO 50
    PRINT "This operator do not work in example"
```

```
50  PRINT "Work IF operator";" TT%=";TT%
```

The result of this example:

```
Work IF operator TT%=40
```

Example:

```
    TT%=40

    IF TT%=40 THEN PRINT "Work IF operator","TT%=";TT% GOTO 10

    PRINT "This operator do not work in example"

10  PRINT "Work IF operator";"TT%=";TT%
```

The result of this example:

```
Work IF operator TT%=40
Work IF operator TT%=40
```

Example:

```
   TT%=40
   IF TT%=40 THEN a=101.1 b=102.1 c=103.1 GOTO 10
   a=1101.1 b=1102.1 c=1103.1
   PRINT "This operator do not work in example"
10 PRINT "a=";a,"b=";b,"c=";c
```

The result of this example:

```
a=101.1   b=102.1   c=103.1
```

5.18.14.2 Block format of IF operator

The block format of IF operator:

```
[N] IF REL-EXP THEN
    operators1
    ELSE
    operators2
    ENDIF
```

where:

N	an optional number of a line (label)
REL-EXP	a test condition. The relational expression can be either arithmetic or string.
THEN	a key word
operators1	an operator or a group of operators
operators2	an operator or a group of operators

If REL-EXP condition is true, operators in a block between THEN and ELSE key words are performed. If REL-EXP condition is false, operators in a block between ELSE and ENDIF key words are performed.

The line end character are placed in a block format of IF operator after THEN key word. If Open Basic execution system meets the line end character after THEN key word, this is considered to be a sign of a block format of IF operator.

A block format of IF operator is used if in case of condition validity several operators are required, which do fit one line or cannot be allocated in one line. For example, PRINT operator should be the only operator in a line since the line end character serve as the restriction of a print list.

Example:

```
    TT%=40

    IF TT%=40 THEN

    PRINT "TT% is 40"
    PRINT "TT% is 40"
    PRINT "TT% is 40"

    ELSE

    PRINT "TT% is not 40"
    PRINT "TT% is not 40"
    PRINT "TT% is not 40"

    ENDIF
```

The result of this example:

```
TT% is 40
TT% is 40
TT% is 40
```

5.18.14.3 Short block format of IF operator

The short block format of IF operator:

```
[N] IF REL-EXP THEN
    operators
    ENDIF
```

where:

N	an optional number of a line (label)
REL-EXP	a test condition. The relational expression can be either arithmetic or string.
THEN	a key word
operators	an operator or a group of operators

If REL-EXP condition is true, operators in a block between THEN and ENDIF key words are performed. If REL-EXP condition is false, an operator in line following ENDIF key word is performed.

IF operator short block format is used in the same cases as IF operator block format, if ELSE condition is not required.

5.18.15 CHECKLOAD operator

This operator serves to prevent the re-loading input stream by load_project method.

```
[N] CHECKLOAD “string”
```

where:

N	an optional number of a line (label)
CHECKLOAD	a key word
“string”	a unique srting in double inverted commas or single inverted commas

During ob_obasic::load_project method execution the interpreter execution system creates a srting table for CHECKLOAD operators. If in case of analysing another CHECKLOAD operator “string” string already exists in a string table for CHECKLOAD operators, load_project method is terminated with ob_obasic::CHECKLOAD_DUPLICATE termination code.

If including CHECKLOAD operator into the beginning of each file with *.bas-program allows to prevent from file reload and label conflict. Note to provide CHECKLOAD operator in *.bas-program before the first label.

On the stage of program execution by ob_obasic::run method CHECKLOAD operator is ignored.

CHECKLOAD operator argument “string” string should be a unique static string indentifying unambiguously a loaded file. CHECKLOAD operator argument cannot be a srting variable since during ob_obasic::load_project method execution no variable still exist.

In case of centralized project control the use of CHECKLOAD operator is not very actual as repeated files from a project are possible to exclude. CHECKLOAD operator is proven in case project files are loaded from a user-defined function.

5.18.16 SUB, ENDSUB, EXITSUB operators

In version 1.94, SUB, ENDSUB, EXITSUB operators are not supported, however, the corresponding key words are determined. A user should not apply SUB, ENDSUB, EXITSUB names (in upper registers) in *.bas-programs.

5.19 Built-in functions

5.19.1 SGN% function

SGN% function determines an expression sign.

```
SGN%(EXP)
```

where:

EXP	is an integer or a floating-point expression

SGN% function returns +1 if EXP>0, -1 if EXP<0 and 0 if EXP=0.
If the argument is specified incorrectly (for example, a string argument is specified), an error is generated.

SGN% function returns integer value.

Example:

```
PRINT "This is example SGN function"
PRINT "<0";SGN%(-1-2);" >0";SGN%(2*3);" =0";SGN%(9+1-10)
```

5.19.2 ABS function

ABS function determines absolute value of an argument.

```
ABS(EXP)
```

where:

EXP	is an integer or a floating-point expression

ABS function returns the floating-point result, even if the argument is of integer type. If the argument is specified incorrectly (for example, a string argument is specified), an error is generated.

ABS function calculates the result in accordance with the rules of fabs function from math.h.

Example:

```
    PRINT "This is example ABS function"
    PRINT "abs(-20.5)=";ABS(-20.5);" abs(20.5)=";ABS(20.5)
```

5.19.3 INT% function

INT% function determines an integer part of an argument.

```
INT%(EXP)
```

where:

EXP	is an floating-point expression

INT% function returns the integer result. If the argument is specified incorrectly (for example, a string argument is specified), an error is generated.

INT% function calculates the result in accordance with the rules of ceil function from math.h.

Example:

```
    PRINT "This is example INT% function"
    PRINT "int(-20.3)=";INT%(-20.3);" int(20.3)=";INT%(20.3)
```

5.19.4 SIN, COS, ATN, SQR, EXP, LOG and LOG10 functions

SIN, COS, ATN, SQR, EXP, LOG and LOG10 functions calculate sine, cosine, arc tangent, square root, exponent, logarithm and common logarithm correspondently.

```
SIN(EXP)
COS(EXP)
ATN(EXP)
SQR(EXP)
EXP(EXP)
LOG(EXP)
LOG10(EXP)
```

where:

EXP	is an integer or a floating-point expression

The functions return the floating-point result.
If the argument is specified incorrectly (for example, a string argument is specified), an error is generated.

SIN function calculates the result according to the rules of sin function from math.h.
COS function calculates the result according to the rules of cos function from math.h.
ATN function calculates the result according to the rules of atan function from math.h.
SQR function calculates the result according to the rules of sqrt function from math.h.
EXP function calculates the result according to the rules of exp function from math.h.
LOG function calculates the result according to the rules of log function from math.h.
LOG10 function calculates the result according to the rules of log10 function from math.h.

Example:

```
PRINT "This is example trigonometric function"

a=SIN(3.14/2)
b=COS(0)
c=ATN(1)
d=SQR(4.0)
e=EXP(1)
f=LOG(EXP(1))
g=LOG10(10)

PRINT "a=";a;" b=";b;" c=";c;" d=";d;" e=";e;" f=";f;" g=";g
```

5.19.5 RND function

RND function generates a pseudorandom number in 0-1 interval.

```
RND()
```

RND function returns the floating-point result. Arguments are ignored.

RND function is interconnected with RANDOMIZE operator.

RND function calculates the result according to the rules of rand function from stdlib.h.

Example:

```
PRINT "This is example RND function"

PRINT RND(),RND(),RND(),RND();

RANDOMIZE

PRINT RND(),RND(),RND(),RND();
```

5.19.6 LEN% function

LEN% function determines string length.

```
LEN%(EXP)
```

where:

EXP	is a string expression

This function returns the integer result. If the argument is specified incorrectly (for example, a integer argument or a floatimg-type argument is specified), an error is generated.

A string should be ended in zero. A final null is not included into string legth.

Example:

```
PRINT "This is example LEN function"

string_this$="aaa bbb ccc ddd"

PRINT "len=";LEN%(string_this$)
```

5.19.7 DAT$ and CLK$ functions

DAT$ and CLK$ functions return the current date and time correspondently.

```
DAT$()
CLK$()
```

The functions return the string result. Arguments are ignored.

The date format: day-month-year
The time format: hour:min:sec

DAT$ and CLK$ functions use time, localtime and localtime_s functions from time.h.

The version of the DAT $ and CLK $ functions for Visual Studio 2017 uses the localtime_s function.
The version of the DAT $ and CLK $ functions for other compilers uses the localtime function.

Example:

```
PRINT "This is example DAT$ & CLK$."
PRINT "data=";DAT$;" time=";CLK$
```

5.19.8 D2STR$, D2HEXSTR$, STR2FLOAT and STR2INT% functions

5.19.8.1 D2STR$ function

D2STR$ function converts a number into a string containing its decimal representation.

```
D2STR$(EXP)
```

where:

EXP	is an integer number or a floating point number

To convert D2STR$ function uses ostringstream stream (wostringstream for the version with Unicode).

This function returns the string result.

5.19.8.2 D2HEXSTR$ function

D2HEXSTR$ function converts an integer number into a string containing its hexadecimal representation.

```
D2HEXSTR$(EXP)
```

where:

EXP	is an integer number

To convert D2STR$ function uses ostringstream stream (wostringstream for the version with Unicode).

Before hexadecimal representation of a number, “0x” is inserted into a string.

The function returns the string result.

5.19.8.3 STR2FLOAT function

STR2FLOAT function converts a string into a number.

```
STR2FLOAT(EXP)
```

where:

EXP	is a string

To convert, STR2FLOAT function uses istringstream stream (wistringstream for the version with Unicode)

The function returns a floating-point number. If a string cannot be converted into a number, zero is returned.

5.19.8.4 STR2INT% function

STR2INT% function converts a string to an integer number.

```
STR2INT%(EXP)
```

where:

EXP	is a string

STR2INT% function returns an integer number. If a string cannot be converted to an integer number, zero is returned.

Example:

```
    PRINT "converter from digit to string =";D2STR$(123)
    PRINT "converter from string to float =";STR2FLOAT("1.234")
    PRINT "converter from string to int =";STR2INT%("1234")
    PRINT "converter from digit to hex string =";D2HEXSTR$(4660)
```

5.19.9 GET_OBASIC_VERSION% function

The GET_OBASIC_VERSION% function returns the version number of the interpreter as an integer. For version 1.94, the number 194 is returned.

```
GET_OBASIC_VERSION%()
```

5.19.10 TIME% function

The TIME% function returns the system time in seconds (the number of seconds since January 1, 1970) as an integer.

```
TIME%()
```

The function TIME% uses the time function of time.h.

5.19.11 OB_CREATE_ARRAY_OR_VAR%, OB_GET_SIZE_ARRAY%, OB_COPY_ARRAY%, OB_GET_TYPE_ARRAY_OR_VAR%, OB_ASSIGN_ARRAY_VAR%, OB_GET_FIRST_ARRAY_ITERATION%, OB_GET_NEXT_ARRAY_ITERATION% functions (general information)

Functions:

1. OB_CREATE_ARRAY_OR_VAR%
2. OB_GET_SIZE_ARRAY%
3. OB_COPY_ARRAY%
4. OB_GET_TYPE_ARRAY_OR_VAR%
5. OB_ASSIGN_ARRAY_VAR%
6. OB_GET_FIRST_ARRAY_ITERATION%
7. OB_GET_NEXT_ARRAY_ITERATION%

These functions are designed to allow users to access variables and arrays of Open Basic using strings containing the names of variables and arrays.

When using these functions, information about the names of variables or arrays can be contained in the string variables of the interpreter, and not in the text of the * .bas-program.

5.19.11.1 OB_CREATE_ARRAY_OR_VAR% function

The function OB_CREATE_ARRAY_OR_VAR% creates a variable with the specified name or an array with the specified name and size.

This function can be used to create arrays instead of the DIM operator. When using the OB_CREATE_ARRAY_OR_VAR% function, information about the name of the created variable or array can be contained in the interpreter's string variable, rather than in the * .bas program text, as in the case of the DIM operator.

The function OB_CREATE_ARRAY_OR_VAR% has four formats. In all formats, the creation of local and global variables and arrays is supported. The type of locality or globality is determined by the name of the variable or array.

1. The first format of the OB_CREATE_ARRAY_OR_VAR% function takes as an argument the variable name string and creates a variable with the specified name. The type of the variable being created is determined by the type of the name. A variable of a string type can be used as an argument.

Examples:

Creating a global variable with a floating point named varname:

```
OB_CREATE_ARRAY_OR_VAR%("varname")
```

Creating a local variable of integer type named _varname%:

```
OB_CREATE_ARRAY_OR_VAR%("_varname%")
```

Creating a global string variable with the name varname$:

```
OB_CREATE_ARRAY_OR_VAR%("varname$")
```

Creating a local variable of an integer type named _varname% using a string type variable as an argument:

```
name$="_varname%"
OB_CREATE_ARRAY_OR_VAR%(name$)
```

When creating variables, numeric variables are initialized to zero. String variables are initialized with an empty string.

2. The second format of OB_CREATE_ARRAY_OR_VAR% function takes as an argument a string of the array name and a set of integers and creates an array with the specified name and size. The type of the array being created is determined by the type of the name. A variable of a string type can be used as an argument.

Examples:

Creating a global one-dimensional array of integer type with the name arrayname% and the size of 10 elements:

```
OB_CREATE_ARRAY_OR_VAR%("arrayname%",10)
```

Create a local two-dimensional array of integer type with the name _arrayname% of size 5 by 10 elements:

```
OB_CREATE_ARRAY_OR_VAR%("_arrayname%",5,10)
```

Create a global two-dimensional array of integer type with the name arrayname% of size 3 by 4 elements using variables of the string and integer types as arguments:

```
name$="arrayname%"
size1%=3
size2%=4

OB_CREATE_ARRAY_OR_VAR%(name$,size1%,size2%)
```

When creating arrays, numeric arrays are initialized by zero. String arrays are initialized by an empty string.

3. The third format of OB_CREATE_ARRAY_OR_VAR% function takes as an argument the string of the array name and the string of the name of the whole variable. A one-dimensional array with the specified name and size is created, which is determined by the value of the variable.

Example:

Creating a global one-dimensional array of integer type with the name arrayname% of 10 elements:

```
size%=10
OB_CREATE_ARRAY_OR_VAR%("arrayname%","size%")
```

When creating arrays, numeric arrays are initialized to zero. String arrays are initialized with an empty string.

4. The fourth format of OB_CREATE_ARRAY_OR_VAR% function takes as an argument the string of the array name and the string of the name of the one-dimensional array of the integer type. A multidimensional array with the specified name and dimension is created, which is determined by an array of the integer type.

Example:

Create a local two-dimensional array of integer type with the name _arrayname% of size 5 by 10 elements:

```
DIM arraysize%(2)

arraysize%(1)=5
arraysize%(2)=10

OB_CREATE_ARRAY_OR_VAR%("_arrayname%","arraysize%")
```

When creating arrays, numeric arrays are initialized to zero. String arrays are initialized with an empty string.

On successful completion, the function OB_CREATE_ARRAY_OR_VAR% returns a value of 1. For runtime errors or errors in arguments, the function OB_CREATE_ARRAY_OR_VAR% returns 0.

5.19.11.2 OB_GET_SIZE_ARRAY% function

The function OB_GET_SIZE_ARRAY% returns the number of measurements or the size of each dimension of the array. The function OB_GET_SIZE_ARRAY% supports local and global arrays. The type of locality or globality is determined by the name of the array.

```
OB_GET_SIZE_ARRAY%("arrayname%",arg%)
```

Where:

"arrayname%"	The name of the array (in this case an integer array)
arg%	Parameter of the function of integer type. If arg% is zero or negative, the function returns the number of dimensions of the array. If the arg% parameter exceeds the number of array dimensions, the function returns zero. If the arg% parameter does not exceed the number of array dimensions, then the function returns the size of the corresponding array dimension.

Example:

```
DIM array%(20,30,40)

a1%=OB_GET_SIZE_ARRAY%("array%",0)
a2%=OB_GET_SIZE_ARRAY%("array%",1)
a3%=OB_GET_SIZE_ARRAY%("array%",2)
a4%=OB_GET_SIZE_ARRAY%("array%",3)
a5%=OB_GET_SIZE_ARRAY%("array%",4)
a6%=OB_GET_SIZE_ARRAY%("array%",- 1)
```

After executing this *.bas program, the variables will have the following values:

```
a1%=3
a2%=20
a3%=30
a4%=40
a5%=0
a6%=3
```

5.19.11.3 OB_COPY_ARRAY% function

The function OB_COPY_ARRAY% copies the source array to the destination array. The function OB_COPY_ARRAY% supports local and global arrays. The type of locality or globality is determined by the name of the array. Copied arrays must be of the same type.

```
OB_COPY_ARRAY%("arraydst%","arraysrc%")
```

Where:

"arraydst%"	The name of the destination array (in this case an array of integer type)
"arraysrc%"	The name of the source array (in this case an integer type array)

If the destination array has a different size or different dimension than the source array, then the destination array changes to become equal to the source array. If the destination array does not exist, then it is created.

Example:

```
DIM arraydst%(10)
DIM arraysrc%(20,30,40)

arraysrc%(10,20,30)=100

OB_COPY_ARRAY%("arraydst%","arraysrc%")
```

After executing this *.bas program, the arraydst% array will have the dimensions 20,30,40 and the element arraydst% (10,20,30) will be equal to 100.

5.19.11.4 OB_ASSIGN_ARRAY_VAR% function

The OB_ASSIGN_ARRAY_VAR% function initializes the destination to the source value. The destination and source of information can be a variable or an array element.

Function OB_ASSIGN_ARRAY_VAR% has three formats. In all formats, the creation of local and global variables and arrays is supported. The type of locality or globality is determined by the name of the variable or array.

1. The first format of the OB_ASSIGN_ARRAY_VAR% function takes as arguments the names of two variables and copies the value of the source variable to the destination variable. Variables or string-type constants that contain object names are used as arguments.

Example:

```
vardst%=10
varsrc%=20

OB_ASSIGN_ARRAY_VAR%("vardst%","varsrc%")
```

After executing this *.bas program, the variable vardst% will be 20.

2. The second format of the OB_ASSIGN_ARRAY_VAR% function takes as argument the variable and array names and the array element indexes in the form of constants or integer variables.

Example:

```
DIM arraydst%(20,30,40)

varsrc%=200

OB_ASSIGN_ARRAY_VAR%("arraydst%","varsrc%",10,20,30)
```

After executing this *.bas program, the element arraysrc% (10,20,30) will be equal to 200.

Example:

```
DIM arraysrc%(20,30,40)

arraysrc%(10,20,30)=100

vardst%=200

OB_ASSIGN_ARRAY_VAR%("vardst%","arraysrc%",10,20,30)
```

After executing this * .bas program, the variable vardst% will be 100.

Who is the source and who is the destination is determined by the order of the names in the function call line. The first name in the function call is treated as the name of the information destination. The second name in the function call is treated as the name of the information source.

3. The third format of OB_ASSIGN_ARRAY_VAR% function is similar to the second format, but array indexes are specified in a single one-dimensional array of integer type.

Example:

```
DIM arraydst%(20,30,40)

DIM arrayindex%(3)

arrayindex%(1)=10
arrayindex%(2)=20
arrayindex%(3)=30

varsrc%=200

OB_ASSIGN_ARRAY_VAR%("arraydst%","varsrc%","arrayindex%")
```

After executing this *.bas program, the element arraysrc% (10,20,30) will be equal to 200.

Example:

```
DIM arraysrc%(20,30,40)

DIM arrayindex%(3)

arrayindex%(1)=10
arrayindex%(2)=20
arrayindex%(3)=30

arraysrc%(10,20,30)=100

vardst%=200
```

```
OB_ASSIGN_ARRAY_VAR%("vardst%","arraysrc%","arrayindex%")
```

After executing this *.bas program, the variable vardst% will be 100.

As in the second format of the OB_ASSIGN_ARRAY_VAR% function, in its third format, who is the source and who is the destination is determined by the order of names in the function call line. The first name in the function call is treated as the name of the information destination. The second name in the function call is treated as the name of the information source.

In all cases, the function returns the source type, casted to the integer type according to the table:

floating-point source	The function returns 0
source of integer type	The function returns 1
source of string type	The function returns 2
source name does not found in the tables of variables and arrays	The function returns 3

If the function arguments are specified with an error (for example, the function is called without arguments), then the function returns (-1).

5.19.11.5 OB_GET_TYPE_ARRAY_OR_VAR% function

The OB_GET_TYPE_ARRAY_OR_VAR% function returns a variable or array type code.
Function OB_GET_TYPE_ARRAY_OR_VAR% supports local and global arrays. The type of locality or globality is determined by the name of the array.

```
OB_GET_TYPE_ARRAY_OR_VAR%("name")
```

"name"	Variable or array name

Rules of the function:

The name of a floating-point variable or array is specified	The function returns 0
The name of an integer variable or array of is specified	The function returns 1
The name of a string variable or array is specified	The function returns 2
Source name does not found in the tables of variables and arrays	The function returns 3

If the argument is not specified or is specified with an error (for example, instead of a name given a number), the function returns (-1).

Example:

```
var%=10

code_type%=OB_GET_TYPE_ARRAY_OR_VAR%("var%")
```

After executing this *.bas program, the variable code_type% will be 1.

5.19.11.6 OB_GET_FIRST_ARRAY_ITERATION% u OB_GET_NEXT_ARRAY_ITERATION% functions

Functions OB_GET_FIRST_ARRAY_ITERATION% and OB_GET_NEXT_ARRAY_ITERATION% are used to generate an array of indexes for the specified array. An array of indexes can be used in the functions OB_CREATE_ARRAY_OR_VAR% (the fourth format of the function OB_CREATE_ARRAY_OR_VAR%) and OB_ASSIGN_ARRAY_VAR% (the third format of the function OB_ASSIGN_ARRAY_VAR%).

The functions OB_GET_FIRST_ARRAY_ITERATION% and OB_GET_NEXT_ARRAY_ITERATION% support local and global arrays. The type of locality or globality is determined by the name of the array.

Functions OB_GET_FIRST_ARRAY_ITERATION% and OB_GET_NEXT_ARRAY_ITERATION% always work in pairs. First, the OB_GET_FIRST_ARRAY_ITERATION% function creates an index array. Next, using OB_GET_NEXT_ARRAY_ITERATION%, the next set of indexes is generated, until all indexes are used. When all indexes of the array are exhausted, the function OB_GET_NEXT_ARRAY_ITERATION% returns a special return code equal to 2.

Example:

```
DIM array%(20,30,40)

DIM arrayindex%(3,5)

ret%=OB_GET_FIRST_ARRAY_ITERATION%("array%","arrayindex%")
```

After executing this *.bas program, the arrayindex% array becomes a one-dimensional array with a size of 3 elements and the values of its elements will be equal (1,1,1).

The function OB_GET_FIRST_ARRAY_ITERATION% returns 0 in case of an error in specifying arguments.
Function OB_GET_FIRST_ARRAY_ITERATION% returns 1 if the array of indexes is successfully generated.

Example:

```
DIM array%(20,30,40)

ret%=OB_GET_NEXT_ARRAY_ITERATION%("array%","arrayindex%")
```

After executing this *.bas-program, the values of arrayindex% array elements will be equal (1,1,2).

The function OB_GET_NEXT_ARRAY_ITERATION% returns 0 in case of an error in specifying arguments.
Function OB_GET_NEXT_ARRAY_ITERATION% returns 1 if the array of indexes is successfully generated.
The function OB_GET_NEXT_ARRAY_ITERATION% returns 2 if all indexes of the array are exhausted.

Example:

```
DIM array% (2,2,3)

array% (1,1,1) = 100
array% (1,1,2) = 200
array% (1,2,3) = 300

DIM arrayindex% (3)

var% = 100

ret% = OB_GET_FIRST_ARRAY_ITERATION% ("array%", "arrayindex%")

PRINT

_local_label1:

OB_ASSIGN_ARRAY_VAR% ("var%", "array%", "arrayindex%")

PRINT "array% ("; arrayindex% (1); ","; arrayindex% (2); "," arrayindex% (3); ") ="; var%

ret% = OB_GET_NEXT_ARRAY_ITERATION% ("array%", "arrayindex%")

IF ret% = 2 THEN GOTO _local_label2

GOTO _local_label1

_local_label2:

STOP
```

```
END
```

After executing this * .bas program, the array% array is printed. The result will look like this:

```
array% (1,1,1) = 100
array% (1,1,2) = 200
array% (1,1,3) = 0
array% (1,2,1) = 0
array% (1,2,2) = 0
array% (1,2,3) = 300
array% (2,1,1) = 0
array% (2,1,2) = 0
array% (2,1,3) = 0
array% (2,2,1) = 0
array% (2,2,2) = 0
array% (2,2,3) = 0
```

5.20 Error handling. ob_err class and ob_err class methods

When Open Basic encounters an error, it generates an exception of ob_err type with an error code. A user should catch this exception and handle it. To handle exceptions, ob_err class has ob_err::getcode and ob_err::release methods.

The method:

```
ob_type_codeerror ob_err::getcode()
```

does not have arguments and returns the error code (an integer number).

The method:

```
void ob_err::release(ob_type_ostreamcommon* os,const ob_type_small_char* currentmessage[]);
```

prints a text message about a mistake from currentmessage table into os stream.

The parameters:

os	a pointer to an output stream
currentmessage[]	a table of diagnostic messages

Open Basic presents ob_englmessage table of diagnostic messages in English. A user can create his/her own table of diagnostic messages in another language and use it as release method second argument. ob_englmessage table of diagnostic messages is located in ob5.cpp file.

A typical scheme of ob_err exeptions handling:

```
int main(int argc, char * argv[]){

int ret=0;//main return code

ifstream fi;//input stream

try{

ob_obasic basic_interpreter;//interpreter creating
fi.open("test1.bas",ios::binary);//input stream in a binary mode opening

basic_interpreter.clear_project();

basic_interpreter.load_project(&fi,0);//*.bas-program loading
```

```
ob_obasic::typeend te=basic_interpreter.run();//*.bas-program running

cout<<endl<<"Stop by stop code";//halt code printing
cout<<endl<<"Stop code=";

switch(te){
case ob_obasic::ENDOPERATORDETECT  :cout<<"ENDOPERATORDETECT";break;
case ob_obasic::ENDFILEEND         :cout<<"ENDFILEEND";break;
case ob_obasic::BREAKPOINTEND      :cout<<"BREAKPOINTEND";break;
case ob_obasic::NOLOADEND          :cout<<"NOLOADEND";break;
case ob_obasic::EOLEND             :cout<<"EOLEND";break;
default:cout<<"Unknown stop code";break;
}//switch

}//try

catch(ob_err& e){

cout<<endl<<"Stop by OB exception catch";//exception handling
ret=e.getcode();                        //error code receiving
e.release(&cout,ob_englmessage);        //text message printing in cout

}//OB catch

return ret;
}
```

More detailed information on exception handling can be found in .\example\ob.cpp file of a console example.