

Интерпретатор Open Basic.

Описание применения.

Версия 1.91.

Copyright (C) 1998–2013 МКТМК Moscow, Russia

<http://sourceforge.net/projects/obasic/>
<http://mktmk.narod.ru>

e-mail: mktmk<at>narod.ru
e-mail: openbasics<at>gmail.com

МКТМК software company. Moscow, Russia

Оглавление

Аннотация.....	2
Производитель.....	2
Обратная связь.....	2
Гарантии.....	2
Ссылки.....	3
Список версий и исправленных ошибок.....	3
Перечень файлов поставки.....	3
1. Описание задачи и назначение программы.....	3
2. Замечания о быстрействии интерпретатора.....	3
3. Условия применения.....	4
4. Входные и выходные данные.....	4
4.1. Общие сведения об интерпретаторе Open Basic. Как использовать интерпретатор. Как загрузить и запустить *bas-программу.....	4
4.2. Обзор технических решений.....	4
4.3. Обзор макросов условной трансляции.....	5
4.4. Обзор системы ввода-вывода.....	6
4.4.1. Первый способ организации ввода-вывода (с использованием классов ввода-вывода).....	6
4.4.2. Второй способ организации ввода-вывода (с использованием пользовательских функций ввода-вывода).....	8
4.5. Методы и перечисления класса ob_obasic, которые используются для загрузки и запуска *bas-программ.....	8
4.5.1. Конструктор ob_obasic::ob_obasic.....	8
4.5.2. Метод ob_obasic::clear_project.....	8
4.5.3. Метод ob_obasic::load_project и перечисление ob_obasic::loadresult.....	8
4.5.4. Метод ob_obasic::run и перечисление ob_obasic::typeend.....	9
4.5.5. Метод ob_obasic::set_current_input_stream.....	9
4.5.6. Метод ob_obasic::get_current_input_stream.....	9
4.6. Методы и перечисления класса ob_obasic, которые используются для организации отладки в интерпретаторе Open Basic.....	9
4.6.1. Перечисление ob_obasic::step.....	9
4.6.2. Метод ob_obasic::setstep.....	9
4.6.3. Метод ob_obasic::getstep.....	9
4.6.4. Функция ob_loadbreakstr.....	9
4.7. Методы и перечисления класса ob_obasic, которые используются для сопряжения ввода и вывода интерпретатора Open Basic с вводом и выводом операционной системы.....	10
4.7.1. Общие сведения.....	10
4.7.2. Методы ob_obasic::setoperatorbreak и ob_obasic::getoperatorbreak.....	10
4.7.3. Методы ob_obasic::setin и ob_obasic::setout.....	10
4.7.4. Перечисление ob_lex::typelex.....	11
4.7.5. Метод ob_obasic::gettypenextlex.....	11
4.8. Синтаксис языка Open Basic. Общие сведения.....	11
4.9. Типы данных и имена Open Basic.....	11
4.10. Метки Open Basic.....	12
4.10.1. Метки в виде номеров строк.....	12
4.10.2. Строковые метки с финальным двоеточием.....	13
4.11. Пользовательские функции в Open Basic.....	13
4.11.1. Прототипы для пользовательской функции.....	14
4.11.2. Параметры пользовательских функций.....	14
4.11.3. Более подробное описание параметров пользовательских функций.....	14
4.11.3.1. Структура параметра parstring.....	15
4.11.3.2. Структура массива descrf (массивы descrt и descrc имеют аналогичную структуру).....	15
4.11.3.3. Структура массива parf (массивы parf и parc имеют аналогичную структуру).....	15
4.11.3.4. Использование параметров пользовательских функций.....	16
4.11.4. Выбор Basic-имени пользовательской функции.....	16
4.12. Подключение пользовательской функции к исполняющей системе Open Basic (общие сведения).....	16
4.13. Методы и перечисления класса ob_obasic для подключения и отключения пользовательских функций.....	17
4.13.1. Перечисление ob_type_del.....	17
4.13.2. Методы ob_obasic::setfun.....	18
4.13.3. Метод ob_obasic::checkfun.....	18
4.13.4. Метод ob_obasic::delfun.....	18
4.14. Доступ из пользовательской функции к данным *bas-программы. Общие сведения.....	18
4.15. Методы и перечисления класса ob_obasic для доступа к данным Open Basic из пользовательских функций.....	19
4.15.1. Методы и перечисления класса ob_obasic для определения типа переменных и массивов Open Basic.....	19
4.15.1.1. Перечисление ob_type_ident.....	19
4.15.1.2. Метод ob_obasic::typevar.....	19
4.15.1.3. Метод ob_obasic::typearray.....	19
4.15.2. Методы класса ob_obasic для определения длины строковых переменных и элементов строковых массивов Open Basic.....	20
4.15.3. Методы класса ob_obasic для определения размеров массивов OpenBasic.....	20
4.15.4. Методы класса ob_obasic для записи переменных Open Basic.....	21
4.15.5. Методы класса ob_obasic для чтения переменных Open Basic.....	21
4.15.6. Методы класса ob_obasic для создания переменных и массивов Open Basic.....	22
4.15.6.1. Методы класса ob_obasic для создания переменных Open Basic.....	22
4.15.6.2. Методы класса ob_obasic для создания массивов OpenBasic.....	22
4.16. Описание других методов класса ob_obasic.....	22
4.17. Операторы интерпретатора Open Basic.....	22
4.17.1. Оператор PRINT.....	23
4.17.2. Оператор INPUT.....	23
4.17.3. Операторы FOR и NEXT.....	23
4.17.4. Оператор GOTO.....	24
4.17.5. Операторы GOSUB и RETURN.....	24
4.17.6. Оператор LET.....	25

4.17.7.	Оператор DIM	25
4.17.8.	Операторы STOP и END	26
4.17.9.	Оператор REM	26
4.17.10.	Операторы OPEN и CLOSE	27
4.17.11.	Оператор KILL	27
4.17.12.	Операторы READ, DATA и RESTORE	27
4.17.13.	Оператор RANDOMIZE	28
4.17.14.	Оператор IF	28
4.17.14.1.	Строковый формат оператора IF	28
4.17.14.2.	Блочный формат оператора IF	29
4.17.14.3.	Блочный укороченный формат оператора IF	29
4.17.15.	Оператор CHECKLOAD	29
4.17.16.	Операторы SUB, ENDSUB, EXITSUB	30
4.18.	Встроенные функции	30
4.18.1.	Функция SGN%	30
4.18.2.	Функция ABS	30
4.18.3.	Функция INT%	30
4.18.4.	Функции SIN, COS, ATN, SQR, EXP, LOG и LOG10	30
4.18.5.	Функция RND	31
4.18.6.	Функция LEN%	31
4.18.7.	Функции DATS и CLKS	31
4.18.8.	Функции D2STRS, D2HEXSTRS, STR2FLOAT и STR2INT%	31
4.18.8.1.	Функция D2STRS	31
4.18.8.2.	Функция D2HEXSTRS	31
4.18.8.3.	Функция STR2FLOAT	31
4.18.8.4.	Функция STR2INT%	32
4.19.	Обработка ошибок. Класс ob_err и методы класса ob_err	32

Аннотация.

Open Basic (OB) представляет собой реализацию интерпретатора языка **Basic**.

Open Basic разработан для встраивания в приложения пользователя в качестве скриптового языка.

Open Basic поставляется в исходных текстах под лицензией **MIT**.

Open Basic имеет возможность расширения системы команд путем подключения пользовательских функций к исполняющей системе **Open Basic**.

Пользовательские функции могут быть написаны на **C/C++**, ассемблере или других языках.

Пользовательские функции могут вызываться из **Basic-программы**, получать параметры разных типов из **Basic-программы** и возвращать результаты своей работы в **Basic-программу**.

Специально разработанный интерфейс вызова пользовательских функций позволяет на этапе исполнения определить тип и порядок следования параметров в вызове.

Open Basic реализует подмножество команд языка **Basic**. **Open Basic** написан полностью на **C++** и реализован в виде класса с именем **ob_obasic**.

Open Basic поддерживает данные следующих типов:

1. Числа с плавающей точкой.
2. Числа с фиксированной точкой со знаком (целые).
3. Строки.
4. Массивы чисел с плавающей точкой.
5. Массивы чисел с фиксированной точкой.
6. Массивы строк.

Open Basic имеет multithread-safe код.

В документе описывается:

1. Порядок работы с интерпретатором **Open Basic**.
2. Синтаксис операторов **Open Basic**.
3. Интерфейс к данным **Open Basic** из пользовательских функций.
4. Правила подключения пользовательских функций.
5. Типы данных **Open Basic**.
6. Возможности отладки.

Open Basic не использует никаких графических библиотек.

Open Basic не использует никаких системных вызовов ОС.

Производитель.

Copyright (C) 1998–2013 МКТМК Moscow, Russia

<http://sourceforge.net/projects/obasic/>

<http://mktmk.narod.ru>

e-mail: mktmk<at>narod.ru

e-mail: openbasicsoft<at>gmail.com

МКТМК software company. Moscow, Russia

Обратная связь.

Если Вы собираетесь применять интерпретатор, и хотите знать о выходе новых версий и о возможных изменениях почтового и WWW адреса, то сообщите об этом по **e-mail**.

МКТМК software company заинтересована в опыте применения интерпретатора:

1. Для чего Вы применяете интерпретатор.
2. Ваш транслятор и номер его версии.
3. Найденные в программе ошибки.
4. Полнота документации и ошибки в документации.
5. Подключали ли Вы пользовательские функции.
6. Замечания к интерфейсу пользовательских функций.
7. С какой ОС Вы работаете.
8. Откуда Вы узнали об интерпретаторе.

Эти сведения помогут нам при сопровождении программы.

Гарантии.

Интерпретатор и документация поставляются на условиях "как есть".

Фирма **МКТМК** не несет ответственности за возможные убытки, понесенные из-за использования интерпретатора.

Мы будем признательны за все найденные в программе ошибки.

Ссылки.

Все упомянутые в тексте торговые марки являются собственностью их владельцев.

Список версий и исправленных ошибок.

16.feb.2013 – версия 1.91 для **Windows**

1. Добавлена поддержка локальных (в файле) меток, переменных и массивов. Локальные метки, переменные и массивы имеют имя, начинающееся с подчеркивания “_”. Поддержка локальных (в файле) меток, переменных и массивов уменьшает конфликт имен при разработке многофайловых проектов.
2. Добавлена возможность продолжить оператор Basic на следующую строку с помощью обратного слеша “\”.
3. Для доступа к локальным переменным и массивам методы доступа к переменным получили новый параметр – указатель на поток (файл), в котором находится переменная. Если задано локальное имя переменной и указатель на поток, то поиск производится в таблице локальных переменных указанного потока.
4. Модифицированы методы **readvar**, **writevar**. Эти методы осуществляют доступ к элементам массива из пользовательских функций. Теперь не поддерживается доступ к элементам массива через задание индексов в имени переменной. Доступ к элементам массива теперь возможен только через явное задание текущих индексов элемента массива в параметре «**as**» методов **readvar**, **writevar**.
5. Базовый класс ввода-вывода переименован. Старое имя базового класса ввода-вывода было **ob_functor_break_base**. Новое имя базового класса ввода-вывода стало **ob_base_class_for_IO_service**.
6. Ввод-вывод через базовый класс **ob_base_class_for_IO_service** стал единственным способом реализации ввода-вывода. Старый механизм ввода-вывода удален.
7. Убран метод **ob_bas::restoreinputpointer()**; При запуске методов **ob_base_class_for_IO_service::run_before()** и **ob_base_class_for_IO_service::run_after()** следующая лексема доступна через метод **ob_bas::gettypenextlex()**;
8. Исправлена ошибка в реализации оператора **REM**. Комментарий **REM**. Теперь при ошибочных лексемах в комментариях не происходит генерация ошибки.
9. Метод **load_project** теперь возвращает результат своей работы.
10. Добавлен оператор **CHECKLOAD**. Оператор **CHECKLOAD** позволяет предотвратить повторную загрузку потока.
11. Добавлен заголовочный файл **obmain.h** и ключи условной трансляции.
12. Исправлена ошибка в реализации оператора **DATA**.
13. Версия 1.91 тестировалась под следующими трансляторами: **MSVC2010**, **Borland C++ Builder 6.0**, **vxDev-C++ (GCC)**.

03.jan.2010 – версия 1.90 для **Windows**

1. Исправлена ошибка в обработке оператора **REM** при загрузке файла. Ранее при комментировании строковой метки с двоеточием возникало зависание при выполнении метода **load_project**.
2. Добавлена возможность создавать проекты из нескольких файлов. Для этого в интерпретатор добавлены новые методы **clear_project** и **load_project** с помощью которых можно загружать несколько *.bas-файлов. Между файлами проекта возможны переходы операторами **GOTO** и вызовы операторами **GOSUB**. Все метки и переменные проекта доступны из любого файла (модуля).
3. Добавлена возможность переходов **GOTO** и **GOSUB** на метку, расположенную в другом файле.
4. Введено ограничение на количество вложенных вызовов **GOSUB**. По умолчанию количество вложенных вызовов не может превышать 4096. Возможно изменение этого параметра с помощью методов **set_max_nested_gosub** и **get_max_nested_gosub**.
5. Добавлен новый механизм запуска пользовательского кода до и после указанного оператора. Старый механизм запуска пользовательского кода базировался на прерывании выполнения метода **run**. Новый механизм запуска пользовательского кода базируется на классах ввода-вывода и позволяет уменьшить количество кода, необходимого для организации ввода-вывода. Старый механизм запуска пользовательского кода признан устаревшим и оставлен для совместимости с существующим кодом. В последующих версиях старый механизм запуска пользовательского кода будет удален и поэтому он не рекомендуется к применению в новых разработках. Подробнее см. описание методов **setoperatorbreak** и **getoperatorbreak**.

18.feb.2007 – версия 1.80 для **Windows** и **Linux**

1. Исправлена ошибка в индексации массива элементом массива.
2. Теперь имя файла в операторе **OPEN** может задаваться не только строковой константой, но и строковой переменной.
3. Начиная с версии 1.80 в *.bas-программе кроме меток в виде чисел (номеров строк) поддерживаются строковые метки с двоеточием в конце.
4. Из класса **ob_obasic** исключены методы **getstarttime()** и **getstoptime()** как не относящиеся к основному назначению интерпретатора. Пользователь легко может самостоятельно проконтролировать время выполнения любого метода интерпретатора.

28.aug.2005 – версия 1.71 для **Windows** и **Linux**

Исправлена ошибка в функции возвращения номера версии.

21.aug.2005 – версия 1.70 для **Windows** и **Linux**

Исправлена ошибка при работе с шестнадцатеричными числами.
Созданы multithread библиотеки для **MSVC7**

23.feb.2005 – версия 1.50 для **Windows** и **Linux**

1998 – первая версия для **DOS**.

Перечень файлов поставки.

```
.\bas          - файлы примеров, которые могут выполняться в консольном приложении ob191_msvc2010.exe
.\bat          - файлы примеров, которые запускают консольное приложение ob191_msvc2010.exe
.\source       - файлы исходных текстов
.\comline     - функции для разбора командной строки
.\example     - main() тестового примера ob191_msvc2010.exe, пользовательские функции и примеры их подключения.
.\exe         - консольное приложение ob191_msvc2010.exe транслированное на MSVC 2010.
.\documentation - файлы описания применения на русском и английском
.\include     - заголовочные файлы
```

1. Описание задачи и назначение программы.

Часто в пользовательских приложениях возникает необходимость в применении простого скриптового языка.

Если приложение оперирует некоторым количеством примитивов, которые должны вызываться в разной последовательности для разных режимов работы, то можно оформить вызов каждого примитива в виде отдельной функции скриптового языка. Такой подход позволяет гибко изменять алгоритм работы приложения без перетрансляции кода.

Например:

1. Запросы к базе данных с разными условиями выборки данных.
2. Тестирование оборудования и/или программного обеспечения.
3. Организация связи с нестандартным оборудованием по различным каналам.
4. Построение графических интерфейсов и манипуляции с объектами **GUI**.
5. Входной язык в программах-терминалах для разбора командной строки.
6. Входной язык оператора-технолога в задачах АСУТП.
7. Входной язык сложных конфигурационных файлов.
8. И многое другое.

Графический интерфейс пользователя затрудняет возможность автоматизации задач. Поэтому часто необходимо дополнить графический интерфейс возможностями, которые предоставляет скриптовый язык.

Интерпретатор **Open Basic** разработан для встраивания в приложения пользователя в качестве скриптового языка.

Open Basic имеет возможность расширения системы команд путем подключения пользовательских функций к исполняющей системе **Open Basic**.

Пользовательские функции могут быть написаны на **C/C++**, ассемблере или других языках.

Пользовательские функции могут вызываться из **Basic-программы**, получать параметры разных типов из **Basic-программы** и возвращать результаты своей работы в **Basic-программу**.

Также пользовательские функции имеют доступ ко всем переменным и массивам **Basic-программы**, а не только передаваемым через параметры. Доступ к переменным и массивам **Basic-программы** осуществляется по именам этих переменных и массивов.

2. Замечания о быстродействии интерпретатора.

В среднем интерпретатор выполняет код в 100 раз медленнее, чем эквивалентный скомпилированный код.

Вместе с тем интерпретатор имеет то преимущества, что интерпретатор не нуждается в компиляторе и позволяет гораздо проще проводить отладку.

Если приложение снабжено интерпретатором, то в алгоритм можно вносить изменение без перекомпиляции приложения.

В целом можно сказать, что линейные алгоритмы с условными переходами и циклы без вложенности выполняются в интерпретаторе довольно быстро. При применении вложенных циклов на больших объемах данных может потребоваться вынести внутренний цикл в пользовательскую функцию (из-за потери скорости). Применение же вложенных циклов с уровнем вложенности три и более имеет смысл только на очень небольших объемах данных (порядка десятков).

При выявлении критического по быстродействию участка алгоритма, необходимо вынести этот участок алгоритма в пользовательскую функцию с соответствующими параметрами. Необходимо правильно распределить задачу между пользовательскими функциями и интерпретатором. На уровне интерпретатора имеет смысл оставлять управление достаточно высокоуровневыми частями алгоритма.

Так как **Open Basic** это интерпретатор, то длина имен меток, переменных, массивов и других внутренних объектов интерпретатора влияет на его быстродействие. В общем случае, тем длиннее имена объектов, тем больше времени затрачивается на поиск этих объектов в таблицах интерпретатора.

3. Условия применения.

Интерпретатор **Open Basic** написан полностью на C++ и не использует системные вызовы какой-либо ОС. Интерпретатор **Open Basic** может использоваться совместно с любой графической библиотекой.

4. Входные и выходные данные.

4.1. Общие сведения об интерпретаторе **Open Basic**. Как использовать интерпретатор. Как загрузить и запустить *.bas-программу.

Интерпретатор **Open Basic** реализован в виде класса с именем **ob_obasic**.

Работа интерпретатора состоит в выполнении программы на языке **Basic**.

В данном описании эта программа называется ***.bas-программой**. Программа ***.bas** является текстовым файлом или буфером в памяти, открытым как поток **istream**.

Далее в описании:

1. "оператор" - означает оператор языка **Basic**. Например, **PRINT**, **FOR** и т.д.
2. "пользовательская функция" - означает пользовательскую функцию, присоединенную к исполняющей системе **Open Basic**
3. "функция" - означает функцию **C++**
4. "метод" - означает метод класса **ob_obasic** или другого класса, входящего в состав интерпретатора.
5. **"*.bas-программа"** - означает программа на языке **Basic**

Чтобы использовать интерпретатор нужно:

1. Скачать со страницы <http://sourceforge.net/projects/obasic/> архив, содержащий интерпретатор.
2. Включить в проект исходные тексты интерпретатора (папка **.\source**).
3. В проекте указать путь для подключения заголовочных файлов интерпретатора **ob.h**, **obmain.h** (папка **.\include**).
4. В проекте указать путь для подключения заголовочных файлов шаблонных контейнеров **mstore.h**, **mvect.h**, **mlist.h**, **mstack.h**, **mhash.h** (папка **.\include**).
5. Включить в свои **C++** файлы директивой **#include** заголовочный файл интерпретатора **ob.h**.
6. Сделать объявление об использовании пространства имен интерпретатора **using namespace ob_charspace;**
7. Создать экземпляр класса **ob_obasic** в своей **C++** программе.
8. В случае использования в ***.bas-программе** операторов **PRINT** и **INPUT** для ввода-вывода, создать пользовательские классы ввода-вывода, производные от класса **ob_base_class_for_IO_service**, и подключить их к интерпретатору.
9. Очистить проект с помощью метода **clear_project**.
10. Загрузить программу для выполнения с помощью метода **load_project**.
11. Запустить программу на выполнение с помощью метода **run**.

Пример:

```
using namespace ob_charspace;

// загрузка и запуск программы test1.bas
ifstream inp1("test1.bas", ios::binary); //открытие входного потока

ob_obasic basic_interpreter; //создание экземпляра интерпретатора

basic_interpreter.clear_project(); //очистка проекта
basic_interpreter.load_project(&inp1, 0); //загрузка test1.bas спомощью метода load_project

basic_interpreter.run(); //запуск программы test1.bas
```

ВНИМАНИЕ! Входной поток должен быть открыт в **ios::binary** режиме.

ПОЯСНЕНИЕ:

Несмотря на то, что ***.bas-программа** является текстовым файлом, открываться как поток она должна в **ios::binary** режиме. Это связано с особенностями обработки символов возврата каретки и перевода строки в **ios::text** режиме.

Пример:

```
using namespace ob_charspace;

ob_obasic basic_interpreter; //создание экземпляра интерпретатора

//после окончания программы test1.bas загрузка и запуск программы test2.bas и test3.bas
ifstream inp2("test2.bas", ios::binary); //открытие входного потока
ifstream inp3("test3.bas", ios::binary); //открытие входного потока

basic_interpreter.clear_project(); //очистка проекта
basic_interpreter.load_project(&inp2, 0); //загрузка *.bas-программы
basic_interpreter.load_project(&inp3, 0); //загрузка *.bas-программы
basic_interpreter.set_current_input_stream(&inp2); //установка потока inp2 первым в очереди запуска

basic_interpreter.run(); //запуск программы test2.bas
```

ВНИМАНИЕ! Входные потоки должны быть открыты в **ios::binary** режиме.

ВНИМАНИЕ! Если проект состоит из нескольких файлов (потоков), то метод **run** запустит последний загруженный файл (поток). Для явного назначения файла (потока), который будет запущен первым, нужно использовать метод **set_current_input_stream**.

4.2. Обзор технических решений.

При проектировании интерпретатора решались следующие основные проблемы:

1. Возможность хранения **Basic-программы** в файле на диске или в буфере памяти. Для обеспечения этой возможности интерпретатор принимает входную программу в виде указателя на поток ввода **istream***. Конкретный вид потока (файл или буфер) определяется пользователем. Аналогично входной поток оператора **INPUT** и выходной поток оператора **PRINT** тоже могут быть буфером или файлом.
2. Один экземпляр интерпретатора может выполнять один проект в данный момент времени. В программе может быть несколько экземпляров интерпретатора. Интерпретатор не имеет статических членов данных (кроме нескольких доступных только по чтению констант). Поэтому несколько экземпляров интерпретатора могут работать одновременно в разных нитях программы.
3. Пользовательские функции в **Open Basic** имеют унифицированный интерфейс. Этот интерфейс позволяет проанализировать типы и порядок следования параметров в вызове пользовательской функции на этапе выполнения программы. Такой подход требует от программиста аккуратности, но зато не ограничивает возможности создания пользовательских функций с разными параметрами.

Для обеспечения гибкости функционирования применяется разделение выполнения программы на три этапа:

1. Первый этап это очистка проекта методом **clear project**. В процессе очистки проекта происходит очистка внутренних таблиц интерпретатора от данных предыдущего проекта (очистка таблицы меток переходов, таблиц переменных и массивов и т.д.).
2. Второй этап это загрузка программы методом **load project**. В процессе загрузки происходит создание таблицы меток переходов и подпрограмм и настройка внутренних переменных. Набор одновременно загруженных в экземпляр интерпретатора файлов (потоков) называется проектом. Для проекта применяется концепция общего пространства имен глобальных переменных, имен массивов и меток. Глобальные метки во всех файлах (потоках) проекта должны быть не повторяющимися. Глобальные переменные, созданные в одном файле проекта, доступны во всех файлах проекта. Имена глобальных меток, переменных, и массивов начинаются с любого буквенного символа, кроме символа подчеркивания "_". Цифровые метки всегда глобальные. Начиная с версии 1.91, существует возможность создать локальные (в файле) метки, переменные, и массивы. Имена локальных меток, переменных, и массивов начинаются с символа подчеркивания "_". Локальные (в файле) метки, переменные, и массивы могут повторяться в разных файлах. В каждом файле локальные переменные, и массивы имеют свое значение, а локальные метки позволяют сделать переход только внутри файла, в котором они определены.
3. Третий этап это выполнение программы методом **run**. В процессе выполнения производится чтение текста программы и выполнение действий (создание переменных, массивов, вычисление и присвоение значений переменным, выполнение операторов и функций и т.д.)

Завершение метода `run` происходит по нескольким причинам:

1. В процессе работы произошло исчерпание входного потока.
2. В процессе работы найдена синтаксическая ошибка в программе
3. В процессе работы найден символ точки останова.
4. В процессе работы найден символ конца строки при установленном флаге пошагового режима работы.

Метод `run` возвращает код завершения. Анализируя код завершения метода `run`, пользователь может узнать причину окончания метода `run`. Если метод `run` завершился без ошибок, то он может быть продолжен без повторной загрузки программы. Например, если метод `run` завершился из-за обнаружения символа точки останова или из-за обнаружения символа конца строки при установленном флаге пошагового режима работы, то метод `run` может быть продолжен без повторной загрузки программы.

Если метод `run` завершился из-за синтаксической или другой ошибки, то перед последующим запуском метода `run` проект должен быть очищен методом **clear project** и перезагружен методом **load project**.

4.3. Обзор макросов условной трансляции.

1. Макрос **OB_USER_WCHAR**.

Макрос **OB_USER_WCHAR** зарезервирован для будущей поддержки кодировки **Unicode**.

Если при компиляции определен макрос **OB_USER_WCHAR**, то **Open Basic** компилируется с поддержкой кодировки **Unicode**.

В версии 1.91 макрос **OB_USER_WCHAR** не поддерживается, но в коде есть ветки, которые будут впоследствии активны при компиляции программы с поддержкой кодировки **Unicode**.

2. Макрос **OB_USER_VERSION_CHARS**.

С помощью макроса **OB_USER_VERSION_CHARS** можно управлять именем пространства имен, в котором находится код интерпретатора.

Код интерпретатора находится в пространстве имен **namespace**.

Разные имена пространств имен сделаны для того, чтобы избежать конфликта имен при создании в одной программе нескольких объектов класса **ob_obasic** разных версий.

В таблице приведены пространства имен интерпретатора в зависимости от макросов **OB_USER_WCHAR** и **OB_USER_VERSION_CHARS** для версии интерпретатора 1.91.

	Макрос OB_USER_WCHAR не определен	Макрос OB_USER_WCHAR определен
Макрос OB_USER_VERSION_CHARS не определен	namespace ob_charspace	namespace ob_wcharspace
Макрос OB_USER_VERSION_CHARS определен	namespace ob_charspace191	namespace ob_wcharspace191

Пример: если при компиляции макрос **OB_USER_VERSION_CHARS** определен и интерпретатор транслируется без поддержки кодировки **Unicode**, то код интерпретатора версии 1.91 будет находиться в пространстве имен **namespace ob_charspace191**.

3. Макрос **OB_USER_HEADERS**.

С помощью макроса **OB_USER_HEADERS** можно определить файлы библиотечных заголовков для компилятора, отличного от **MSVC2010**.

Если при компиляции макрос **OB_USER_HEADERS** не определен, то файлы библиотечных заголовков имеют вид, принятый в компиляторах **MSVC2010** или **Borland C++ Builder 6.0**.

Если при компиляции макрос **OB_USER_HEADERS** определен, то файлы библиотечных заголовков должны быть определены пользователем.

Для переопределения файлов библиотечных заголовков используется механизм замены основного заголовочного файла интерпретатора.

Создается пользовательский файл **ob.h** со следующим содержанием:

```
#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 <obmain.h>

#endif
```

В проекте используется пользовательский файл **ob.h**, при этом файлы библиотечных заголовков будут иметь имена, нужные для компилятора пользователя.

4. Макросы **OB_USER_INT_AND_FLOAT** и **OB_USER_POW**.

С помощью макроса **OB_USER_INT_AND_FLOAT** можно переопределить типы переменных с фиксированной точкой и переменных с плавающей точкой, с которыми работает интерпретатор.

Если при компиляции макрос **OB_USER_INT_AND_FLOAT** не определен, то переменные с фиксированной точкой в интерпретаторе соответствуют типу **int** а переменные с плавающей точкой в интерпретаторе соответствуют типу **float** в используемом компиляторе **C++**.

Если при компиляции макрос **OB_USER_INT_AND_FLOAT** определен, то переменные фиксированной точкой и переменные с плавающей точкой в интерпретаторе должны быть определены пользователем с помощью директивы **typedef**.

При изменении типов переменных нужно учитывать, что в интерпретаторе операция возведения в степень использует функцию возведения в степень **pow(double x, double y)** из **math.h**. Поэтому одновременно с изменением типов переменных в интерпретаторе нужно изменить тип функции, которая обслуживает операцию возведения в степень. Также нужно переопределить тип **ob_type_pow**, который используется в качестве аргументов для функции возведения в степень.

С помощью макроса **OB_USER_POW** можно переопределить функцию, которая используется в операторе возведения в степень и типы аргументов этой функции.

Если при компиляции макрос **OB_USER_POW** не определен, то в интерпретаторе для возведения в степень используется функция **double pow(double x, double y)** из **math.h**.

Для переопределения всех этих параметров используется механизм замены основного заголовочного файла интерпретатора.

Создается пользовательский файл **ob.h** со следующим содержанием:

```

#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 <obmain.h>

#endif

```

В проекте используется **ob.h**, при этом в интерпретаторе операция возведения в степень использует функцию **powl(long double x, long double y)**. В проекте используется пользовательский файл **ob.h**, при этом переменные целого типа в интерпретаторе будут соответствовать типу **long int**, а переменные с плавающей точкой будут соответствовать типу **long double**.

5. Макросы **OB_USER_MAXLENGTHSTRING**, **OB_USER_MAXNUPPAR**, **OB_USER_MAXARRAYDIMENTION** и **OB_USER_MAXNUPPARSUBROUTINE**.

С помощью макроса **OB_USER_MAXLENGTHSTRING** можно переопределить максимальную длину строки, с которой работает интерпретатор (по умолчанию 4096). С помощью макроса **OB_USER_MAXNUPPAR** можно переопределить максимальное количество параметров в пользовательской функции (по умолчанию 64). С помощью макроса **OB_USER_MAXARRAYDIMENTION** можно переопределить максимальное количество измерений массива в интерпретаторе (по умолчанию 20). С помощью макроса **OB_USER_MAXNUPPARSUBROUTINE** можно переопределить максимальное количество параметров в подпрограмме с параметрами (в версии 1.91 не поддерживается).

Для переопределения всех этих значений используется механизм замены основного заголовочного файла интерпретатора.

Например, чтобы интерпретатор работал со строками максимальной длины 8192 символа, нужно создать файл **ob.h** со следующим содержанием:

```

#ifndef OB_THIS
#define OB_THIS

#define OB_USER_MAXLENGTHSTRING (8192)

#include <obmain.h>

#endif

```

6. Макрос **OB_USER_HASHLENGTH**.

Многие объекты интерпретатора, такие как метки, переменные, массивы хранятся в хэш-таблицах. Хэш-таблица в Open Basic представляет собой вектор списков. Длина хэш-таблицы в Open Basic это длина этого вектора.

С помощью макроса **OB_USER_HASHLENGTH** можно переопределить длину хэш-таблиц в интерпретаторе.

Этот макрос влияет на быстродействие интерпретатора.

Общее правило состоит в том, что длина хэш-таблицы должна быть в 10-100 раз меньше, чем количество содержащихся в ней объектов.

Сейчас по умолчанию **OB_USER_HASHLENGTH** равен 131. Это соответствует ***.bas-программам**, которые содержат порядка тысяч меток, переменных и массивов.

Если в интерпретаторе будут выполняться достаточно большие ***.bas-программы**, которые содержат десятки тысяч меток, переменных и массивов, то для сохранения быстродействия необходимо увеличить длину хэш-таблиц в интерпретаторе до значений порядка тысяч.

Для переопределения этого значения используется механизм замены основного заголовочного файла интерпретатора.

Чтобы увеличить длину хэш-таблиц в интерпретаторе до 1031, нужно создать файл **ob.h** со следующим содержанием:

```

#ifndef OB_THIS
#define OB_THIS

#define OB_USER_HASHLENGTH (1031)

#include <obmain.h>

#endif

```

При доступе к данным в хэш-таблице индекс вектора в **Open Basic** вычисляется как остаток от деления хэш-значения хранимого объекта на длину хэш-таблицы.

В качестве длины хэш-таблицы **OB_USER_HASHLENGTH** рекомендуется использовать простое число, так как остаток от целочисленного деления на простое число лучше приближает равномерное распределение.

7. Макрос **_CRT_SECURE_NO_WARNINGS**.

В интерпретаторе есть несколько предопределённых пользовательских функций. Две из них возвращают текущее время и текущую дату. Для работы эти функции используют вызов **localtime**. Вызов **localtime** также используется в файле примера **ob.cpp**. Однако, в **MSVC2010** использование **localtime** вызывает появление предупреждения:

```

warning C4996: 'localtime': This function or variable may be unsafe. Consider using localtime_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help for details.

```

Чтобы подавить появление этого предупреждения нужно использовать макрос **_CRT_SECURE_NO_WARNINGS**.

В интерпретаторе не используется вызов **localtime_s**, чтобы сохранить совместимость с более ранними версиями компиляторов, в которых не определен вызов **localtime_s**.

4.4. Обзор системы ввода-вывода.

Существует два альтернативных способа организации ввода-вывода данных в интерпретаторе.

1. С использованием операторов **PRINT** и **INPUT** и классов ввода-вывода.
2. С использованием пользовательских функций ввода-вывода.

4.4.1. Первый способ организации ввода-вывода (с использованием классов ввода-вывода).

Ввод-вывод в интерпретаторе Open Basic организован с помощью операторов ввода и вывода **PRINT** и **INPUT**. Оператор вывода **PRINT** форматирует выходную строку. Оператор ввода **INPUT** форматирует входную строку.

Во время исполнения ***.bas-программы** интерпретатор работает с внутренними переменными, массивами, и пользовательскими функциями. Все действия с этими объектами выполняются только средствами исполняющей системы интерпретатора, без обращения к сервисам ОС. Однако, вид и способ ввода-вывода может сильно различаться для разных ОС и разных приложений. Поэтому операторы ввода и вывода **PRINT** и **INPUT** требуют обращения к сервисам ОС. Для сопряжения ввода-вывода интерпретатора с вводом-выводом операционной системы используются методы **setoperatorbreak**, **setin**, **setout**.

Оператор **PRINT** в интерпретаторе выводит данные в текстовый поток вывода. Этот поток вывода прикрепляется к интерпретатору методом **setout**. Обычно в качестве потока вывода используется класс **ostrstream** со статическим буфером. Далее данные, подготовленные оператором **PRINT**, нужно вывести в поток, определенный пользователем. В консольном примере в качестве финишного потока вывода отформатированной строки используется поток **cout**.

Оператор **INPUT** в интерпретаторе вводит данные из текстового потока ввода. Этот поток ввода прикрепляется к интерпретатору методом **setin**. Обычно в качестве потока вывода используется класс **istrstream** со статическим буфером. В консольном примере в качестве стартового потока ввода для ввода символов с консоли используется поток **cin**.

Для сопряжения операторов **PRINT** и **INPUT** с операционной системой пользователь должен создать два класса, производных от класса **ob_base_class_for_IO_service**. Базовый класс **ob_base_class_for_IO_service** имеет вид:

```
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;
};
```

Классы, производные от класса **ob_base_class_for_IO_service**, называются классами ввода-вывода для соответствующего оператора.

В классах, производных от **ob_base_class_for_IO_service**, пользователь переопределяет методы **run_before** и **run_after**. После этого создаются экземпляр пользовательского класса ввода-вывода и с помощью метода **setoperatorbreak** пользователь устанавливает указатель на экземпляр пользовательского класса ввода-вывода в интерпретаторе.

Методы **run_before** и **run_after** будут запущены соответственно до и после выполнения операторов ввода-вывода. В методах **run_before** пользователь подготавливает потоки ввода-вывода для интерпретатора. В методах **run_after** пользователь обрабатывает результаты ввода-вывода.

Экземпляры классов ввода-вывода должны находиться в той же области видимости, что и экземпляр интерпретатора **ob_obasic**. Если в программе есть несколько экземпляров интерпретатора **ob_obasic**, то для каждого экземпляра интерпретатора **ob_obasic** нужно создать свои экземпляры классов ввода-вывода (для обеспечения работы в многопоточном окружении).

Пример:

```
//Создание класса ввода-вывода для оператора PRINT
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
//-----
}
}
};
```

```
//Создание класса ввода-вывода для оператора INPUT
class ob_class_input_service_def:public ob_base_class_for_IO_service{
char inputbuffer[ob_maxlengthstring];
char inputbufferl[ob_maxlengthstring];
strstream pr;
public:
ob_class_input_service_def():pr(inputbufferl,ob_maxlengthstring){}
virtual ~ob_class_input_service_def(){}
void run_before(ob_obasic* basicptr){
ob_lex::typelex tll=basicptr->gettypenextlex();
if(tll!=ob_lex::SHARP){
inputbuffer[0]=0;
ob_type_char ch;
ob_type_stringsize i;
for(i=0; i<ob_maxlengthstring; 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;
}
}
}
pr.clear();
pr.seekp(0);
pr<<inputbuffer;
pr.seekg(0);
basicptr->setin(&pr);
}
};
```

```

}
void run_after(ob_obasic* basicptr){
};

//создание экземпляров классов ввода-вывода
ob_class_print_service_def ob_class_print_service;
ob_class_input_service_def ob_class_input_service;

//установка ссылки на классы ввода-вывода в экземпляре интерпретатора
example_basic_interpreter.setoperatorbreak("PRINT",&ob_class_print_service);
example_basic_interpreter.setoperatorbreak("INPUT",&ob_class_input_service);

```

В примере ввод-вывод осуществляется из/в стандартные потоки **cin** и **cout**. Приведенные примеры классов ввода-вывода можно использовать, если заменить выделенные строки:

```

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

```

пользовательским кодом ввода-вывода из/в соответствующего окна **Windows**.

Для оператора **PRINT** вывод осуществляется построчно. Для оператора **INPUT** ввод осуществляется посимвольно. Различия связаны с тем, что операция **cin.get(ch)** блокирует работу приложения.

Подробнее об организации ввода-вывода с использованием классов ввода-вывода см. п. "Методы и перечисления класса **ob_obasic**, которые используются для сопряжения ввода и вывода интерпретатора **Open Basic** с вводом и выводом операционной системы".

4.4.2. Второй способ организации ввода-вывода (с использованием пользовательских функций ввода-вывода).

Ввод-вывод может быть организован без использования операторов **PRINT** и **INPUT** с помощью пользовательских функций ввода-вывода. Пользователь пишет свои пользовательские функции ввода-вывода и регистрирует эти функции в интерпретаторе. При таком способе организации ввода-вывода тексте ***.bas-программ** вместо операторов **PRINT** и **INPUT** используются пользовательские функции ввода-вывода.

4.5. Методы и перечисления класса **ob_obasic**, которые используются для загрузки и запуска *.bas-программ.

4.5.1. Конструктор **ob_obasic::ob_obasic**.

Конструктор

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

создает экземпляр интерпретатора **Open Basic**.

4.5.2. Метод **ob_obasic::clear_project**.

Метод

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

очищает все таблицы интерпретатора, кроме таблицы пользовательских функций.

В процессе работы интерпретатор создает таблицы:

1. Таблицы переменных (локальных и глобальных).
2. Таблицы массивов (локальных и глобальных).
3. Таблицы меток (локальных и глобальных).
4. Таблицы циклов оператора **FOR** (таблицы циклов всегда локальные).
5. Таблица пользовательских функций (одна глобальная таблица для каждого экземпляра интерпретатора).
6. Таблица открытых файлов (одна глобальная таблица для каждого экземпляра интерпретатора).
7. Таблица строк оператора **CHECKLOAD** (одна глобальная таблица для каждого экземпляра интерпретатора).
8. Таблица (поток) данных оператора **DATA**.
9. Стек подпрограмм **GOSUB**.

Таблица пользовательских функций очищается только при вызове метода **void ob_obasic::clrtablfun()**; или при вызове деструктора класса **ob_obasic::~ob_obasic**.

Метод **clear_project** должен всегда применяться перед началом работы интерпретатора для очистки таблиц интерпретатора от предыдущих данных.

4.5.3. Метод **ob_obasic::load_project** и перечисление **ob_obasic::loadresult**.

Метод

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

загружает одну или несколько ***.bas-программ** в интерпретатор. В процессе загрузки создаются таблицы локальных и глобальных меток.

Параметры:

1. Параметр **i** это указатель на входной поток, в котором содержится ***.bas-программа**.
2. Параметр **id** это указатель на идентификационную строку входного потока. Обычно в качестве идентификационной строки используется имя файла. Идентификационная строка может использоваться для отладки при доступе к локальным переменным. Если идентификационную строку не планируется использовать, то в качестве параметра **id** можно передавать нулевой указатель.

ВНИМАНИЕ! Поток **i** должен быть открыт в **ios::binary** режиме.

ВНИМАНИЕ! Указатель на данные в потоке **i** должен указывать на начало программы.

После окончания работы интерпретатора указатель на данные в потоке указывает на конец программы.

Метод **load_project** может загрузить несколько потоков. Набор одновременно загруженных потоков называется проектом.

Метод `load_project` возвращает значение из перечисления:

```
enum ob_obasic::loadresult(LOAD_OK,CHECKLOAD_DUPLICATE,STREAM_DUPLICATE,STREAM_NULL);
```

1. **LOAD_OK** – нормальное завершение загрузки.
2. **CHECKLOAD_DUPLICATE** – загрузка прервана из-за обнаружения повторной загрузки оператором **CHECKLOAD**. При загрузке ***.bas-программ** интерпретатор составляет таблицу строк операторов **CHECKLOAD**. При обнаружении уже существующей в этой таблице строки загрузка прерывается с кодом возврата **CHECKLOAD_DUPLICATE**. Подробнее см. описание оператора **CHECKLOAD**.
3. **STREAM_DUPLICATE** – загрузка прервана из-за обнаружения повторной загрузки одного и того же потока. При загрузке ***.bas-программ** для каждого потока интерпретатор создает таблицы для локальных переменных, массивов и меток. Дополнительно к локальным переменным, массивам и меткам эти таблицы содержат указатель на входной поток. При обнаружении уже существующей таблицы для потока загрузка прерывается с кодом возврата **STREAM_DUPLICATE**.
4. **STREAM_NULL** – загрузка прервана так как в качестве параметра **i** передан нулевой указатель.

4.5.4. Метод `ob_obasic::run` и перечисление `ob_obasic::typeend`.

Метод

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

запускает загруженную программу или продолжает выполнение программы после останова. Если проект состоит более чем из одного потока, то запускается последний загруженный поток. Для явного указания запускаемого потока нужно применять метод `set_current_input_stream`.

Метод `run` возвращает значение из перечисления:

```
enum ob_obasic::typeend(DUMMYEND,ENDFILEEND,ENDOPERATORDETECT,EOLEND,BREAKPOINTEND,NOLOADEND);
```

Коды завершения:

1. **DUMMYEND** – при нормальной работе никогда не возвращается методом `run`. Используется для внутренних переходов.
2. **ENDFILEEND** – метод `run` окончился из-за исчерпания входного потока. Это нормальное завершение ***.bas-программы**, если в ***.bas-программе** отсутствует оператор **END**.
3. **ENDOPERATORDETECT** – метод `run` окончился при обнаружении оператора **END**. Это нормальное завершение ***.bas-программы**.
4. **EOLEND** – метод `run` окончился после исполнения очередной строки, т.к. был установлен флаг `step=STEP` методом `setstep`.
5. **BREAKPOINTEND** – метод `run` окончился после считывания символа '@', который является символом точки останова для **Open Basic**.
6. **NOLOADEND** – метод `run` окончился т.к. не был выполнен метод `load_project` и входной поток не открыт.

Обычно метод `run` запускает в отдельном потоке, который по приоритету ниже чем основной поток приложения.

4.5.5. Метод `ob_obasic::set_current_input_stream`.

Метод

```
void ob_obasic::set_current_input_stream(ob_type_istreamcommon* i);
```

устанавливает в интерпретаторе поток **i**, который должен начать исполняться первым при вызове метода `run()`. Поток **i** должен быть предварительно загружен с помощью метода `load_project`.

4.5.6. Метод `ob_obasic::get_current_input_stream`.

Метод

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

возвращает текущий исполняемый поток. Метод `ob_type_istreamcommon* ob_obasic::get_current_input_stream()` используется для доступа к локальным переменным интерпретатора **Open Basic**.

4.6. Методы и перечисления класса `ob_obasic`, которые используются для организации отладки в интерпретаторе **Open Basic**.

В **Open Basic** есть два режима отладки: пошаговое выполнение ***.bas-программ** и назначение точек останова.

Возможность пошагового выполнения ***.bas-программ** реализована с помощью метода `setstep`.

Символ '@' отдельной строке в тексте ***.bas-программы** реализует точку останова. Точек останова в ***.bas-программе** может быть несколько. После внесения хотя бы одной точки останова в ***.bas-программу** весь проект нужно очистить методом `clear_project` и повторно перезагрузить методом `load_project`.

При передаче управления на строку с символом '@' произойдет останов программы. Код завершения метода `run` будет **BREAKPOINTEND**. Для продолжения программы после точки останова нужно снова запустить метод `run()`.

4.6.1. Перечисление `ob_obasic::step`.

Перечисление

```
enum ob_obasic::step(NOSTEP,STEP);
```

определяет режим пошагового выполнения программы. Перечисление `step` используется в методах `setstep` и `getstep`.

При `step=NOSTEP` программа выполняется непрерывно.

При `step=STEP` программа завершается с кодом `typeend=EOLEND` после выполнения каждой строки программы.

По умолчанию `step=NOSTEP`.

4.6.2. Метод `ob_obasic::setstep`.

Метод

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

устанавливает или снимает режим пошагового выполнения программы. Этот метод можно использовать в любой момент работы интерпретатора. При установке пошагового режима метод `run()` будет завершаться после выполнения каждой строки ***.bas-программы**. Для продолжения программы нужно снова запустить метод `run()`; Обычно метод `setstep` вызывает из потока, отдельного от потока метода `run`.

4.6.3. Метод `ob_obasic::getstep`.

Метод

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

возвращает текущий режим пошагового выполнения программы.

4.6.4. Функция `ob_loadbreakstr`.

Функция

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

Функция **ob_loadbreakstr** ищет текущую, предыдущую и последующую строки в файле и загружает в указанные буфера соответствующие строки выполняемой программы. Обычно функция **ob_loadbreakstr** после останова ***.bas-программы**.

Параметры:

```
ob_type_istreamcommon* in    - текущая входная исполняемая *.bas-программа
lengthbuffers                - длина буферов ptrbefore, ptrcurrent, ptrafter
ptrbefore                    - буфер для строки, предшествующей текущей исполняемой
ptrcurrent                    - буфер для текущей исполняемой строки
ptrafter                      - буфер для строки, следующей за текущей исполняемой
```

Функция **ob_loadbreakstr** не является методом класса **ob_obasic**.

Пример:

```
//Создание интерпретатора и запуск программы в пошаговом режиме с печатью выполняемых строк.
ob_type_char str_before[ob_maxlengthstring];//буфер
ob_type_char str_current[ob_maxlengthstring];//буфер
ob_type_char str_after[ob_maxlengthstring];//буфер

ob_obasic basic_interpreter;//Создание интерпретатора
//загрузка и запуск программы test1.bas
//с пошаговым выполнением программы
//и с печатью выполняемых строк

ifstream inpl("test1.bas",ios::binary);//открытие входного потока
basic_interpreter.clear_project();//очистка проекта
basic_interpreter.load_project(&inpl,0);//загрузка программы test1.bas
basic_interpreter.setstep(ob_obasic::STEP);//установка пошагового режима
while(1){
if(basic_interpreter.run()!=ob_obasic::EOLEND) break;//запуск *.bas-программы
ob_loadbreakstr(&inpl,ob_maxlengthstring,str_before,str_current,str_after);
cout<<endl<<"String current: "<<str_current;
} //while
```

4.7. Методы и перечисления класса **ob_obasic**, которые используются для сопряжения ввода и вывода интерпретатора **Open Basic** с вводом и выводом операционной системы.

4.7.1. Общие сведения.

Во время исполнения ***.bas-программы** интерпретатор работает с внутренними переменными, массивами, и пользовательскими функциями. Все действия с этими объектами выполняются только средствами исполняющей системы интерпретатора, без обращения к сервисам ОС. Однако, вид и способ ввода-вывода может сильно различаться для разных ОС и разных приложений. Поэтому операторы ввода и вывода **PRINT** и **INPUT** требуют обращения к сервисам ОС. Для сопряжения ввода-вывода интерпретатора с вводом-выводом операционной системы используются методы **setoperatorbreak**, **setin**, **setout**.

4.7.2. Методы **ob_obasic::setoperatorbreak** и **ob_obasic::getoperatorbreak**.

В **Open Basic** для каждого оператора (**PRINT**, **INPUT**, **FOR** и т.д.) имеется возможность выполнить пользовательский код до и после выполнения оператора. Обычно эта возможность используется только для операторов **PRINT**, **INPUT**.

Метод

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

устанавливает указатель на класс ввода-вывода для оператора «name».

Параметры:

1. **name** - имя оператора
2. **ob_base_class_for_IO_service* b** - указатель на экземпляр класса ввода-вывода (производный от класса **ob_base_class_for_IO_service**)

возвращаемое значение:

true - успешное завершение
false - оператор не найден, возможно, произошла ошибка задания имени оператора

Метод

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

позволяет получить указатель на класс ввода-вывода. Если для указанного оператора класс ввода-вывода не был установлен, то **b=0**. Возвращает **false** при ошибке задания имени оператора. По умолчанию для всех операторов классы ввода-вывода не установлены.

Параметры:

1. **name** - имя оператора
2. **ob_base_class_for_IO_service** b** - указатель на указатель на экземпляр класса ввода-вывода (производный от класса **ob_base_class_for_IO_service**)

возвращаемое значение:

true - успешное завершение
false - оператор не найден, возможно, произошла ошибка задания имени оператора

4.7.3. Методы **ob_obasic::setin** и **ob_obasic::setout**.

Методы

```
void setin(ob_type_istreamcommon* i);
void setout(ob_type_ostreamcommon* o);
```

позволяют переопределить входной и выходной потоки.

ВНИМАНИЕ! Входной поток должен быть открыт в **ios::binary** режиме.
ВНИМАНИЕ! Выходной поток должен быть открыт в **ios::text** режиме.

Методы **setin** и **setout** используются для организации ввода-вывода.

4.7.4. Перечисление `ob_lex::typelex`.

Перечисление

```
enum ob_lex::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, LABEL, BYTE_ORDER_MARK1
};
```

определяет тип лексемы.

Перечисление `ob_lex::typelex` используется в методе `ob_obasic::gettypenextlex()`.

4.7.5. Метод `ob_obasic::gettypenextlex`.

Метод

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

предназначен для облегчения реализации операторов ввода и вывода.

Перед запуском переопределенных методов класса ввода-вывода исполняющая система интерпретатора производит выборку и распознавание следующей лексемы. Для доступа к типу следующей лексемы метод `gettypenextlex` можно применять только из переопределенных методов класса ввода-вывода. При завершении метода `run` доступ к типу следующей лексемы через метод `gettypenextlex` не поддерживается.

Метод возвращает тип следующей лексемы. Некоторые операторы (например оператор `INPUT`) имеют одинаковый синтаксис для ввода из файла и с консоли. Текущее направление ввода определяется следующей за именем оператора лексемой «#» (лексема `SHARP`). С помощью метода `gettypenextlex()` можно выяснить тип следующей лексемы. И таким образом можно выяснить, откуда ожидается ввод.

В примере п. "Обзор системы ввода-вывода" метод `gettypenextlex` используется для определения направления ввода в операторе `INPUT`. Если обнаружена лексема `SHARP`, то ввод с консоли не производится. Соответствующий код проверки типа следующей лексемы для оператора `PRINT` не нужен, так как вывод не блокирует работу.

4.8. Синтаксис языка *Open Basic*. Общие сведения.

Интерпретатор *Open Basic* поддерживает подмножество операторов языка *Basic*. Эти операторы описаны ниже. Имеется возможность работать с тремя типами переменных:

1. Переменные с плавающей точкой
2. Переменные с фиксированной точкой (целые переменные)
3. Строковые переменные

Имеется возможность работать с массивами из этих трех типов переменных.

Все данные для переменных и массивов располагаются в свободной памяти оператором `new`.

Переменные в *Open Basic* не нужно описывать перед первым использованием. Первое появление переменной в программе должно быть слева от оператора присвоения.

Массивы в *Open Basic* нужно описывать до первого использования оператором `DIM`.

Массивы могут быть многомерными. Максимальные размеры массива зависят от ограничений реализации оператора `new` в применяемой операционной системе.

При создании массива оператором `DIM` элементы числовых массивов инициализируются нулями, элементы строковых массивов инициализируются пустыми строками.

В *Open Basic* индекс массива начинается с единицы.

В отличие от стандартного языка *Basic* оператор `DIM` в *Open Basic* является исполняемым. Он выполняется каждый раз, когда на него передается управление.

Допускается повторное описание массива с тем же именем в операторе `DIM`. Новый массив может иметь другие размерности. Старые данные при этом теряются.

Повторное описание массива можно применять для освобождения памяти. Например, после создания большого массива оператором `DIM` можно освободить память, описав массив с тем же именем и размером в один элемент.

При описании массива в операторе `DIM` в качестве размерностей могут использоваться не только константы, но переменные целого типа.

Диапазон представления данных с плавающей точкой и данных с фиксированной точкой зависит от того, как транслируется интерпретатор. Обычно целые это `int`, а плавающие это `float`. В файле `ob.h` типы переменных определены с помощью `typedef`:

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

Имеется возможность переопределить типы `ob_type_flo` и `ob_type_int` с помощью макросов условной трансляции.

Максимальная длина строковых переменных определяется константой `ob_maxlengthstring` в файле `ob.h`. Для *Open Basic* версии 1.91 `ob_maxlengthstring=4096`.

ВНИМАНИЕ! Максимальная длина константных строк и имен в *Open Basic* не может превышать `ob_maxlengthstring`. Однако на операцию суммирования строк это ограничение не распространяется. Поэтому, если сложить две строковых переменных длиной `ob_maxlengthstring`, то длина суммарной строки будет $(2 * ob_maxlengthstring)$. Таким образом, операция суммирования строк позволяет получить строки любой длины.

В **.bas-программе* могут быть пустые строки.

Интерпретатор *Open Basic* поддерживает следующие операции над типами с фиксированной точкой и типами с плавающей точкой:

```
+ - сложение
- - вычитание
* - умножение
/ - деление
^ - возведение в степень
- - унарный минус
+ - унарный плюс
= - присвоение
```

Приоритеты операций стандартные для *C++*. Приоритет операций можно изменить с помощью скобок.

Интерпретатор *Open Basic* поддерживает следующие выражения отношения:

```
< - "меньше"
> - "больше"
= - "равно"
<> - "не равно"
<= - "меньше или равно"
>= - "больше или равно"
```

Выражения отношения используются только в операторе `IF`.

Для строковых данных поддерживаются только операции сложения, сравнения и присвоения строк. Операции сравнения строк производятся по правилам сравнения массивов в шаблонном классе `mvect` (`[mktmk]vect[or]`). Класс `mvect` описан в файле `mvect.h`.

4.9. Типы данных и имена *Open Basic*.

В качестве имен переменных, массивов и пользовательских функций можно использовать любые последовательности букв и цифр. Имя переменной или массива должно начинаться с буквы.

Символ подчеркивания '_', символ денежной единицы '\$' и символ процента '%' считаются буквами и могут входить в состав имени.

Тип переменной или массива в **Open Basic** определяется по имени. Правила определения типа переменной в **Open Basic**:

1. Если в имени последним символом является символ '%', то это переменная или массив целого типа.
2. Если в имени последним символом является символ '\$', то это переменная или массив строкового типа.
3. Если в имени последним символом является любой другой символ, то это переменная или массив с плавающей точкой.

Кроме типа каждой переменной и массиву присвоено свойство глобальности/локальности.

Локальность или глобальность переменной или массива в **Open Basic** определяется по имени, так же, как и тип. Правила определения локальной/глобальной переменной в **Open Basic**:

1. Если в имени переменной или массива первым символом является символ подчеркивания '_', то локальная переменная или локальный массив.
2. Если в имени переменной или массива первым символом не является символ подчеркивания '_', то это глобальная переменная или глобальный массив.

Глобальные переменные и глобальные массивы доступны из любого файла проекта.

Локальные переменные и локальные массивы доступны только из того файла, в котором они определены.

Многие типы **Open Basic** описаны в файле **obmain.h** с помощью **typedef**, например:

1. Тип **ob_type_flo** отображается в тип **float** с помощью **typedef**.
2. Тип **ob_type_int** отображается в тип **int** с помощью **typedef**.

В пользовательских функциях рекомендуется использовать имена типов **ob_type_flo** и **ob_type_int**.

При смене типов, например, при смене **ob_type_flo** с **float** на **double** будет обеспечена совместимость типов.

Буквы в имени могут быть прописные или строчные. Приведения букв в именах к одному регистру не производится, поэтому имена **ARRAY**, **array** и **ArRay** являются разными именами.

Максимальная длина имени определяется константой **ob_maxlengthstring** в файле **ob.h**. Для **Open Basic** версии 1.91 **ob_maxlengthstring=4096**.

Пример:

Объявление глобального массива **array** и инициализация двух глобальных переменных - целой переменной **counter%** и переменной с плавающей точкой **data**. Заметьте, что последнее измерение массива **array** задано переменной. Тип элементов массива **array** - с плавающей точкой. Переменная **counter%** имеет целый тип. Переменная **data** имеет тип с плавающей точкой. Все массивы и переменные в этом примере являются глобальными, так как их имена не начинаются со знака подчеркивания.

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

Пример:

Объявление локального массива **_array** и инициализация двух локальных переменных - целой переменной **_counter%** и переменной с плавающей точкой **_data**. Заметьте, что последнее измерение массива **_array** задано переменной. Тип элементов массива **_array** - с плавающей точкой. Переменная **_counter%** имеет целый тип. Переменная **_data** имеет тип с плавающей точкой. Все массивы и переменные в этом примере являются локальными, так как их имена начинаются со знака подчеркивания.

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

Пример:

Объявление глобального массива **array** и использование его. Повторное объявление массива с тем же именем, но другой размерностью и использование его.

```
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%
```

Если в выражении используются данные разных типов, то производится приведение типов. Если в выражении используются целые данные и данные с плавающей точкой, результирующий тип будет с плавающей точкой. При присвоении тип правого выражения приводится к типу выражения слева от знака присвоения **=**. Если слева от знака присвоения целая переменная, а справа с плавающей точкой, то дробная часть отбрасывается (в соответствии с правилами округления в C++).

4.10. Метки Open Basic.

Интерпретатор **Open Basic** поддерживает два типа меток:

1. Метки в виде номеров строк.
2. Строковые метки с финальным двоеточием.

Каждой метке в интерпретаторе **Open Basic** присвоено свойство глобальности/локальности.

Глобальные метки доступны из любого файла проекта.

Глобальные метки служат для передачи управления в другой файл проекта.

Используя операторы **GOTO** или **GOSUB** можно передать управление в другой файл проекта на строку с глобальной меткой.

Локальные метки доступны только из того файла, в котором они определены.

Осуществить переход операторами **GOTO** и **GOSUB** можно только строку с меткой. Можно поставить метку и у пустой строки.

4.10.1. Метки в виде номеров строк.

Метки в виде номеров строк всегда глобальные.

Метки в виде номеров строк должны стоять в начале строки и отделяться как минимум одним пробелом от других лексем. Номера строк не обязательно должны идти подряд.

Пример:

```
ii%=6
ik%=8

IF ii%<>6 THEN GOTO 10

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

```

IF ik%=7 THEN GOSUB 11
PRINT "ik%=";ik%;
GOTO 12
11 PRINT " ik%!=7"
RETURN
12 STOP
END

```

Метки в виде номеров строк признаны устаревшими. В следующих версиях интерпретатора они не будут поддерживаться. Пользователям не рекомендуется применять метки в виде номеров строк в новых разработках.

4.10.2. Строковые метки с финальным двоеточием.

Начиная с версии 1.80 в **Open Basic** кроме меток в виде номеров строк поддерживаются строковые метки с финальным двоеточием. Двоеточие является частью идентификатора метки и должно следовать за идентификатором метки без разделителя. В операторах **GOTO** и **GOSUB** идентификатор строковой метки пишется без финального двоеточия. В программе могут одновременно применяться метки в виде номеров строк и строковые метки. Строковая метка с финальным двоеточием должна начинаться с буквы.

Строковые метки с финальным двоеточием могут быть локальными и глобальными.

Локальность или глобальность строковой метки с финальным двоеточием в **Open Basic** определяется по имени, так же, как и локальность переменных и массивов. А именно:

1. Если в имени метки первым символом является символ подчеркивания '_', то это локальная метка.
2. Если в имени метки первым символом не является символ подчеркивания '_', то это глобальная метка.

Пример:

Создание и использование глобальных строковых меток с финальным двоеточием.

```

REM Test global string label
PRINT
PRINT "Start-----"
PRINT
GOSUB gosub_label_1
STOP
END
gosub_label_1:
PRINT "Work local label"
RETURN

```

Пример:

Создание и использование локальных строковых меток с финальным двоеточием.

```

REM Test local string label
PRINT
PRINT "Start-----"
PRINT
GOSUB _gosub_label_1
STOP
END
_gosub_label_1:
PRINT "Work local label"
RETURN

```

Глобальные метки должны быть уникальными во всех файлах проекта.

Локальные метки должны быть уникальными в том файле, в котором они определены.

При обнаружении повторных глобальных меток в разных файлах проекта или в одном файле проекта метод **load_project** заканчивается генерацией ошибки.

При обнаружении повторных локальных меток в одном файле проекта метод **load_project** заканчивается генерацией ошибки.

4.11. Пользовательские функции в Open Basic.

Open Basic имеет возможность расширения системы команд путем подключения пользовательских функций к исполняющей системе **Open Basic**.

Параметры пользовательских функций позволяют определить тип и порядок следования аргументов в вызове во время исполнения программы.

Пользовательские функции подключаются к исполняющей системе **Open Basic** под **Basic-именами**. **Basic-имена** пользовательских функций не обязательно должны совпадать с именами пользовательских функций в **C++**. Каждая пользовательская функция подключается под своим **Basic-именем**. После подключения к исполняющей системе **Open Basic** пользовательскую функцию можно вызывать в тексте ***.bas-программы** по **Basic-имени**. Можно передавать пользовательским функциям параметры и получать от пользовательских функций результаты. Можно вызывать пользовательские функции без параметров.

Пользовательские функции могут быть написаны на **C/C++**, ассемблере или других языках.

Каждая пользовательская функция в **Open Basic** имеет 2 имени:

1. **C++** имя пользовательской функции в составе проекта, по которому пользовательскую функцию можно вызывать из **C++** программы.
2. **Basic-имя** функции, по которому пользовательскую функцию можно вызывать из ***.bas-программы**.

Первое имя пользовательские функции получают при их написании на языке **C++**.

Второе имя пользовательские функции получают при их подключения к исполняющей системе **Open Basic** с помощью метода **setfun**.

Пользовательская функция может иметь один из трех прототипов:

1. Функция, возвращающая значение с плавающей точкой **ob_type_flo**.
2. Функция, возвращающая значение с фиксированной точкой **ob_type_int**.
3. Функция, возвращающая строку **ob_type_char***.

Чтобы создать пользовательскую функцию и подключить ее к исполняющей системе **Open Basic** необходимо:

1. Выбрать тип возвращаемого значения для пользовательской функции.
2. Выбрать **C++** имя пользовательской функции.
3. Написать пользовательскую функцию по одному из трех прототипов.

4. Выбрать **Basic-имя** пользовательской функции.
5. Подключить пользовательскую функцию к исполняющей системе **Open Basic** с помощью метода `setfun` используя **Basic-имя** и **C++ имя** в качестве параметров.

После этого можно использовать **Basic-имя** пользовательской функции в ***.bas-программе**.

Пользовательские функции всегда глобальные.

Имена пользовательских функций не должны повторяться.

Если под одним и тем же именем будут подключены к исполняющей системе **Open Basic** несколько пользовательских функций, то из ***.bas-программы** будет вызываться последняя подключенная.

Есть возможность удалить пользовательскую функцию из исполняющей системы **Open Basic** с помощью метода `delfun`.

4.11.1. Прототипы для пользовательской функции.

Пользовательская функция может иметь один из трех прототипов:

1. Функция, возвращающая значение с плавающей точкой, имеет прототип следующего вида:

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

2. Функция, возвращающая значение с фиксированной точкой, имеет прототип следующего вида:

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

3. Функция, возвращающая строку, имеет прототип следующего вида:

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

Эти прототипы различаются только типом возвращаемого значения.

В файле **ob.h** содержится макрос `OB_DECLARE_USER_FUNCTION(name, typeret)`, который можно использовать для объявления пользовательских функций. Параметр **name** - это имя функции в **C++**, параметр **typeret** - это тип возвращаемого значения.

Пример:

Объявление пользовательской функции с помощью макроса `OB_DECLARE_USER_FUNCTION`. **C++** имя пользовательской функции **myfun1**. Тип возвращаемого значения - **ob_type_flo**.

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

Пример:

Определение пользовательской функции с помощью макроса `OB_DECLARE_USER_FUNCTION`. **C++** имя пользовательской функции **myfun2**. Тип возвращаемого значения - **ob_type_int**.

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

Пример:

Определение пользовательской функции с помощью макроса `OB_DECLARE_USER_FUNCTION`. **C++** имя пользовательской функции **myfun3**. Тип возвращаемого значения - **ob_type_char***.

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

4.11.2. Параметры пользовательских функций.

Интерфейс пользовательских функций **Open Basic** разработан для того, чтобы обеспечить возможность контроля фактических параметров в вызове на этапе исполнения программы.

Набор параметров пользовательских функций содержит две группы массивов.

Первая группа массивов это три массива описаний параметров **descri**, **descri**, **descri**. Массивы **descri**, **descri**, **descri** содержат информацию о типах, количестве и порядке следования фактических параметров в вызове пользовательской функции.

Вторая группа массивов это три массива значений параметров **parf**, **pari**, **parc**. Массивы **parf**, **pari**, **parc** содержат значения фактических параметров в вызове пользовательской функции.

Назначение и структура массивов **descri**, **descri**, **descri** и **parf**, **pari**, **parc** подробнее описана в следующем разделе.

4.11.3. Более подробное описание параметров пользовательских функций.

Существуют три типа пользовательских функций. Все они имеют одинаковые параметры и разный тип возвращаемого результата. Рассмотрим параметры пользовательских функций на примере функции, которая возвращает значение с плавающей точкой.

```
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  
);
```

Параметры пользовательских функций:

1. **ob_obasic* basic_ptr** - указатель на экземпляр интерпретатора **ob_obasic**, к которому подключена функция. Используется для доступа к переменным программы с помощью методов **ob_obasic**.
2. **const char* parstring** - строковые представления параметров.
3. Три массива описания параметров **descrf**, **descri**, **descrc**.
4. Три массива значений параметров **parf**, **pari**, **parc**.

Имя массива **parstring** образовано от слов **par[ameters] string**.

Имя массива **descrf** образовано от слов **descr[ription] f[loat]**.
Имя массива **descri** образовано от слов **descr[ription] i[nt]**.
Имя массива **descrc** образовано от слов **descr[ription] c[har]**.

Имя массива **parf** образовано от слов **par[ameters] f[loat]**.
Имя массива **pari** образовано от слов **par[ameters] i[nt]**.
Имя массива **parc** образовано от слов **par[ameters] c[har]**.

4.11.3.1. Структура параметра parstring.

Параметр **parstring** содержит все параметры текущего вызова пользовательской функции в виде строк.

Пример:

Если в качестве параметров функции **USER_FUN1** в ее вызове указано **USER_FUN1(1.123,a%,1+2)**, то в массиве **parstring** будут находиться следующие строки:
"1.123","a%", "1+2".

Эти строки будут лежать подряд, каждая строка будет заканчиваться нулевым байтом.

4.11.3.2. Структура массива descrf (массивы descri и descrc имеют аналогичную структуру).

Нулевой элемент массива **descrf[0]** содержит общее число параметров с плавающей точкой в текущем вызове. Остальные элементы массива **descrf** это порядковые номера параметров с плавающей точкой в текущем вызове. Порядковые номера фактических параметров в массиве **descrf** считаются с 1.

Пример:

a_a% - целая переменная
1.123 - константа с плавающей точкой
1+2 - целое выражение

Если в качестве параметров функции **USER_FUN1** в ее вызове указано **USER_FUN1(1.123,a%,1+2)**, то:

1. В массиве **descrf** будут находиться числа: 1,1.
2. В массиве **descri** будут находиться числа: 2,2,3.
3. В массиве **descrc** будет находиться число: 0.

Пример:

"str1" - строковая переменная
"str2" - строковая переменная
"str3" - строковая переменная
1.1 - константа с плавающей точкой
1.2 - константа с плавающей точкой
1.3 - константа с плавающей точкой
1.4 - константа с плавающей точкой

Если в качестве параметров функции **USER_FUN1** в ее вызове указано **USER_FUN1("str1","str2"+"str3",1.1,1.2,1.3,1.4)**, то:

1. В массиве **descrf** будут находиться числа: 4,3,4,5,6.
2. В массиве **descri** будет находиться число: 0.
3. В массиве **descrc** будут находиться числа: 2,1,2.

4.11.3.3. Структура массива parf (массивы pari и parc имеют аналогичную структуру).

Существуют три массива значений параметров:

1. **const ob_type_flo* parf** - массив значений для параметров с плавающей точкой.
2. **const ob_type_int* pari** - массив значений для параметров с фиксированной точкой.
3. **const ob_type_char* parc** - массив значений для строковых параметров.

Массивы **parf**, **pari**, **parc** это массивы, описывающие значения всех параметров соответствующих типов, переданных пользовательской функции в текущем вызове из ***.bas-**программы.

В массиве **parc** строки, как и в параметре **parstring**, отделяются нулевыми байтами.

Пример:

a_a% - целая переменная
[**a_a%**] - значение переменной с именем **a_a%**
1.123 - константа с плавающей точкой
1+2 - целое выражение

Если в качестве параметров функции **USER_FUN1** в ее вызове указано **USER_FUN1(1.123,a%,1+2)**, то:

1. В массиве **parf** будет находиться число: 1.123.
2. В массиве **pari** будут находиться числа: [**a_a%**],3.
3. В массиве **parc** не будет значений.

Пример:

"str1" - строковая переменная
"str2" - строковая переменная
"str3" - строковая переменная
1.1 - константа с плавающей точкой
1.2 - константа с плавающей точкой
1.3 - константа с плавающей точкой
1.4 - константа с плавающей точкой

Если в качестве параметров функции **USER_FUN1** в ее вызове указано **USER_FUN1("str1","str2"+"str3",1.1,1.2,1.3,1.4)**, то:

1. в массиве `parf` будут находиться числа: 1.1,1.2,1.3,1.4.
2. в массиве `pari` не будет значений.
3. в массиве `parc` будут находиться следующие строки: "str1","str2str3".

Максимальное число параметров в пользовательской функции определяется константой `ob_maxnumpar` в файле `ob.h`. Для Open Basic версии 1.91 `ob_maxnumpar=64`;

4.11.3.4. Использование параметров пользовательских функций.

Используя массивы `descrf`, `descri`, `descrc` и массивы `parf`, `pari`, `parc` пользователь может организовать контроль типов и контроль порядка следования параметров в текущем вызове пользовательской функции. Анализ массивов `descrf`, `descri`, `descrc` дает полную информацию о типах, количестве и порядке следования параметров в текущем вызове пользовательской функции.

Пример:

В пользовательскую функцию, вычисляющую синус, передана в качестве параметра строковая переменная. При этом массивы `descrf`, `descri` и `descrc` будут содержать значения:

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

Анализируя информацию в массивах `descrf`, `descri` и `descrc`, пользователь может совершить нужные ему действия, например:

1. Генерировать пользовательское исключение.
2. Подставить аргумент по умолчанию, например 0.
3. Выполнить останов программы с печатью предупреждения.
4. Попыаться преобразовать строку в число с помощью `atof`.

Для облегчения доступа к строковым данным Open Basic предоставляет функцию `ob_getstringparam`:

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

Параметры функции `ob_getstringparam`:

1. `descrc` - массив описания строковых параметров.
2. `parc` - массив значений строковых параметров.
3. `i` - номер строки, переданной в параметрах (нумерация с нуля).

С помощью функции `ob_getstringparam` можно получить доступ к *i*-й строке, записанной в строковые параметры или в имена пользовательской функции. Если `i` больше или равно, чем `descrc[0]`, то функция `ob_getstringparam` возвращает нулевой указатель. Если `descrc[0]` равно нулю, то функция `ob_getstringparam` возвращает нулевой указатель.

Функция `ob_getstringparam` не является методом класса `ob_obasic`.

4.11.4. Выбор Basic-имени пользовательской функции.

Basic-имя пользовательской функции должно отличаться от имен операторов, имен переменных, имен массивов и имен других пользовательских функций ***.bas-программы**.

4.12. Подключение пользовательской функции к исполняющей системе Open Basic (общие сведения).

Подключение пользовательской функции к исполняющей системе Open Basic производится с помощью трех методов `ob_obasic::setfun`.

Прототипы методов `setfun`:

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

Параметры методов `setfun`:

1. `name` - **Basic-имя** пользовательской функции.
2. `f` - указатель на пользовательскую функцию.
3. `dt` - тип размещения размещения в памяти возвращаемого значения (только для функций, возвращающих строку).

Пример:

К экземпляру интерпретатора подключается функция `userfun1`. Функция `userfun1` возвращает значение с плавающей точкой. Функция `userfun1` печатает сообщение "Hello world" в стандартный поток вывода и возвращает значение 3.1416. При подключении к экземпляру интерпретатора функция `userfun1` получает **Basic-имя** `TEST_FUN1`, поэтому в ***.bas-программах** функцию `userfun1` можно вызывать по имени `TEST_FUN1`.

Создание функции `userfun1` с помощью макроса `OB_DECLARE_USER_FUNCTION`.

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

Создание экземпляра интерпретатора и подключение функции `userfun1` с помощью метода `setfun`.

```
void main(){
ob_obasic onebasic;//создается экземпляр интерпретатора с именем onebasic
onebasic.setfun("TEST_FUN1",userfun1);//функция userfun1 подключается к экземпляру интерпретатора под Basic-именем TEST_FUN1
ifstream inonebasicf("EXAMPLE.BAS",ios::binary);//открывается файл EXAMPLE.BAS
onebasic.clear_project();//очищается проект
onebasic.load_project(&inonebasicf,0);//файл EXAMPLE.BAS загружается в интерпретатор
onebasic.run();//запускается *.bas-программа
}
```

Для функции, возвращающей строковый тип есть отличие в порядке подключения. Для функции, возвращающей строковый тип возможно два варианта размещения в памяти возвращаемого строкового значения:

1. Размещение возвращаемого строкового значения в куче оператором `new[]`.
2. Размещение возвращаемого строкового значения в статической памяти.

В первом случае пользовательская функция подключается с параметром `dt=OB_DELETE`. После окончания работы такой функции Open Basic освободит память оператором `delete[]`. Оператор `new[]` в пользовательской функции должен быть совместим с оператором `delete[]` в Open Basic. В пользовательской функции должен быть использован именно оператор `new[]`, а не оператор `new`, так как освобождение памяти в Open Basic производится с помощью оператора `delete[]`, а не оператора `delete`.

Во втором случае пользовательская функция подключается с параметром `dt=OB_NODELETE`. После окончания работы такой функции Open Basic не будет освобождать память оператором `delete[]`.

Пример:

В примере создается экземпляр интерпретатора, открывается файл с именем **EXAMPLE.BAS** и загружается в интерпретатор. После этого к исполняющей системе подключаются четыре функции **TEST_FUN1**, **TEST_FUN2**, **TEST_FUN3**, **TEST_FUN4**.

1. Функция **TEST_FUN1** печатает список своих параметров с плавающей точкой и возвращает значение с плавающей точкой, равное сумме своих параметров с плавающей точкой.
2. Функция **TEST_FUN2** печатает список своих целых параметров и возвращает целое значение равное сумме своих параметров с фиксированной точкой.
3. Функция **TEST_FUN3** печатает свой первый строковый параметр и возвращает строку **"This is string"**. Функция **TEST_FUN3** располагает возвращаемое значение в куче оператором **new[]**.
4. Функция **TEST_FUN4** печатает свой первый строковый параметр и возвращает строку **"This is string"**. Функция **TEST_FUN4** располагает возвращаемое значение в статической памяти.

Все функции контролируют правильность типов своих параметров. В случае отсутствия параметров нужного типа все функции печатают сообщения.

```
#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(descr[0]==0) cout<<endl<<"Not int parameters";

for(i=0;i!=descr[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 obebasic;//создается экземпляр интерпретатора с именем obebasic

obebasic.setfun("TEST_FUN1",userfun1);//подключается функция TEST_FUN1
obebasic.setfun("TEST_FUN2",userfun2);//подключается функция TEST_FUN2
obebasic.setfun("TEST_FUN3",userfun3,OB_DELETE);//подключается TEST_FUN3
obebasic.setfun("TEST_FUN4",userfun4,OB_NODELETE);//подключается TEST_FUN4

ifstream inonebasicf("EXAMPLE.BAS",ios::binary);//открывается файл EXAMPLE.BAS

obebasic.clear_project();//очищается проект
obebasic.load_project(&inonebasicf,0);//файл EXAMPLE.BAS загружается в интерпретатор

obebasic.run();//запускается *.bas-программа
}
```

При обнаружении в тексте ***.bas-программы** имени подключенной функции исполняющая система **Open Basic** производит следующие действия:

1. Вычисляет все аргументы в текущем вызове ***.bas-программы**.
2. Заполняет массивы **descrf**, **descr** и **descrc**.
3. Заполняет массивы **parf**, **pari**, **parc**.
4. Заполняет параметр **parstring**.
5. Передает управление пользовательской функции.

Пользовательская функция должна контролировать типы, количество и порядок следования параметров в текущем вызове.

4.13. Методы и перечисления класса **ob_obasic** для подключения и отключения пользовательских функций.

4.13.1. Перечисление **ob_type_del**.

Перечисление

```
enum ob_type_del(OB_DELETE,OB_NODELETE);
```

определяет тип размещения в памяти возвращаемого пользовательской функцией значения.

Перечисление **enum ob_type_del** используется в методе **setfun** чтобы подключить пользовательские функции, возвращающие тип **ob_type_char***.

При **dt=OB_DELETE** возвращаемое значение располагается в свободной памяти оператором **new[]**. После окончания работы пользовательской функции возвращаемое значение будет удалено оператором **delete[]**. Оператор **new[]** в пользовательской функции должен быть совместим с оператором **delete[]** в **Open Basic**. В пользовательской функции должен быть использован именно оператор **new[]**, а не оператор **new**, так как освобождение памяти в **Open Basic** производится с помощью оператора **delete[]**, а не оператора **delete**.

При **dt=OB_NODELETE** возвращаемое значение не будет удалено оператором **delete[]**.

Перечисление **ob_type_del** не является членом класса **ob_obasic**.

ВНИМАНИЕ: Неправильное употребление параметра **ob_type_del** ведет к краху программы.

4.13.2. Методы `ob_obasic::setfun`.

Методы

```
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);
```

подключают пользовательские функции к исполняющей системе **Open Basic**.

Параметры:

1. **name** - **Basic**-имя пользовательской функции.
2. **f** - указатель на пользовательскую функцию.
3. **dt** - тип размещения размещения в памяти возвращаемого значения (только для функций, возвращающих строку).

Если функция с именем **name** еще не подключена к исполняющей системе **Open Basic**, метод подключает функцию с именем **name** и возвращает **false**. Если функция с именем **name** уже подключена к исполняющей системе **Open Basic**, метод заменяет функцию с именем **name** на новую функцию и возвращает **true**.

Типы **ob_type_flofun**, **ob_type_intfun**, **ob_type_charfun** это указатели на функции, описанные в **ob.h**. Пользовательские функции в **Open Basic** отличаются типом возвращаемого значения.

4.13.3. Метод `ob_obasic::checkfun`.

Метод

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

проверяет наличие пользовательской функции в **Open Basic**.

Параметры:

name - **Basic**-имя пользовательской функции

Метод **checkfun** возвращает значение **true**, если функция с именем **name** уже подключена к экземпляру интерпретатора **Open Basic**. Метод **checkfun** возвращает значение **false**, если функция с именем **name** еще не подключена к экземпляру интерпретатора **Open Basic**.

4.13.4. Метод `ob_obasic::delfun`.

Метод

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

отключает пользовательскую функцию от **Open Basic**.

Параметры:

name - **Basic**-имя пользовательской функции

Метод **delfun** возвращает значение **true**, если функция успешно отключена от экземпляра интерпретатора **Open Basic**. Метод **delfun** возвращает значение **false**, если такой функции не существует в экземпляре интерпретатора **Open Basic**.

4.14. Доступ из пользовательской функции к данным *.bas-программы. Общие сведения.

Пользовательская функция может принять параметры из *.bas-программы.

Пользовательская функция может передать в *.bas-программу выходное значение.

Пользовательская функция имеет доступ к локальным и глобальным переменным и массивам *.bas-программы по чтению и записи.

Если в *.bas-программе используется переменная или массив, то пользовательская функция может считывать и записывать эту переменную и элементы массива.

Пользовательская функция имеет средства проверки наличия переменной с заданным именем в программе.

Пользовательская функция имеет средства создания переменных с заданным именем в таблице переменных *.bas-программы.

Все возможности по доступу к данным *.bas-программы реализуются с использованием методов класса **ob_obasic**.

Методы класса **ob_obasic** для доступа к локальным и глобальным переменным и локальным и глобальным массивам можно разделить на две группы.

К первой группе относятся собственно методы для доступа к данным.

В первой группе 12 методов:

1. `ob_type_ident typevar(ob_type_char* name,ob_type_istreamcommon* i=OB_NULL).`
2. `ob_type_ident typearray(ob_type_char* name,ob_type_istreamcommon* i=OB_NULL).`
3. `void createvar(ob_type_char* name,ob_type_istreamcommon* i=OB_NULL).`
4. `void createarray(ob_type_char* name,ob_type_arraydimension kr,ob_type_arraysize* as,ob_type_istreamcommon* i=OB_NULL).`
5. `ob_type_ident strlenvar(ob_type_char* name,ob_type_stringsize* len,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon* i=OB_NULL).`
6. `bool writevar(ob_type_char* name,ob_type_flo val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon* i=OB_NULL).`
7. `bool writevar(ob_type_char* name,ob_type_int val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon* i=OB_NULL).`
8. `bool writevar(ob_type_char* name,ob_type_char* val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon* i=OB_NULL).`
9. `bool readvar(ob_type_char* name,ob_type_flo* val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon* i=OB_NULL).`
10. `bool readvar(ob_type_char* name,ob_type_int* val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon* i=OB_NULL).`
11. `bool readvar(ob_type_char* name,ob_type_char* val,ob_type_arraysize* as=OB_NULL,ob_type_istreamcommon* i=OB_NULL).`
12. `const ob_type_arraysize* getarraysize(ob_type_char* name,ob_type_arraydimension* kr,ob_type_istreamcommon* i=OB_NULL).`

Ко второй группе относятся вспомогательные методы, которые обеспечивают информацией методы первой группы.

Во второй группе 2 метода:

1. `ob_type_istreamcommon* get_current_input_stream().`
2. `bool local_name_detect(ob_type_char* name).`

Параметр **name** во всех методах первой группы это имя переменной или массива **Open Basic**.

Параметр **ob_type_istreamcommon** во всех методах первой группы это указатель на поток при доступе к локальной переменной.

Правила обслуживания локальных и глобальных переменных в методах первой группы:

1. Вначале анализируется имя **name** на локальность или глобальность. Если имя **name** глобальное, то поиск осуществляется в таблицах глобальных переменных и массивов независимо от значения параметра **ob_type_istreamcommon**.
2. Если имя **name** локальное, то поиск осуществляется в таблицах локальных переменных и массивов, которые принадлежат потоку (файлу) указанному в параметре **ob_type_istreamcommon**.
3. Если имя **name** локальное, и параметр **ob_type_istreamcommon** равен нулю, то поиск всегда заканчивается неудачно.

Для правильного применения методов доступа нужно реализовать следующий алгоритм:

1. Получить указатель на текущий поток методом `get_current_input_stream()`.
2. Вызвать метод доступа с полученным потоком в качестве параметра **i**.

Такой алгоритм гарантирует, что поиск локального и глобального имени будет выполнен правильно.

Метод `local_name_detect(ob_type_char* name)` позволяет определить локальность или глобальность имени.

Если нужно выполнить поиск локального имени в потоке, который не является текущим, то указатель на поток нужно получать из внешней (по отношению к интерпретатору) процедуры.

Подробнее параметры методов доступа к переменным и массивам описаны в следующих пунктах.

4.15. Методы и перечисления класса `ob_obasic` для доступа к данным Open Basic из пользовательских функций.

Методы этого раздела запускаются обычно в пользовательских функциях, поэтому в примерах они запускаются с помощью указателя `ob_obasic* basic_ptr`.

Имя `basic_ptr` имеет указатель на экземпляр интерпретатора Open Basic во всех пользовательских функциях, если определять пользовательские функции с помощью макроса `OB_DECLARE_USER_FUNCTION`.

В общем случае методы этого раздела могут запускаться в любом месте C++ программы, в которой используется интерпретатор. Например, после останова *.bas-программы с помощью точки останова, можно распечатать значения переменных и массивов.

С помощью методов и перечислений этого раздела можно перед началом работы программы создавать и инициализировать переменные и массивы.

4.15.1. Методы и перечисления класса `ob_obasic` для определения типа переменных и массивов Open Basic.

4.15.1.1. Перечисление `ob_type_ident`.

Перечисление

```
enum ob_type_ident(OB_IDENTFLO,OB_IDENTINT,OB_IDENTSTR,OB_NOIDENT);
```

определяет тип переменной или массива.

Перечисление `ob_type_ident` используется в методах `typevar` и `typearray` чтобы определить тип переменной и массива.

`OB_IDENTFLO` означает, что переменная имеет тип с плавающей точкой.

`OB_IDENTINT` означает, что переменная имеет целый тип.

`OB_IDENTSTR` означает, что переменная имеет строковый тип.

`OB_NOIDENT` означает, что переменная не обнаружена.

Перечисление `ob_type_ident` не является членом класса `ob_obasic`.

4.15.1.2. Метод `ob_obasic::typevar`.

Метод

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

возвращает тип переменной с именем `name`.

Параметры:

1. `name` - имя Basic-переменной.
2. `i` - указатель на поток, в котором ищется переменная (для глобальных переменных указатель на поток может быть равен нулю).

возвращаемое значение:

`OB_IDENTFLO` - переменная имеет тип с плавающей точкой

`OB_IDENTINT` - переменная имеет целый тип

`OB_IDENTSTR` - переменная имеет строковый тип

`OB_NOIDENT` - переменная не обнаружена

4.15.1.3. Метод `ob_obasic::typearray`.

Метод

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

возвращает тип массива с именем `name`.

Параметры:

1. `name` - имя Basic-массива.
2. `i` - указатель на поток, в котором ищется переменная (для глобальных массивов указатель на поток может быть равен нулю).

возвращаемое значение:

`OB_IDENTFLO` - переменная имеет тип с плавающей точкой

`OB_IDENTINT` - переменная имеет целый тип

`OB_IDENTSTR` - переменная имеет строковый тип

`OB_NOIDENT` - переменная не обнаружена

Пример:

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

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

В примере определены глобальные переменные и глобальные массивы. Доступ к ним осуществляется следующим образом:

```
ob_type_ident ident;
ident=basic_ptr->typevar("a$");//после выполнения этой строки ident имеет значение OB_IDENTINT
ident=basic_ptr->typevar("b$");//после выполнения этой строки ident имеет значение OB_IDENTSTR
ident=basic_ptr->typevar("c");//после выполнения этой строки ident имеет значение OB_IDENTFLO

ident=basic_ptr->typearray("array1$");//после выполнения этой строки ident имеет значение OB_IDENTINT
ident=basic_ptr->typearray("array2$");//после выполнения этой строки ident имеет значение OB_IDENTSTR
ident=basic_ptr->typearray("array3");//после выполнения этой строки ident имеет значение OB_IDENTFLO
```

Где `basic_ptr` это первый параметр в пользовательских функциях, указатель на экземпляр интерпретатора.

В примере *.bas-программы нет переменной с именем `VAR` и массива с именем `ARR`.

Поэтому применение методов `typevar` и `typearray` даст следующие результаты:

```
ident=basic_ptr->typevar("VAR");//после выполнения этой строки ident имеет значение OB_NOIDENT
ident=basic_ptr->typearray("ARR");//после выполнения этой строки ident имеет значение OB_NOIDENT
```

Для доступа к локальным переменным и локальным массивам нужно указывать указатель на поток, в котором определена локальная переменная и локальный массив.

Пример:

```
DIM_array1$(10,10), _array2$(10,10), _array3(10,10)
_a%=10
_b$="string example"
_c=101.4
```

В примере определены локальные переменные и локальные массивы. Доступ к ним осуществляется следующим образом:

```
ob_type_istreamcommon* stream_ptr=basic_ptr->get_current_input_stream();
ob_type_ident ident;
ident=basic_ptr->typevar("_a%",stream_ptr);//после выполнения этой строки ident имеет значение OB_IDENTINT
ident=basic_ptr->typevar("_b$",stream_ptr);//после выполнения этой строки ident имеет значение OB_IDENTSTR
ident=basic_ptr->typevar("_c",stream_ptr); //после выполнения этой строки ident имеет значение OB_IDENTFLO
ident=basic_ptr->typearray("_array1%",stream_ptr);//после выполнения этой строки ident имеет значение OB_IDENTINT
ident=basic_ptr->typearray("_array2%",stream_ptr);//после выполнения этой строки ident имеет значение OB_IDENTSTR
ident=basic_ptr->typearray("_array3",stream_ptr); //после выполнения этой строки ident имеет значение OB_IDENTFLO
```

Если переменные определены не в текущем выполняемом интерпретатором потоке, то указатель на поток нельзя получить с помощью метода `get_current_input_stream`. В этом случае указатель на поток нужно передавать через внешние по отношению к интерпретатору процедуры.

4.15.2. Методы класса `ob_obasic` для определения длины строковых переменных и элементов строковых массивов Open Basic.

Метод

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

определяет тип переменной и длину переменной или элемента массива с именем `name`.

Параметры:

1. `name` - имя **Basic**-переменной или **Basic**-массива.
2. `*len` - выходной параметр. Для строковых переменных содержит длину строки без завершающего нуля. Для переменных с плавающей точкой и целых содержит `sizeof(ob_type_float)` и `sizeof(ob_type_int)` соответственно.
3. `*as` - индексы текущего элемента массива в случае если пале имя массива.
4. `i` - указатель на поток, в котором ищется переменная (для глобальных переменных и глобальных массивов указатель на поток может быть равен нулю).

Для переменных с плавающей точкой и целых результат работы метода `strlenvar` зависит от платформы, на которой транслируется интерпретатор **Open Basic** и от типов переменных в **Open Basic**. Например, для платформы **x86-32** `sizeof(float)` будет равен 4 и `sizeof(int)` тоже будет равен 4.

Метод `strlenvar` можно вызывать для **Basic**-переменных или элементов **Basic**-массивов. Если параметр `as` равен нулю, то имя `name` ищется в таблице переменных. Если параметр `as` не равен нулю, то имя `name` ищется в таблице массивов.

Метод `strlenvar` возвращает `OB_NOIDENT` в случае ошибки задания имени (например, если имя начинается не с буквы или является именем не созданного на момент вызова метода массива или переменная не существует).

При нормальном завершении метод `strlenvar` возвращает тип переменной.

Параметр `*len` для строковых переменных содержит длину строки без завершающего нуля.
Параметр `*len` для переменных с плавающей точкой содержит `sizeof(ob_type_float)`.
Параметр `*len` для переменных с фиксированной точкой содержит `sizeof(ob_type_int)`.

Пример:

Вызов метода `strlenvar` для элемента массива с указанием текущих индексов в параметре `*as`.

Если есть ***.bas**-программа:

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

после выполнения этого участка кода в пользовательских функциях, подключенных к интерпретатору, применение метода `strlenvar` даст следующие результаты:

```
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
```

Пример:

Вызов метода `strlenvar` для локальных переменных.

Если есть ***.bas**-программа:

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

после выполнения этого участка кода в пользовательских функциях, подключенных к интерпретатору применение метода `strlenvar` даст следующие результаты:

```
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
```

4.15.3. Методы класса `ob_obasic` для определения размеров массивов OpenBasic.

Метод

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

возвращает количество измерений массива и размер каждого измерения.

Параметры:

1. `name` - имя массива.

2. ***kr** - выходной параметр количество измерений массива (заполняется методом `getarraysize`).
3. **i** - указатель на поток, в котором ищется переменная (для глобальных массивов указатель на поток может быть равен нулю).

Возвращаемый параметр - массив размеров каждого измерения. Массив размеров каждого измерения создается при описании массива оператором **DIM**. Этот параметр представляет собой внутренний буфер **Open Basic** и должен только читаться.

ВНИМАНИЕ! Изменение возвращаемого массива приведет к краху программы.

Пример:

Применение метода `getarraysize` для глобального массива.

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

```
const ob_type_arraysize *sr;
ob_type_arraydimension kr;
sr=basic_ptr->getarraysize("Array_1",&kr);
```

После выполнения этого **C++** кода `getarraysize kr=3`, а указатель `sr` указывает на массив {10,11,12}.

Пример:

Применение метода `getarraysize` для локального массива.

```
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);
```

После выполнения этого **C++** кода `getarraysize kr=3`, а указатель `sr` указывает на массив {10,11,12}.

4.15.4. Методы класса `ob_obasic` для записи переменных **Open Basic**.

Методы

```
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).
```

осуществляют запись значения в переменную с указанным именем или в элемент массива с указанным именем.

Методы `readvar` переопределены для разных типов второго аргумента.

Параметры:

1. **name** - имя **Basic**-переменной или **Basic**-массива.
2. **val** - значения, записываемые в переменные или элементы массива.
3. ***as** - индексы текущего элемента массива в случае если **name** имя массива.
4. **i** - указатель на поток, в котором ищется переменная (для глобальных переменных и глобальных массивов указатель на поток может быть равен нулю).

Методы `writevar` можно вызывать для **Basic**-переменных или элементов **Basic**-массивов. Если параметр `as` равен нулю, то имя `name` ищется в таблице переменных. Если параметр `as` не равен нулю, то имя `name` ищется в таблице массивов.

Методы `writevar` записывают переменную с именем `name` в таблицу переменных и присваивают ей значение `val`. Если переменная с именем `name` не существует, то она создается. При создании переменной производится проверка на тип имени, то есть создается переменная соответствующая типу имени.

Если тип присваиваемого значения не соответствует типу имени (например при попытке создать переменную целого типа со вторым аргументом `ob_type_char* val`), то метод генерирует исключение.

Если метод `writevar` обращается к массиву который не существует, то массив не создается. Массив нужно создавать либо оператором **DIM**, либо методом `createarray`.

Методы `writevar` возвращают `false` в случае ошибки задания имени, например если оно начинается не с буквы или является именем не описанного на момент использования массива. При нормальном завершении методы возвращают `true`.

Пример:

Чтобы в пользовательской функции присвоить локальной переменной `_VAR` значение 10.4, нужно написать следующий код:

```
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";
```

Пример:

Чтобы в пользовательской функции присвоить третьему элементу локального строкового массива `_array_string$(10)` строковое значение "Hello world", нужно написать следующий код:

```
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";
```

В этом примере параметр `as` не равен нулю так как индекс элемента массива указан в не в имени `name`, а в параметре `as`. Заметьте, что локальный строковый массив `_array_string$(10)` определен в текущем входном потоке и указатель на текущий входной поток предварительно запрашивается методом `get_current_input_stream`.

4.15.5. Методы класса `ob_obasic` для чтения переменных **Open Basic**.

Методы

```
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).
```

осуществляют чтение значения из переменной с указанным именем или из элемента массива с указанным именем.

Методы `readvar` переопределены для разных типов второго аргумента.

Параметры:

1. **name** - имя **Basic**-переменной или **Basic**-массива.

2. ***val** - значение, прочитанное из переменной или элемента массива.
3. ***as** - индексы текущего элемента массива в случае если **name** имя массива.
4. **i** - указатель на поток, в котором ищется переменная (для глобальных переменных и глобальных массивов указатель на поток может быть равен нулю).

Методы **readvar** можно вызывать для **Basic-переменных** или элементов **Basic-массивов**. Если параметр **as** равен нулю, то имя **name** ищется в таблице переменных. Если параметр **as** не равен нулю, то имя **name** ищется в таблице массивов.

Методы **readvar** считывают переменную с именем **name** из таблицы переменных в указатель **val**. Если переменная не существует, то она не создается.

Методы **readvar** возвращают **false** в случае ошибки задания имени, например если оно начинается не с буквы или является именем не описанного на момент использования массива или переменная не существует. При нормальном завершении методы возвращают **true**.

Если тип переменной и параметра **val** не совпадают (например при попытке присвоить строковой переменной значение целого числа), метод генерирует исключение.

Пример:

Чтобы в пользовательской функции считать значение глобальной переменной **VARI** в **C++** переменную с именем **val1**, нужно написать следующий код:

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

Пример:

Чтобы в пользовательской функции считать значение четвертого элемента глобального массива **array_string\$(10)**, нужно написать следующий код:

```
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";
```

Буфер **pchar** должен иметь размер, достаточный для копирования элемента массива **array_string\$(4)**. Для статического буфера из примера задан максимальный размер строки в **Open Basic**, равный **ob_maxlengthstring**. Но в реальной программе длина строковой переменной может превышать **ob_maxlengthstring** из-за того, что длина суммы строк может быть больше **ob_maxlengthstring**. В реальной программе лучше пользоваться методом **strlenvar** для определения длины строковой переменной и использовать динамический буфер для хранения считанного строкового значения.

4.15.6. Методы класса **ob_obasic** для создания переменных и массивов **Open Basic**.

4.15.6.1. Методы класса **ob_obasic** для создания переменных **Open Basic**.

Метод

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

создает переменную с именем **name** в таблицах интерпретатора **Open Basic**.

Параметры:

1. **name** - имя **Basic-переменной**.
2. **i** - указатель на поток, в котором ищется переменная (для глобальных переменных указатель на поток может быть равен нулю).

Метод **createvar** создает переменную с именем **name** в таблице переменных интерпретатора. Тип переменной определяется по наличию в имени символов '%' и '\$'. Если создается локальная переменная, то она создается в таблице локальных переменных для указанного входного потока.

4.15.6.2. Методы класса **ob_obasic** для создания массивов **OpenBasic**.

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

создает массив с именем **name** в таблицах интерпретатора **Open Basic**.

Параметры:

1. **name** - имя **Basic-массива**.
2. **kr** - количество измерений массива.
3. ***as** - размеры массива по каждому измерению.
4. **i** - указатель на поток, в котором ищется переменная (для глобальных массивов указатель на поток может быть равен нулю).

Метод **createarray** создает массив с именем **name** в таблице массивов интерпретатора. Если массив с таким именем уже существует, он удаляется и создается заново с указанными параметрами.

Тип элементов создаваемого массива определяется по наличию в имени массива символов '%' и '\$'. Метод эквивалентен оператору **DIM**. Если создается локальный массив, то он создается в таблице локальных массивов для указанного входного потока.

Пример:

Чтобы из пользовательской функции создать глобальный массив целых **ARRAY1\$(5,10,20)**, нужно написать следующий код:

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

4.16. Описание других методов класса **ob_obasic**.

1. **~ob_obasic()**; - деструктор.
2. Метод **static const ob_type_char* about()** - возвращает строку информации о программе. Строка может содержать символы новой строки.
3. Метод **static const ob_type_serialnum serial_number()** - возвращает номер экземпляра программы.
4. Метод **static const ob_type_int version()** - возвращает номер текущей версии программы, умноженный на 100. Для версии 1.91 метод возвращает 191.
5. Метод **void set_max_nested_gosub(ob_type_countlist m)**; - устанавливает максимальное количество вложенных вызовов для оператора **GOSUB**. По умолчанию количество вложенных вызовов для оператора **GOSUB** равно 4096. Если происходит более 4096 вложенных вызовов для оператора **GOSUB**, то генерируется исключение. Если установить максимальное количество вложенных вызовов для оператора **GOSUB** равным нулю, то проверка на максимальное количество вложенных вызовов не производится.
6. Метод **ob_type_countlist get_max_nested_gosub()**; - возвращает установленное максимальное количество вложенных вызовов для оператора **GOSUB**.

Пример:

```
cout<<ob_obasic::about();//печатать информации о программе
cout<<ob_obasic::serial_number();//печатать серийного номера программы
cout<<ob_obasic::version();//печатать номера версии программы
```

4.17. Операторы интерпретатора **Open Basic**.

Операторы могут располагаться в одной строке и отделяться друг от друга пробелами.

Некоторые операторы, например **DATA** и **PRINT** должны быть единственными операторами в строке, так как список аргументов этих операторов ограничивается символом конца строки.

При описании операторов символы в квадратных скобках означают необязательные символы.

Интерпретатор **Open Basic** поддерживает следующие операторы:

4.17.1. Оператор PRINT.

Оператор **PRINT** выводит в выходной поток список переменных. Выходной поток прикрепляется к интерпретатору с помощью метода **setout**.

Формат оператора **PRINT**:

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

где:

N - необязательный номер строки (метка).

LIST - список элементов печати, представленных в виде констант, переменных, строковых или числовых выражений.

EXP - выражение, определяющее номер канала вывода, открытого оператором **OPEN**.

Оператор **PRINT** без списка элементов выводит символ новой строки. Если элементом списка является выражение, **Open Basic** вычисляет это выражение и печатает результат.

Пример:

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

Результат работы примера:

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

Элементы списка печати отделяются друг от друга запятыми или точками с запятой. Если элементы списка печати отделены запятыми, то производится вывод символа табуляции между элементами списка. Если элементы списка печати отделены точками с запятой, то не производится вывод никаких разделителей между элементами списка.

Если в конце списка элементов стоит запятая или точка с запятой, то после оператора **PRINT** не производится перевод строки.

Оператор **PRINT** со списком элементов должен быть единственным оператором в строке.

4.17.2. Оператор INPUT.

Оператор **INPUT** вводит из входного потока данные. Входной поток прикрепляется к интерпретатору с помощью метода **setin**.

Формат оператора **INPUT**:

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

где:

N - необязательный номер строки (метка).

VAR1, [VAR2, VAR3, ...] - список переменных.

EXP - выражение, определяющее номер канала ввода, открытого оператором **OPEN**.

Встретив оператор **INPUT** **Open Basic** вводит из входного потока данные, представленные в числовой форме и присваивает их переменным из списка. Строковые данные во входном потоке должны быть заключены в кавычки.

Пример:

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

4.17.3. Операторы FOR и NEXT.

Операторы **FOR** и **NEXT** позволяют организовать цикл.

Формат оператора **FOR**:

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

где:

N - необязательный номер строки (метка).

VAR - управляющая переменная (индекс цикла).

EXP1 - начальное значение индекса (любое числовое выражение).

EXP2 - конечное значение индекса (любое числовое выражение).

EXP3 - приращение величины индекса (любое числовое выражение, может быть положительным или отрицательным, по умолчанию равен 1).

Управляющая переменная (индекс цикла) может быть элементом массива.

Операторы **FOR** и **NEXT** используются только в паре. Оператор **FOR** определяет начало цикла, оператор **NEXT** - конец цикла.

Формат оператора **NEXT**:

```
[N] NEXT VAR1
```

где:

N - необязательный номер строки (метка).

VAR1 - управляющая переменная, использованная в операторе **FOR**.

Если начальное значение переменной цикла больше конечного значения, то цикл не выполняется. Передавать управление внутрь цикла недопустимо.

Рекомендуется в качестве переменных цикла использовать целые переменные, во избежании ошибок округления.

Циклы можно вкладывать друг в друга, внутренний цикл должен заканчиваться до внешнего.

Пример:

```
FOR i%=1 TO 3
FOR ii%=4 TO 1 STEP -2
PRINT "Work FOR operator;" ii%;" i%;" i%
NEXT ii%
NEXT i%
```

Результат работы примера:

```
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
```

В связи с тем, что **Open Basic** это интерпретатор оператор **FOR** имеет некоторые особенности реализации. А именно, при входе в цикл производится проверка условия цикла. Если условие цикла не выполняется, цикл не будет выполнен ни разу. В этом случае производится поиск соответствующего оператора **NEXT** и передача управления ему. При этом не учитываются возможные ветвления алгоритма операторами **GOTO**.

Пример:

```
GOTO 30
10
NEXT i%
GOTO 20
30 FOR i%=10 TO k% STEP 1
GOTO 10
20
STOP
END
```

Этот пример будет правильно работать, если **k%** больше 10. Но если **k%** меньше или равно 10, то произойдет аварийный останов, так как поиск оператора **NEXT** производится только в направлении от оператора **FOR** к концу файла. И в приведенном примере оператор **NEXT** не будет найден.

Рекомендуется применять операторы **FOR** и **NEXT** в соответствии с их предназначением, так, чтобы оператор **NEXT** всегда следовал за соответствующим ему оператором **FOR**.

4.17.4. Оператор GOTO.

Оператор **GOTO** вызывает непосредственный переход к указанной строке с нарушением естественного порядка выполнения операторов программы.

Формат оператора **GOTO**:

[N] **GOTO** line_number

или

[N] **GOTO** label_string

где:

N - необязательный номер строки (метка).
line_number - номер строки, к которой осуществляется переход.
label_string - метка.

Пример:

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

Результат работы примера:

```
Work GOTO operator
```

Пример:

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

Результат работы примера:

```
Work GOTO operator
```

4.17.5. Операторы GOSUB и RETURN.

Операторы **GOSUB** и **RETURN** осуществляют связь программы с подпрограммой.

Формат оператора **GOSUB**:

[N] **GOSUB** line_number

или

[N] **GOSUB** label_string

где:

N - необязательный номер строки (метка).
line_number - номер строки, к которой осуществляется переход.
label_string - метка, к которой осуществляется переход.

Встретив в программе оператор **GOSUB**, **Open Basic** передает управление строке, заданной в операторе **GOSUB**. Программа продолжает выполняться с этой строки. Когда встретится оператор **RETURN** управление передается строке, следующей за строкой с оператором **GOSUB**.

Операторы **GOSUB** и **RETURN** используются только в паре.

Формат оператора **RETURN**:

[N] **RETURN**

где:

N - необязательный номер строки (метка).

Подпрограммы можно вкладывать друг в друга.

Пример:

```
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
```

Результат работы примера:

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

Пример:

```
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
```

Результат работы примера:

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

4.17.6. Оператор LET.

Оператор **LET** присваивает значение переменной.

Формат оператора **LET**:

[N] [LET] VAR=EXP

где:

N - необязательный номер строки (метка).
LET - необязательное имя оператора.
VAR - переменная, принимающая новое значение.
EXP - выражение, определяющее новое значение.

Пример:

```
20 LET a=100.1

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

Результат работы примера:

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

4.17.7. Оператор DIM.

Оператор **DIM** резервирует место для числового или строкового массива.

Формат оператора **DIM**:

[N] DIM LIST

где:

N - необязательный номер строки (метка).
LIST - список имен массивов, разделенный запятыми.

Массивы в **Open Basic** располагаются в свободной памяти. Массивы в **Open Basic** могут быть любой размерности и объема, с учетом ограничений ОС. В связи с тем, что место под массивы в **Open Basic** резервируется оператором **C++ new[]**, существуют ограничения на максимальный размер массива, которые связаны с особенностями реализации оператора **new[]** на конкретной платформе. Например в **MS DOS** оператор **new[]** не запрашивает более 64К байт. Поэтому в **MS DOS** общий размер массива в интерпретаторе **Open Basic** не может быть больше 64К байт, при этом количество измерений также не может превышать 64К байт.

Массивы в **Open Basic** нужно описывать до первого использования при помощи оператора **DIM**.

Массивы могут быть многомерными. Максимальные размеры массива зависят от ограничений реализации оператора **new[]** в применяемой операционной системе. При создании массива оператором **DIM** элементы числовых массивов инициализируются нулями, элементы строковых массивов инициализируются пустыми строками.

В **Open Basic** индекс массива начинается с единицы.

В отличие от стандартного языка **Basic** оператор **DIM** в **Open Basic** является исполняемым. Он выполняется каждый раз, когда на него передается управление. Допускается повторное описание массива с тем же именем в операторе **DIM**. Новый массив может иметь другие размерности. Старые данные при этом теряются.

Повторное описание массива можно применять для освобождения памяти. Например, после создания большого массива оператором **DIM** можно освободить память, описав массив с тем же именем и размером в один элемент.

При описании массива в операторе **DIM** в качестве размерностей могут использоваться не только константы, но переменные целого типа.

Тип создаваемого оператором **DIM** массива определяется (как и тип переменных) по наличию в конце имени массива символов % и \$.

Пример:

```
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)
```

Результат работы примера:

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

4.17.8. Операторы STOP и END.

Операторы **STOP** и **END** используются для завершения работы программы.

Оператор **STOP** вызывает очистку всех таблиц **Open Basic** - переменных, массивов, циклов и так далее.

Оператор **END** также вызывает очистку всех таблиц **Open Basic** и дополнительно выставляет внутренний флаг останова. Сброс внутреннего флага останова производится в методе **ob_obasic::load_project**.

Операторы **STOP** и **END** не вызывают очистку таблицы пользовательских функций.

В норме все ***.bas-программы** должны завершаться выполнением операторов **STOP** и **END**.

В версии **Open Basic 1.91** оператор **STOP** является избыточным, так как оператор **END** выполняет все функции оператора **STOP**. Однако рекомендуется в ***.bas-программах** применять именно пару операторов **STOP** и **END** так как в последующих версиях **Open Basic** к оператору **STOP** планируется добавить отладочные функции.

Формат операторов **STOP** и **END**:

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

где:

N - необязательный номер строки (метка).

Пример:

```
PRINT "example STOP and REM operator"
10 STOP
20 END
```

4.17.9. Оператор REM.

Оператор **REM** вводит комментарии в программу.

Формат оператора **REM**:

```
[N] REM COMMENT
```

где:

N - необязательный номер строки (метка).

COMMENT - текст комментария.

Пример:

```
a=1
10 REM this text is comment

PRINT "a=";a

STOP
END
```

Пример:

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

GOSUB label1
GOSUB label2

STOP
END

label1:
  a=b+c
  RETURN

REM this text is comment

label2:
  b=a+c
  RETURN
```

Операторы **REM** и пустые строки в ***.bas-программе** увеличивают время загрузки и время выполнения программы и занимают место в памяти или на диске.

Если на оператор **REM** никогда не передается управление, то оператор **REM** не увеличивает время выполнения программы (но увеличивает время загрузки программы).

4.17.10. Операторы OPEN и CLOSE.

Оператор **OPEN** открывает файл. Оператор **CLOSE** закрывает файл.

Формат оператора **OPEN**:

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

где:

N - необязательный номер строки (метка).
filename - имя файла.
EXP - выражение, которое вычисляется и приводится к целому типу (обычно используют константы).
AS FILE - ключевые слова.

Оператор **OPEN FOR INPUT** открывает файл для чтения.
Оператор **OPEN FOR OUTPUT** открывает файл для записи.

Формат оператора **CLOSE**:

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

где:

N - необязательный номер строки (метка).
EXP - выражение, которое вычисляется и приводится к целому типу (обычно используют константы).

Пример:

```
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
```

4.17.11. Оператор KILL.

Оператор **KILL** удаляет файл.

Формат оператора **KILL**:

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

где:

N - необязательный номер строки (метка).
STR1, STR2, STR3 - имена файлов.

Пример:

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

4.17.12. Операторы READ, DATA и RESTORE.

Операторы **READ** и **DATA** используются для организации блока данных, который считывается интерпретатором **Open Basic** во время выполнения программы.

Формат оператора **READ**:

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

где:

N - необязательный номер строки (метка).
VAR1, VAR2, VAR3 - переменные, которым присваиваются значения из списка оператора **DATA**.

Формат оператора **DATA**:

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

где:

N - необязательный номер строки (метка).
EXP1, EXP2 - выражения, которые вычисляются и присваиваются переменной из списка **READ**. Выражения могут быть числовыми или строковыми. Обычно используют константы.

Формат оператора **RESTORE**:

```
[N] RESTORE
```

где:

N - необязательный номер строки (метка).

Перед выполнением программы **Open Basic** просматривает все операторы **DATA** в порядке их появления и создает блок данных. Каждый раз, когда в программе встречается оператор **READ**, блок данных выдает последовательно соответствующее значение для переменных этого оператора в том порядке, в котором они заданы в блоке данных.

После выполнения оператора **READ** положение последних считанных данных запоминается. Следующий оператор **READ** начинает выбирать данные с той позиции, которая была установлена предыдущим оператором **READ**.

Open Basic осуществляет повторное чтение одних и тех же данных с помощью оператора **RESTORE**. Оператор **RESTORE** устанавливает указатель чтения блока **DATA** на начальную позицию.

Оператор **DATA** должен быть единственным оператором в строке.

Пример:

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

Результат работы примера:

```
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
```

4.17.13. Оператор RANDOMIZE.

Оператор **RANDOMIZE** инициализирует случайный генератор новым значением. В качестве нового значения используется текущее системное время в секундах. Поэтому повторно оператор **RANDOMIZE** нужно применять не раньше, чем через 1 секунду после его предыдущего применения.

Формат оператора **RANDOMIZE**:

[N] RANDOMIZE

где:

N - необязательный номер строки (метка).

Оператор **RANDOMIZE** помещается перед первым использованием функции случайных чисел (функция **RND**). При выполнении функции **RND** оператор **RANDOMIZE** изменяет начальное значение случайного числа таким образом, что при последующем проходе функция **RND** дает другие числа.

Пример:

```
10 RANDOMIZE
```

4.17.14. Оператор IF.

Оператор **IF** служит для организации условных переходов. Оператор **IF** имеет три формата: строковый формат, блочный формат и блочный укороченный формат.

4.17.14.1. Строковый формат оператора IF.

Строковый формат оператора **IF**:

[N] IF REL-EXP THEN operators

где:

N - необязательный номер строки (метка).
REL-EXP - проверяемое условие. Выражение отношения может быть арифметическим или строковым.
THEN - ключевое слово.
operators - оператор или группа операторов.

Если условие **REL-EXP** истинно, то выполняются операторы в строке за ключевым словом **THEN**, если условие **REL-EXP** ложно, то выполняется оператор в следующей за оператором **IF** строке.

В строковом формате оператора **IF** сразу за ключевым словом **THEN** следуют операторы, которые нужно выполнить в случае истинности условия. Если исполняющая система **Open Basic** не встречает символ конца строки после ключевого слова **THEN**, то это считается признаком строкового формата оператора **IF**.

Строковый формат оператора **IF** используется если в случае истинности условия нужно выполнить один оператор, либо несколько операторов, которые умещаются на одной строке.

Пример:

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

Результат работы примера:

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

Пример:

```
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%
```

Результат работы примера:

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

Пример:

```
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
```

Результат работы примера:

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

4.17.14.2. Блочный формат оператора IF.

Блочный формат оператора **IF**:

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

где:

N - необязательный номер строки (метка).
REL-EXP - проверяемое условие. Выражение отношения может быть арифметическим или строковым.
THEN - ключевое слово
operators1 - оператор или группа операторов
operators2 - оператор или группа операторов

Если условие **REL-EXP** истинно, то выполняются операторы в блоке между ключевыми словами **THEN** и **ELSE**. Если условие **REL-EXP** ложно, то выполняются операторы в блоке между ключевыми словами **ELSE** и **ENDIF**.

В блочном формате оператора **IF** сразу за ключевым словом **THEN** следует символ конца строки. Если исполняющая система **Open Basic** встречает символ конца строки после ключевого слова **THEN**, то это считается признаком блочного формата оператора **IF**.

Блочный формат оператора **IF** используется если в случае истинности условия нужно несколько операторов, которые не уместятся на одной строке, либо не могут быть размещены на одной строке. Например оператор **PRINT** должен быть единственным оператором в строке, так как символ конца строки служит ограничителем списка печати.

Пример:

```
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
```

Результат работы примера:

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

4.17.14.3. Блочный укороченный формат оператора IF.

Блочный укороченный формат оператора **IF**:

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

где:

N - необязательный номер строки (метка).
REL-EXP - проверяемое условие. Выражение отношения может быть арифметическим или строковым.
THEN - ключевое слово
operators - оператор или группа операторов

Если условие **REL-EXP** истинно, то выполняются операторы в блоке между ключевыми словами **THEN** и **ENDIF**. Если условие **REL-EXP** ложно, то выполняется оператор в следующей за ключевым словом **ENDIF** строке.

Блочный укороченный формат оператора **IF** используется в тех же случаях, что и блочный формат оператора **IF**, если условие **ELSE** не нужно.

4.17.15. Оператор CHECKLOAD.

Оператор **CHECKLOAD** служит для предотвращения повторной загрузки потока методом **load_project**.

```
[N] CHECKLOAD "string"
```

где:

N - необязательный номер строки (метка).
CHECKLOAD - ключевое слово
"string" - уникальная строка в двойных или одинарных кавычках.

Во время выполнения метода **ob_obasic::load_project** исполняющая система интерпретатора создает таблицу строк для операторов **CHECKLOAD**. Если при анализе очередного оператора **CHECKLOAD** строка **"string"** уже присутствует в таблице строк для операторов **CHECKLOAD**, то метод **load_project** завершается с кодом завершения **ob_obasic::CHECKLOAD_DUPLICATE**.

Если включать оператор **CHECKLOAD** в начало каждого файла с ***.bas-программой**, то можно предотвратить повторную загрузку файлов и конфликт меток. Важно, чтобы оператор **CHECKLOAD** появился в ***.bas-программе** до первой метки.

На этапе выполнения программы методом **ob_obasic::run** оператор **CHECKLOAD** игнорируется.

Аргумент оператора **CHECKLOAD** строка **"string"** должна быть уникальной статической строкой, однозначно идентифицирующей загружаемый файл. Аргумент оператора **CHECKLOAD** не может быть строковой переменной, так как во время выполнения метода **ob_obasic::load_project** никакие переменные еще не существуют.

При централизованном управлении проектами использование оператора **CHECKLOAD** не очень актуально, так как можно перед загрузкой исключить повторяющиеся файлы из проекта. Использование оператора **CHECKLOAD** оправдано в случае реализации загрузки файлов проекта из пользовательской функции.

4.17.16. Операторы SUB, ENDSUB, EXITSUB .

В версии 1.91 операторы **SUB**, **ENDSUB**, **EXITSUB** не поддерживаются, однако соответствующие ключевые слова определены. Пользователь не должен применять в *.bas-программах имена **SUB**, **ENDSUB**, **EXITSUB** (на верхнем регистре).

4.18. Встроенные функции.

4.18.1. Функция SGN%.

Функция **SGN%** определяет знак выражения.

Формат: **SGN%(EXP)**

где: **EXP** - целое или выражение с плавающей точкой.

Функция **SGN%** возвращает +1 если **EXP**>0, -1 если **EXP**<0 и 0 если **EXP**=0. Если аргумент задан неверно (например задан строковый аргумент), то генерируется ошибка.

Функция **SGN%** возвращает целое значение.

Пример:

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

4.18.2. Функция ABS.

Функция **ABS** определяет абсолютное значение аргумента.

Формат: **ABS(EXP)**

где: **EXP** - целое или выражение с плавающей точкой.

Функция **ABS** возвращает результат с плавающей точкой, даже при целом аргументе. Если аргумент задан неверно (например задан строковый аргумент), то генерируется ошибка.

Функция **ABS** вычисляет результат по правилам функции **fabs** из **math.h**.

Пример:

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

4.18.3. Функция INT%.

Функция **INT%** определяет целую часть аргумента.

Формат: **INT%(EXP)**

где: **EXP** - выражение с плавающей точкой.

Функция **INT%** возвращает целый результат. Если аргумент задан неверно (например задан строковый аргумент), то генерируется ошибка.

Функция **INT%** вычисляет результат по правилам функции **ceil** из **math.h**.

Пример:

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

4.18.4. Функции SIN, COS, ATN, SQR, EXP, LOG и LOG10.

Функции вычисляют синус, косинус, арктангенс, квадратный корень, экспоненту, логарифм, и десятичный логарифм соответственно.

Форматы функций:

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

где: **EXP** - целое или выражение с плавающей точкой.

Функции возвращают результат с плавающей точкой. Если аргумент задан неверно (например задан строковый аргумент), то генерируется ошибка.

Функция **SIN** вычисляет результат по правилам функции **sin** из **math.h**.

Функция **COS** вычисляет результат по правилам функции **cos** из **math.h**.

Функция **ATN** вычисляет результат по правилам функции **atan** из **math.h**.

Функция **SQR** вычисляет результат по правилам функции **sqrt** из **math.h**.

Функция **EXP** вычисляет результат по правилам функции **exp** из **math.h**.

Функция **LOG** вычисляет результат по правилам функции **log** из **math.h**.

Функция **LOG10** вычисляет результат по правилам функции **log10** из **math.h**.

Пример:

```
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
```

4.18.5. Функция RND.

Функция **RND** генерирует псевдослучайное число в интервале 0-1.

Формат: **RND ()**

Функция **RND** возвращает результат с плавающей точкой. Аргументы игнорируются.

Функция **RND** взаимодействует с оператором **RANDOMIZE**.

Функция **RND** вычисляет результат по правилам функции **rand** из **stdlib.h**.

Пример:

```
PRINT "This is example RND function"
PRINT RND(),RND(),RND(),RND();
RANDOMIZE
PRINT RND(),RND(),RND(),RND();
```

4.18.6. Функция LEN%.

Функция **LEN%** определяет длину строки.

Формат: **LEN% (EXP)**

где: **EXP** - строковое выражение.

Функция возвращает целый результат. Если аргумент задан неверно (например задан целый аргумент или аргумент с плавающей точкой), то генерируется ошибка.

Строка должна заканчиваться нулем. Конечный ноль не входит в длину строки.

Пример:

```
PRINT "This is example LEN function"
string_this$="aaa bbb ccc ddd"
PRINT "len=";LEN$(string_this$)
```

4.18.7. Функции DAT\$ и CLK\$.

Функции **DAT\$** и **CLK\$** возвращают дату и время соответственно.

Форматы:

DAT\$ ()

CLK\$ ()

Функции возвращают строковый результат. Аргументы игнорируются.

Формат даты: **day-month-year**

Формат времени: **hour:min:sec**

Функции **DAT\$** и **CLK\$** используют функцию **time** из **time.h**.

Пример:

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

4.18.8. Функции D2STR\$, D2HEXSTR\$, STR2FLOAT и STR2INT%.

4.18.8.1. Функция D2STR\$.

Функция **D2STR\$** конвертирует число в строку с его десятичным представлением.

Формат: **D2STR\$ (EXP)**

где: **EXP** - целое число или число с плавающей точкой

Функция **D2STR\$** для конвертации использует поток **ostringstream** (**wostringstream** для версии с **Unicode**).

Функция возвращает строковый результат.

4.18.8.2. Функция D2HEXSTR\$.

Функция **D2HEXSTR\$** конвертирует целое число в строку с его шестнадцатеричным представлением.

Формат: **D2HEXSTR\$ (EXP)**

где: **EXP** - целое число

Функция **D2STR\$** для конвертации использует поток **ostringstream** (**wostringstream** для версии с **Unicode**).

Перед шестнадцатеричным представлением числа в строке вставляется префикс «0x».

Функция возвращает строковый результат.

4.18.8.3. Функция STR2FLOAT.

Функция **STR2FLOAT** конвертирует строку в число с плавающей точкой.

Формат: **STR2FLOAT (EXP)**

где: **EXP** - строка

Функция **STR2FLOAT** для конвертации использует поток **istringstream** (**wistringstream** для версии с **Unicode**).

Функция возвращает число с плавающей точкой. Если строка не может быть конвертирована в число, возвращается ноль.

4.18.8.4. Функция STR2INT%.

Функция **STR2INT%** конвертирует строку в целое число.

Формат: **STR2INT%(EXP)**

где: **EXP** - строка

Функция **STR2INT%** для конвертации использует поток **istringstream** (**wistringstream** для версии с **Unicode**).

Функция возвращает целое число. Если строка не может быть конвертирована в целое число, возвращается ноль.

Пример:

```
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)
```

4.19. Обработка ошибок. Класс **ob_err** и методы класса **ob_err**.

При обнаружении ошибки **Open Basic** генерирует исключение типа **ob_err** с кодом ошибки. Пользователь должен перехватывать это исключение и обрабатывать его. Для обработки исключений класс **ob_err** имеет методы **ob_err::getcode** и **ob_err::release**.

Метод

ob_type_codeerror ob_err::getcode()

не имеет аргументов и возвращает код ошибки (целое число).

Метод

void ob_err::release(ob_type_ostreamcommon* os, const ob_type_small_char* currentmessage[]);

печатает текстовое сообщение об ошибке из таблицы **currentmessage** в поток **os**.

Параметры:

1. **os** - указатель на выходной поток
2. **currentmessage[]** - таблица диагностических сообщений

Open Basic предоставляет таблицу диагностических сообщений на английском языке **ob_englmessage**. Пользователь может создать собственную таблицу диагностических сообщений на другом языке и использовать ее в качестве второго аргумента метода **release**. Таблица диагностических сообщений **ob_englmessage** расположена в файле **ob5.cpp**.

Типовая схема работы с исключениями **ob_err**:

```
int main(int argc, char * argv[]){
int ret=0;//код возврата main
ifstream fi;//входной поток
try{
ob_obasic basic_interpreter;//Создание интерпретатора
fi.open("test1.bas",ios::binary);//открытие входного потока в бинарном режиме
basic_interpreter.clear_project();
basic_interpreter.load_project(&fi,0);//загрузка *.bas-программы
ob_obasic::typeend te=basic_interpreter.run();//запуск *.bas-программы
cout<<endl<<"Stop by stop code";//печать кода останова
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";//обработка исключения
ret=e.getcode(); //получение кода ошибки
e.release(&cout,ob_englmessage); //печать текстового сообщения в cout
};//OB catch
return ret;
}
```

Более подробно обработку исключений можно посмотреть в файле консольного примера **.\example\ob.cpp**.