В.А. Лобанова
О.А. Воронина

АЛГОРИТМИЧЕСКИЕ ОСНОВЫ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ

ОРЛОВСКИЙ
ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ
имени И.С.Тургенева

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ И.С. ТУРГЕНЕВА»

В.А. Лобанова, О.А. Воронина

АЛГОРИТМИЧЕСКИЕ ОСНОВЫ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Орел 2016

УДК 621.3 (075.8)
ББК 22.193 я 73
Л68

Печатается по решению
редакционно-издательского совета
ОГУ имени И.С. Тургенева.
Протокол № 4 от 29.11.2016 г.

Рецензенты:

доктор технических наук, профессор кафедры
«Электроника, вычислительная техника и информационная безопасность»
федерального государственного бюджетного
образовательного учреждения высшего образования
«Орловский государственный университет имени И.С. Тургенева»
А.П. Фисун,

кандидат технических наук, доцент, заведующий кафедрой
«Системы информационной безопасности»
федерального государственного бюджетного
образовательного учреждения высшего профессионального образования
«Брянский государственный технический университет»
М.Ю. Рытов

Лобанова, В.А.

Л68 Алгоритмические основы и языки программирования: учебное пособие / В.А. Лобанова, О.А. Воронина. – Орел: ОГУ имени И.С. Тургенева, 2016. – 227 с.

Учебное пособие содержит изложение основ алгоритмизации задач, а также их компьютерную реализацию на языках программирования высокого уровня. Состоит из трех разделов. Информация первого раздела полезна для изучения программирования на низком уровне, а также при изучении ассемблирования и принципов работы с регистрами. Второй раздел посвящен основам алгоритмизации и программирования на языке C++. В третьем разделе изложены основы технологии работы в среде программирования на C#.

Предназначено для самостоятельной работы студентов, обучающихся по направлению 10.03.01 (090900) «Информационная безопасность». Также может быть полезно студентам других направлений, изучающим программирование и использующим его в прикладных задачах при выполнении расчетно-графических и курсовых работ по специальным дисциплинам. Пособие можно использовать при подготовке студентов по таким дисциплинам, как «Информационные технологии», «Технологии и методы программирования», «Вычислительная техника и программирование».

УДК 621.3 (075.8)
ББК 22.193 я 73

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Язык программирования – это формальная знаковая система, предназначенная для записи компьютерных программ.

Язык программирования определяет набор лексических, синтаксических и семантических правил, задающих внешний вид программы, и действия, которые выполнит исполнитель (компьютер) под её управлением.

Со времени создания первых программируемых машин человечество придумало более восьми тысяч языков программирования (включая абстрактные и нестандартные языки). Каждый год их число увеличивается. Некоторыми языками умеет пользоваться только небольшое число их собственных разработчиков, другие становятся известны миллионам людей. Профессиональные программисты иногда применяют в своей работе более десятка разнообразных языков программирования.

Языки программирования предназначены для написания компьютерных программ, которые применяются для передачи компьютеру инструкций по выполнению того или иного вычислительного процесса и организации управления отдельными устройствами. Языки программирования отличаются от естественных языков тем, что предназначены для передачи команд и данных от человека к компьютеру, в то время как естественные языки используются для общения людей. Можно обобщить определение «языков программирования»: это способ передачи команд, приказов, чёткого руководства к действию, тогда как человеческие языки служат еще и для обмена информацией. Язык программирования может использовать специальные конструкции для определения и манипулирования структурами данных и управления процессом вычислений.

Первые языки программирования возникали еще до появления современных электронных вычислительных машин: уже в XIX веке были изобретены устройства, которые можно с долей условности назвать программируемыми – к примеру, механические пианино и ткацкие станки. Для управления ими использовались наборы инструкций, которые в рамках современной классификации можно назвать предметно-ориентированными языками программирования. К началу XX века для кодирования данных и управления разнообразными механическими операциями начали применяться перфокарты. Позднее, в 1930 – 1940 годах, А. Чёрч и А. Тьюринг разработали

математические абстракции – лямбда-исчисление и машину Тьюринга соответственно – для формализации алгоритмов; первая из упомянутых абстракций сохраняет свое влияние на построение языков программирования и по сей день.

В это же время, в 1940-е годы, появились электрические цифровые компьютеры и был разработан язык «Plankalkül», который можно считать первым высокоуровневым языком программирования для ЭВМ. Язык был создан немецким инженером К. Цузе в период с 1943 по 1945 годы. Строилось программное обеспечение и для американского компьютера «Марк-1»; одна из активных участниц этого процесса, программист Грейс Хоппер, впоследствии разработала первый компилятор для языков программирования.

Программисты ЭВМ начала 1950-х годов, в особенности таких, как UNIVAC и IBM 701, при создании программ пользовались непосредственно машинным языком, который принято считать языком программирования первого поколения. Вскоре на смену такому методу программирования пришло применение языков второго поколения, также ограниченных спецификациями конкретных машин, но более простых для запоминания. Они традиционно известны под наименованием языков ассемблера и автокодов. Позднее, к концу десятилетия, языки второго поколения были усовершенствованы: в них появилась поддержка макрокоманд. Одновременно с этим начали появляться уже и языки третьего поколения, такие как Фортран, Лисп и Кобол. Языки программирования этого типа более абстрактны и универсальны, не имеют жесткой зависимости от конкретной аппаратной платформы и используемых на ней машинных команд. Обновленные версии перечисленных языков до сих пор имеют хождение в разработке программного обеспечения, и каждый из них оказал определенное влияние на последующее развитие языков программирования. Тогда же, в конце 1950-х годов, появился Алгол, также послуживший основой для ряда дальнейших разработок в этой сфере. Необходимо заметить, что на формат и применение ранних языков программирования в значительной степени влияли интерфейсные ограничения.

В период 1960-х – 1970-х годов были разработаны основные парадигмы языков программирования, используемые в настоящее время, хотя во многих аспектах этот процесс представлял собой лишь улучшение идей и концепций, заложенных еще в первых языках третьего поколения. В данный период были разработаны следующие языки программирования:

– язык APL оказал влияние на функциональное программирование и стал первым языком, поддерживавшим обработку массивов;

– язык ПЛ/1 (NPL) был разработан в 1960-х годах как объединение лучших черт Фортрана и Кобола;

– язык Симула, появившийся примерно в это же время, впервые включал поддержку объектно-ориентированного программирования. В середине 1970-х группа специалистов представила язык Smalltalk, который был уже всецело объектно-ориентированным;

– в период с 1969 по 1973 годы велась разработка языка Си, популярного и по сей день;

– в 1972 году был создан Пролог – первый язык логического программирования;

– в 1978 году в языке ML была реализована расширенная система полиморфной типизации, положившая начало типизированным языкам функционального программирования.

Каждый из этих языков породил по семейству потомков, и большинство современных языков программирования в конечном счете основано на одном из них.

Кроме того, в 1960 – 1970-х годах активно велись споры о необходимости поддержки структурного программирования в тех или иных языках. В частности, голландский специалист Э. Дейкстра выступал в печати с предложениями о полном отказе от использования инструкций GOTO во всех высокоуровневых языках. Развивались также приемы, направленные на сокращение объема программ и повышение продуктивности работы программиста и пользователя; в итоге наборы инструкций на языках четвертого поколения уже требовали существенно меньшего количества перфокарт для их записи, нежели аналогичные программы на языках третьего поколения.

В 1980-е годы наступил период, который можно условно назвать временем консолидации. Язык C++ объединил в себе черты объектно-ориентированного и системного программирования, правительство США стандартизировало язык Ада, производный от Паскаля и предназначенный для использования в бортовых системах управления военными объектами, в Японии и других странах мира осуществлялись значительные инвестиции в изучение перспектив так называемых языков пятого поколения, которые включали бы в себя конструкции логического программирования. Сообщество функциональных языков приняло в качестве стандарта ML и Лисп. В целом этот период характеризовался скорее опорой на заложенный в предыдущем десятилетии фундамент, нежели разработкой новых парадигм.

Важной тенденцией, которая наблюдалась в разработке языков программирования для крупномасштабных систем, было сосредото-

чение на применении модулей – объемных единиц организации кода. Хотя некоторые языки, такие, как ПЛ/1, уже поддерживали соответствующую функциональность, модульная система нашла свое отражение и применение также и в языках Модула-2, Оберон, Ада и ML. Часто модульные системы объединялись с конструкциями обобщенного программирования.

В 1990-х годах в связи с активным развитием Интернета распространение получили языки, позволяющие создавать сценарии для веб-страниц . Например, Perl, развившийся из скриптового инструмента для Unix-систем, и Java. Возрастала также и популярность технологий виртуализации. Эти изменения, однако, также не представляли собой фундаментальных новаций, являясь, скорее, совершенствованием уже существовавших парадигм и языков (в последнем случае – главным образом, семейства Си).

В настоящее время развитие языков программирования идет в направлении повышения безопасности и надежности, создания новых форм модульной организации кода и интеграции с базами данных.

1. СРАВНИТЕЛЬНЫЙ АНАЛИЗ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

1.1. Стандартизация языков программирования

Язык программирования может быть представлен в виде набора спецификаций, определяющих его синтаксис и семантику. Для многих широко распространённых языков программирования созданы международные стандарты. Специальные организации проводят регулярное обновление и публикацию спецификаций и формальных определений соответствующего языка. В рамках таких комитетов продолжается разработка и модернизация языков программирования и решаются вопросы о расширении или поддержке уже существующих и новых языковых конструкций.

Типы данных. Современные цифровые компьютеры являются двоичными и данные хранят в двоичном (бинарном) коде (хотя возможны реализации и в других системах счисления). Эти данные, как правило, отражают информацию из реального мира (имена, банковские счета, измерения и др.), представляющую высокоуровневые концепции. Особая система, по которой данные организуются в программе, называется системой *типов* языка программирования; разработка и изучение систем типов известна под названием теории типов. Языки можно разделить на имеющие *статическую типизацию* и *динамическую типизацию*, а также *бестиповые языки* (например, *Forth*).

Статически типизированные языки могут быть в дальнейшем подразделены на языки с *обязательной декларацией*, где каждая переменная и объявление функции имеют обязательное объявление типа, и языки с *выводимыми типами*. Иногда динамически типизированные языки называют *латентно типизированными*.

Структуры данных. Системы типов в языках высокого уровня позволяют определять сложные, составные типы, так называемые структуры данных. Как правило, структурные типы данных образуются как декартово произведение базовых (атомарных) типов и ранее определённых составных типов. Основные структуры данных (списки, очереди, хеш-таблицы, двоичные деревья и пары) часто представлены особыми синтаксическими конструкциями в языках высокого уровня. Такие данные структурируются автоматически.

Семантика языков программирования. Существует несколько подходов к определению семантики языков программирования.

Наиболее широко распространены разновидности следующих трёх: операционного, деривационного (аксиоматического) и денотационного (математического).

· При описании семантики в рамках *операционного* подхода обычно исполнение конструкций языка программирования интерпретируется с помощью некоторой воображаемой (абстрактной) ЭВМ.

· *Деривационная* семантика описывает последствия выполнения конструкций языка с помощью языка логики и задания пред- и постусловий.

· *Денотационная* семантика оперирует понятиями, типичными для математики – множества, соответствия, а также суждения, утверждения и др.

Парадигма программирования. Язык программирования строится в соответствии с той или иной базовой моделью вычислений и парадигмой программирования.

Несмотря на то, что большинство языков ориентировано на императивную модель вычислений, задаваемую фон-неймановской архитектурой ЭВМ, существуют и другие подходы. Можно упомянуть языки со стековой вычислительной моделью (Forth, Factor, Postscript и др.), а также функциональное (Лисп, Haskell, ML и др.) и логическое программирование (Пролог) и язык Рефал, основанный на модели вычислений, введённой советским математиком А. А. Марковым-младшим.

В настоящее время активно развиваются также декларативные и визуальные языки программирования, методы и средства разработки проблемно-специфичных языков.

Языки программирования могут быть реализованы как компилируемые и интерпретируемые.

Программа на компилируемом языке при помощи компилятора (особой программы) преобразуется (компилируется) в машинный код (набор инструкций) для данного типа процессора и далее собирается в исполнимый модуль, который может быть запущен на исполнение как отдельная программа. Другими словами, компилятор переводит исходный текст программы с языка программирования высокого уровня в двоичные коды инструкций процессора.

Если программа написана на интерпретируемом языке, то интерпретатор непосредственно выполняет (интерпретирует) ис-

ходный текст без предварительного перевода. При этом программа остаётся на исходном языке и не может быть запущена без интерпретатора. Процессор компьютера в этой связи можно назвать интерпретатором для машинного кода.

Разделение на компилируемые и интерпретируемые языки является условным. Так, для любого традиционно компилируемого языка, как, например, Паскаль, можно написать интерпретатор. Кроме того, большинство современных «чистых» интерпретаторов не исполняют конструкции языка непосредственно, а компилируют их в некоторое высокоуровневое промежуточное представление (например, с разыменованием переменных и раскрытием макросов).

Для любого интерпретируемого языка можно создать компилятор – например, язык Лисп, изначально интерпретируемый, может компилироваться без каких бы то ни было ограничений. Создаваемый во время исполнения программы код может так же динамически компилироваться во время исполнения.

Как правило, скомпилированные программы выполняются быстрее и не требуют для выполнения дополнительных программ, так как уже переведены на машинный язык. Вместе с тем, при каждом изменении текста программы требуется её перекомпиляция, что замедляет процесс разработки. Кроме того, скомпилированная программа может выполняться только на том же типе компьютеров и, как правило, под той же операционной системой, на которую был рассчитан компилятор. Чтобы создать исполняемый файл для машины другого типа, требуется новая компиляция.

Интерпретируемые языки обладают некоторыми специфическими дополнительными возможностями, кроме того, программы на них можно запускать сразу же после изменения, что облегчает разработку. Программа на интерпретируемом языке может быть зачастую запущена на разных типах машин и операционных систем без дополнительных усилий.

Однако интерпретируемые программы выполняются заметно медленнее, чем компилируемые, кроме того, они не могут выполняться без программы-интерпретатора.

Некоторые языки, например, Java и C#, находятся между компилируемыми и интерпретируемыми. А именно программа компилируется не в машинный язык, а в машинно-независимый код низкого уровня, байт-код. Далее байт-код выполняется виртуальной машиной. Для выполнения байт-кода обычно используется интерпретация, хотя

отдельные его части для ускорения работы программы могут быть транслированы в машинный код непосредственно во время выполнения программы по технологии компиляции «на лету» (Just-in-time compilation, JIT). Для Java байт-код исполняется виртуальной машиной Java (Java Virtual Machine, JVM), для C# – Common Language Runtime.

Подобный подход в некотором смысле позволяет использовать плюсы как интерпретаторов, так и компиляторов. Следует упомянуть, что есть языки, имеющие и интерпретатор, и компилятор (Форт).

Языки программирования низкого уровня. Первые компьютеры приходилось программировать двоичными машинными кодами. Однако программировать таким образом – довольно трудоемкая и тяжелая задача. Для упрощения этой задачи начали появляться языки программирования низкого уровня, которые позволяли задавать машинные команды в понятном для человека виде. Для преобразования их в двоичный код были созданы специальные программы – трансляторы.

Трансляторы делятся на компиляторы и интерпретаторы. Компиляторы превращают текст программы в машинный код, который можно сохранить и после этого использовать уже без компилятора (примером являются исполняемые файлы с расширением *.exe) . Интерпретаторы превращают часть программы в машинный код, выполняют его и после этого переходят к следующей части. При этом каждый раз при выполнении программы используется интерпретатор.

Примером языка низкого уровня является *ассемблер*. Языки низкого уровня ориентированы на конкретный тип процессора и учитывают его особенности, поэтому для переноса программы на ассемблере на другую аппаратную платформу ее нужно почти полностью переписать. Определенные различия есть и в синтаксисе программ под разные компиляторы. Правда, центральные процессоры для компьютеров фирм «AMD» и «Intel» практически совместимы и различаются лишь некоторыми специфическими командами. А вот специализированные процессоры для других устройств, например, видеокарт и телефонов, имеют существенные различия.

Языки низкого уровня, как правило, используют для написания небольших системных программ, драйверов устройств, модулей стыков с нестандартным оборудованием, а также для программирования специализированных микропроцессоров, когда важнейшими требованиями являются компактность, быстродействие и возможность прямого доступа к аппаратным ресурсам.

Язык низкого уровня – ассемблер – широко применяется до сих пор.

Языки программирования высокого уровня. Особенности конкретных компьютерных архитектур в таких языках не учитываются, поэтому созданные приложения легко переносятся с компьютера на компьютер. В большинстве случаев достаточно просто перекомпилировать программу под определенную компьютерную архитектурную и операционную систему. Разрабатывать программы на таких языках гораздо проще и ошибок допускается меньше. Значительно сокращается время разработки программы, что особенно важно при работе над большими программными проектами.

Сейчас в среде разработчиков считается, что языки программирования, которые имеют прямой доступ к памяти и регистрам или имеют ассемблерные вставки, нужно считать языками программирования с низким уровнем абстракции. Поэтому большинство языков, известных как языки высокого уровня до 2000 года, сейчас уже таковыми не являются.

Недостатком некоторых языков высокого уровня является большой размер программ в сравнении с программами на языках низкого уровня. С другой стороны, для алгоритмически и структурно сложных программ при использовании суперкомпиляции преимущество может быть на стороне языков высокого уровня. Сам текст программ на языке высокого уровня меньше, однако если взять в байтах, то код, изначально написанный на ассемблере, будет более компактным. Поэтому языки высокого уровня используются в основном для разработки программного обеспечения компьютеров и устройств, которые имеют большой объем памяти. А разные подвиды ассемблера применяются для программирования других устройств, где критичным является размер программы.

Современные языки программирования рассчитаны на использование ASCII, то есть доступность всех *графических* символов ASCII является необходимым и достаточным условием для записи любых конструкций языка. *Управляющие* символы ASCII используются ограниченно: допускаются только возврат каретки CR, перевод строки LF и горизонтальная табуляция HT (иногда также вертикальная табуляция VT и переход к следующей странице FF).

Используемые символы. Ранние языки, возникшие в эпоху 6-битных символов, использовали более ограниченный набор. Например, алфавит Фортрана включает 49 символов (включая пробел): A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 = + - * / () . , $ ' :

Заметным исключением является язык APL, в котором используется много специальных символов.

Использование символов за пределами ASCII (например, символов KOI8-R или символов Юникода) зависит от реализации: иногда они разрешаются только в комментариях и символьных/строковых константах, а иногда и в идентификаторах. В СССР существовали языки, где все ключевые слова писались русскими буквами, но большую популярность подобные языки не завоевали (исключение составляет Встроенный язык программирования 1С: Предприятие).

Расширение набора используемых символов сдерживается тем, что многие проекты по разработке программного обеспечения являются международными. Очень сложно было бы работать с кодом, где имена одних переменных записаны русскими буквами, других - арабскими, а третьих - китайскими иероглифами. Вместе с тем, для работы с текстовыми данными языки программирования нового поколения (Delphi 2006, C#, Java) поддерживают Unicode.

Математически обоснованные языки программирования. Ряд известных авторов выделяют в особую категорию *«языки, наследованные от математики»* (англ. *mathematically-derived languages)*. Алан Кэй также отделяет языки, являющиеся «стилем во плоти» *(crystalization of style),* от прочих языков, являющихся «склеиванием возможностей» *(agglutination of features)*.

Это языки, семантика которых является *непосредственным* воплощением некой математической модели, незначительно адаптированной (без нарушения целостности) для того, чтобы быть более практичным языком для разработки реальных программ. Лишь некоторые языки попадают под эту категорию, большинство же языков проектируются приоритетно - исходя из возможности эффективной трансляции в машину Тьюринга, и имеют лишь некое *подмножество* в своём составе, воплощающее ту или иную математическую модель - от арифметики до средств параллелизма (например, Occam-π - это Occam, дополненный набором конструкций, воплощающих ^--исчисление).

Примеры математически обоснованных языков и воплощаемых ими математических моделей:

- Agda - интуиционистская теория типов[еп] Мартин - Лёфа.

. APL и его потомки (J, **K**) - оригинальная семантика, не имеющая названия, воплощающая нотацию Айверсона для исчисления массивов (часто встречается термин *«array languages»)*.

• Coq - исчисление индуктивных конструкций.

• Erlang - исчисление процессов (первоначально в форме модели акторов, позже также построено обоснование на тг-исчислении).

. Forth - стековая машина и конкатенативный язык программирования.

• Haskell - теория категорий (включая «декартово замкнутую категорию», воплощающую лямбда-исчисление; категорию монад для моделирования побочных эффектов; расширение системы типов Хиндли - Милнера; систему родов; и др.).

. Joy - композиция функций и гомоморфизм (иначе говоря, чистый конкатенативный язык программирования, и, как следствие, чистый функциональный).

. Lisp - лямбда-исчисление Чёрча (в том числе язык S-выражений, воплощающий нотацию пар чёрча).

. Scheme - «облагороженный» диалект Лиспа (сильнее типизированный, в большей степени гомознаковый, ограничивающийся гигиеническими макроопределениями и соблюдающий числовую башню), дополненный нотацией продолжений.

. ML - типизированное лямбда-исчисление, то есть лямбда-исчисление, дополненное системой типов Хиндли - Милнера.

• Prolog - исчисление предикатов.

• Mercury - исчисление предикатов, дополненное системой типов Хиндли - Милнера.

. Smalltalk - теория множеств (с соблюдением числовой башни).

• SQL - исчисление кортежей (вариант реляционного исчисления, в свою очередь основанного на исчислении предикатов первого порядка).

• SGML и его потомки (HTML, XML) - нотация деревьев (важный случай графов).

• Unlambda - комбинаторная логика.

. Регулярные выражения.

• Рефал - оригинальная семантика Турчина, носящая название «Рефал-машины» или «Рефал-автомата», созданная на основе нормального алгоритма Маркова, воплощающая композицию теории автоматов, сопоставления с образцом и переписывания термов.

Наличие математического обоснования для языка может гарантировать (или, как минимум, обещать с очень высокой вероятностью) некоторые или все из нижеследующих положительных свойств:

. *Существенное повышение стабильности программ.* В одних случаях - за счёт упрощения формальной верификации программ, вплоть до построения доказательства корректности для самого языка

(см. типобезопасность – примерами служат Standard ML и Haskell) и даже получения языка, который сам является системой доказательства (Coq, Agda). В других случаях – за счёт быстрого обнаружения ошибок на первых же пробных запусках программ (Forth и регулярные выражения).

· *Обеспечение потенциально более высокой эффективности программ*. Даже если семантика языка далека от архитектуры целевой платформы компиляции, к нему могут быть применимы формальные методики глобального анализа программ (хотя трудоёмкость написания даже тривиального транслятора может оказаться выше). Например, для языков Scheme и Standard ML существуют развитые глобально-оптимизирующие компиляторы (вплоть до суперкомпиляторов), результат работы которых может уверенно конкурировать по скорости с языком низкого уровня Си и даже опережать последний (хотя ресурсоёмкость работы самих компиляторов оказывается значительно выше). Одна из самых быстрых СУБД – KDB – написана на языке K. Язык Scala (унаследовавший математику от ML) обеспечивает на платформе «JVM» более высокую скорость, чем «родной» для неё язык Java. С другой стороны, Forth имеет репутацию одного из самых нетребовательных к ресурсам языков (менее требователен, чем Си) и используется для разработки приложений реального времени под самые маломощные ЭВМ; кроме того, транслятор Форта является одним из наименее трудоёмких в реализации на ассемблере.

· *Заранее известен* (неограниченный или, наоборот, чётко очерченный) *предел роста сложности программных компонентов, систем и комплексов, которые можно выразить средствами этого языка с сохранением показателей качества*. Языки, не имеющие математического обоснования (а именно такие наиболее часто применяются в мейнстриме: C++, Java, C#, Delphi и др.), на практике ограничивают реализуемую функциональность и/или снижают качество по мере усложнения системы, так как им присущи экспоненциальные кривые роста сложности, как касательно работы одного отдельно взятого человека, так и касательно сложности управления проектом в целом. *Прогнозируемая* сложность системы приводит либо к поэтапной декомпозиции проекта на множество более мелких задач, каждая из которых решается соответствующим языком, либо к языково-ориентированному программированию для случая, когда адресуемой языком задачей является как раз описание семантик и/или символьные вычисления (Lisp, ML, Haskell, Рефал, Регулярные выра-

жения). Языки с неограниченным пределом роста сложности программ нередко относят к метаязыкам (что в непосредственном толковании термина неверно, но на практике приемлимо, так как всякий мини-язык, выбранный для решения некоторой подзадачи в составе общей задачи, может быть представлен в виде синтаксического и семантического подмножества данного языка, не требуя трансляции).

· *Удобство для человека при решении задач, на которые этот язык ориентирован по своей природе.* Данное свойство в некоторой степени также способно (косвенно) повлиять на повышение стабильности результирующих программ за счёт повышения вероятности обнаружения ошибок в исходном коде и снижения дублирования кода.

Следует иметь в виду, что языки, наследованные от «наследованных от математики», уже не обязательно будут обладать перечисленными выше свойствами. Например, язык Python соединяет в себе несколько упомянутых моделей, но для их совмещения не существует обоснования, поэтому он не может считаться «наследованным от математики», и, как следствие, ему присуще лишь последнее из указанных свойств.

Методология программирования – это совокупность определённых способов написания, отладки и сопровождения программ. Первая наиболее известная и распространенная методология программирования получила название « структурного программирования».

Появление структурного программирования связано с именами Эдсгера Дейкстры и Чарльза Хоара. Начиная с 1960-х годов стали появляться языки структурного программирования. Первым из них был Алгол-60, разработанный Дейкстрой, затем был создан Паскаль. Другие первоначально «неструктурные» языки стали также приобретать «структурные свойства» (Турбо Бейсик, Фортран-77 и пр.). Структурное программирование до настоящего времени остается важнейшей методологией программирования. Соблюдение его принципов позволяет программисту составлять ясные, безошибочные, надёжные программы.

В 1990-х годах, с развитием объектно-ориентированной парадигмы программирования, а также средств графического интерфейса на персональных компьютерах, возникает новая технология программирования – визуальное программирование. Визуальная технология программирования позволяет программисту легко и быстро строить наглядный графический интерфейс для своих программ на основе стандартного набора шаблонов, графически отображаемых на экране объектов.

1.2. Ревизия системных ресурсов

После загрузки одной из первых задач является проверка, куда мы попали: на каком типе IBM PC запущена задача?... под какой версией (напр., MS DOS?), ... сколько имеется памяти?... все ли необходимое оборудование присутствует? Имеется три способа получения этой информации. Наименее элегантный способ – спросить об этом у пользователя (но знает ли он ответы?). Намного лучше получить всю доступную информацию из установки переключателей на системной плате. Но эта установка не всегда соответствует реальности. Поэтому лучше всего использовать третью возможность – получить прямой доступ к требуемому оборудованию или прочитать нужную информацию из области данных BIOS. Поскольку установка переключателей может служить отправной точкой для получения требуемой информации, то этот раздел начинается с обсуждения микросхемы, содержащей эту информацию – микросхемы интерфейса с периферией 8255.

Программа может получить доступ к оборудованию только двумя способами. Она может обратиться к любому из портов ввода/вывода, соответствующему присоединенному оборудованию (обычно бывает занята лишь малая доля из 65535 возможных адресов портов), либо к любому из более чем миллиону адресов оперативной памяти.

Доступ к микросхеме интерфейса с периферией 8255. Микросхема интерфейса с периферией Intel 8255 – лучшее место, с которого надо начинать, чтобы получить информацию об имеющемся оборудовании. Эта микросхема предназначена для многих целей. Она сообщает об установке переключателей на системной плате, принимает для компьютера ввод с клавиатуры, управляет рядом периферийных устройств, включая микросхему таймера 8253. Из машин семейства IBM PC только AT не использует микросхему 8255; он хранит информацию об оборудовании вместе с часами реального времени в специальной микросхеме с независимым питанием. Однако AT использует те же адреса портов, что и 8255, для работы с клавиатурой и управления микросхемой таймера.

Микросхема 8255 имеет три однобайтных регистра, называемых от порта A до порта C. Адреса этих портов от 60H до 62H сответственно. Все три порта можно читать, но писать можно только в порт B. Для PC установка бита 7 порта B в 1 изменяет информацию, со-

держащуюся в порте А. Аналогично для PC установка бита 2 определяет содержимое четырех младших битов порта С, а установка бита 3 делает то же самое для XT. Содержимое этих регистров следующее:

Порт А (60H)

когда в порте В бит 7=0

биты 0-7 PC,XT,PCjr,AT: 8-битные скан-коды с клавиатуры

когда в порте В бит 7=1 для PC

бит 0 PC: 0 = нет накопителей на дискетах

1 PC: не используется

2-3 PC: число банков памяти на системной плате

4-5 PC: тип дисплея (11 = монохромный,
10 = цветной 80*25, 01 = цветной 40*25)

6-7 PC: число накопителей на дискетах

Порт В (61H)

бит 0 PC,XT,PCjr: управляет каналом 2 таймера 8253

1 PC,XT,PCjr: вывод на динамик

2 PC: выбор содержимого порта С
PCjr: 1 = символьный режим, 0 = графический

3 PC,PCjr: 1 = кассетный мотор выключен
XT: выбор содержимого порта С

4 PC,XT: 0 = разрешение ОЗУ
PCjr: 1 = запрет динамика и мотора кассеты

5 PC,XT: 0 = разрешение ошибок щелей расширения

6 PC,XT: 1 = разрешение часов клавиатуры

5-6 PCjr: выбор динамика (00 = 8253, 01 = кассета,
10 = ввод/вывод, 11 = микросхема 76496)

7 PC: выбор содержимого порта А
PC,XT: подтверждение клавиатуры

Порт С (62H)

когда в порте В бит 2=1 для PC или бит 3=1 для XT

биты 0-3 PC: нижняя половина переключателя 2 конфигурации (ОЗУ на плате расширения)

0 PCjr: 1 = введенный символ потерян

1 XT: 1 = есть мат. сопроцессор
PCjr: есть карта модема

2 PCjr: есть карта НГМД

2-3 XT: число банков памяти на системной плате

3 PCjr: 0 = 128K памяти
4 PC,PCjr: ввод с кассеты
XT: не используется
5 PC,XT,PCjr: выход канала 2 8253
6 PC,XT: 1 = проверка ошибок щелей расширения
PCjr: 1 = данные с клавиатуры
7 PC,XT: 1 = контроль ошибок четности
PCjr: 0 = кабель клавиатуры подсоединен

когда в порте B бит 2=0 для PC или бит 3=0 для XT

биты 0-3 PC: верхняя половина переключателя 2 конфигурации (не используется)
0-1 XT: тип дисплея (11 = монохромный,
10 = цветной 80*25, 01 = цветной 40*25)
2-3 XT: число накопителей НГМД (00 = 1 и т.д.)
4-7 PC,XT: то же, что и с установленными битами

Отметим, что 0 в одном из битов регистра соответствует установке переключателя "off".

AT хранит информацию о конфигурации в микросхеме MC146818 фирмы «Motorola» вместе с часами реального времени. Он вовсе не имеет микросхемы 8255, хотя для управления микросхемой таймера и приема данных с клавиатуры используются те же самые адреса портов. Микросхема имеет 64 регистра, пронумерованных от 00 до 3FH.

Для чтения регистра нужно сначала послать его номер в порт с адресом 70H, а затем прочитать его через порт 71H. Различные параметры конфигурации обсуждаются на последующих страницах.

Приведем здесь только краткую сводку:

Номер регистра	Использование
10H	тип накопителя НГМД
12H	тип накопителя фиксированного диска
14H	периферия
15H	память на системной плате (младший байт)
16H	память на системной плате (старший байт)
17H	общая память (младший байт)
18H	общая память (старший байт)
30H	память сверх 1 мегабайта (младший байт)
31H	память сверх 1 мегабайта (старший байт)

Высокий уровень

Имеется множество примеров доступа к этим портам. Ниже приводится программа на Бейсике, устанавливающая число дисковых накопителей, присоединенных к IBM PC. Прежде чем прочитать два старших бита порта A, бит 7 порта B должен быть установлен в 1. Существенно, что пользователь должен вернуть значение этого бита назад в 0 перед дальнейшей работой, иначе клавиатура будет заперта и для восстановления работоспособности машины ему придется выключить ее. Бейсик не позволяет двоичное представление чисел, что затрудняет работу с цепочками битов. Простая подпрограмма может заменить любое целое вплоть до 255 (максимальное значение, которое может принимать номер порта) на восьмисимвольную двоичную строку. После этого строковая функция MID$ позволяет вырезать нужные биты для анализа.

```
100 A = INP(&H61)          'получает значение из порта B
110 A = A OR 128           'устанавливает бит 7
120 OUT &H61,A             'посылает байт назад в порт B
130 B = INP(&H60)          'получает значение из порта A
140 A = A AND 128          'сбрасывает бит 7
150 OUT &H61,A             'восстанавливает значение порта B
160 GOSUB 1000             'преобразует в двоичную строку
170 NUMDISK$ = RIGHT$(B$,1)  'получает нулевой бит
180 IF D$ = 1 THEN NUMDISK = 0: GOTO 230 'нет дисков
190 C$ = LEFT$(B$,2)       'берет два старших бита строки
200 TALLEY = 0             'переменная для числа дисков
210 IF RIGHT$(C$,1) = " 1 " THEN TALLEY = 2 'берет старший бит
220 IF LEFT$(C$,1) = " 1 " THEN TALLEY = TALLEY + 1 'и младший
230 TALLEY = TALLEY + 1    'счет начинается с 1, а не с 0
                  'теперь имеется число накопителей
1000 "'Подпрограмма преобразования байта в двоичную строку
1010 B$ = ""               'заводит строку
1020 FOR N = 7 TO 0 STEP -1  'проверка очередной степени 2
1030 Z = B - 2^N           '
1040 IF Z >= 0 THEN B = Z: B$ = B$+"1" ELSE B$ = B$+"0"
1050 NEXT                  'повторяет для каждого бита
1060 RETURN                'все закончено
```

Низкий уровень

Ассемблерная программа получает число имеющихся дисковых накопителей тем же способом, что и в вышеприведенном примере,

но более просто. Напоминаем, что нельзя забывать о восстановлении первоначального значения в порте B.

```
IN   AL,61H          ; получает значение из порта B
OR   AL,10000000B    ; устанавливает бит 7 в 1
OUT  61H,AL          ; заменяем байт
IN   AL,60H          ; получает значение из порта A
MOV  CL,6            ; подготовка для сдвига AL
SHR  AL,CL           ; сдвигает 2 старших бита на 6 позиций
INC  AL              ; начинает счет с 1, а не с 0
MOV  NUM_DRIVES,AL   ; получает число накопителей
IN   AL,61H          ; подготовка к восстановлению порта B
AND  AL,01111111B    ; сбрасывает бит 7
OUT  61H,AL          ; восстанавливает байт
```

Определение типа IBM PC

Имеются проблемы совместимости между различными типами IBM PC. Для того чтобы программа могла работать на любом из IBM PC, используя все его возможности, необходимо, чтобы она могла определить тип машины, в которую она загружена. Эта информация содержится во втором с конца байте памяти по адресу FFFFE в ROM-BIOS, с использованием следующих ключевых чисел:

Компьютер	Код
PC	FF
XT	FE
PCjr	FD
AT	FC

Высокий уровень

В Бейсике надо просто использовать PEEK для чтения значения:

```
100 DEF SEG = &HF000       'указывает на верхние 64K памяти
110 X = PEEK(&HFFFE)       'читает второй с конца байт
120 IF X = &HFD THEN ...   '... тогда это PCjr
```

Низкий уровень

В языке ассемблера:

```
; --- Определение типа компьютера:
  MOV  AX,0F000H          ; указывает ES на ПЗУ
```

```
MOV  ES,AX              ;
MOV  AL,ES:[0FFFEH]     ; получает байт
CMP  AL,0FDH            ; это PCjr?
JE   INITIALIZE_JR      ; переходит на инициализацию
```

Определение версии MS DOS

По мере развития MS DOS к ней добавлялись новые возможности, многие из которых существенно облегчают написание определенных частей программы по сравнению с предыдущими версиями. Чтобы иметь гарантию того, что программа будет работать с любой версией MS DOS, она должна использовать только функции, доступные в MS DOS 1.0. В системе предусмотрено прерывание, возвращающее номер версии MS DOS. Это число может использоваться для проверки выполнимости программы пользователя. Минимально, программа может при старте выдавать сообщение об ошибке, сообщая что ей нужна другая версия MS DOS.

Средний уровень

Функция 30H прерывания 21H возвращает номер версии MS DOS.

Старший номер версии (2 из 2.10) возвращается в AL, а младший номер версии (10 из 2.10) возвращается в AH. AL может содержать 0, что указывает на версию MS DOS – меньшую, чем 2.0. Это прерывание меняет содержимое регистров BX и CX, в которые возвращается значение 0.

```
; --- Определение версии MS DOS:
  MOV   AH,30H          ; номер функции получения версии
  INT   21H             ; получить номер версии
  CMP   AL,2            ; проверка на версию 2.x
  JL    WRONG_DOS       ; если меньше 2, то выдать сообщение
```

Определение числа и типов адаптеров дисплея

Программе может оказаться необходима информация о том, будет ли она работать в системе с монохромным адаптером, с цветной графической картой или с EGA, а также о наличии второго адаптера.

Но как передать управление от одного адаптера к другому? Байт статуса оборудования, хранящийся в области данных ROM-BIOS,

по адресу 0040:0010 сообщает установку переключателя 1, который показывает, какая из карт активна. В принципе, байты должны иметь значение 11 для монохромной карты, 10 – для цветной карты 80*25, 01 – для цветной карты 40*25 и 00 для EGA.

Однако при наличии EGA он может установить биты отличными от 00, в зависимости от установки его собственных переключателей. Поэтому пользователь должен сначала другими средствами установить наличие EGA, а затем, если его нет, то по данным BIOS определить, является ли активным цветной или монохромный адаптер. Для проверки наличия EGA надо прочитать байт по адресу 0040:0087. Если он равен 0, то EGA отсутствует. Если этот байт ненулевой, то если бит 3 равен 0, то EGA является активным адаптером, иначе (равен 1) активен второй адаптер.

Если активным адаптером является EGA, то проверка наличия монохромного или цветного адаптера осуществляется записью значения в регистр адреса курсора микросхемы 6845, последующего чтения значения и проверки их на совпадение. Для монохромной карты необходимо послать 0FH в порт 3B4H, чтобы указать на регистр курсора, а затем прочитать и записать адрес курсора через порт 3B5H. Соответствующие порты для цветной карты – это 3D4H и 3D5H. Когда карта отсутствует, то порт возвращает значение 0FFH; но поскольку это значение может содержаться в регистре, то простой проверки на это значение недостаточно.

При наличии EGA имеются два добавочных вопроса, на которые могут потребоваться ответы: сколько имеется памяти на его карте и какой тип монитора подсоединен? Для определения типа дисплея необходимо проверить бит 1 по адресу 0040:0087; когда он установлен, то подсоединен монохромный дисплей, а когда он равен нулю – цветной. Если программа использует цветной графический режим с 350 строками, то надо также определить, присоединен ли дисплей IRGB или R'G'B'RGB, где последняя аббревиатура соответствует улучшеному цветному дисплею IBM. Это определяется установкой четырех переключателей на карте EGA. Установка этих переключателей возвращается в CL при обращении к функции 12H прерывания 10H.

Цепочка четырех младших битов должна быть 0110 для улучшенного цветного дисплея. Та же самая функция сообщает и наличие памяти на карте EGA. Она возвращает BL, содержащий 0 для 64K, 1 – для 128, 2 – для 192 и 3 – для полных 256K памяти дисплея.

Высокий уровень

Приведенные фрагменты кода позволяют определить тип текущего монитора и режим его работы, а также типы видеоадаптеров, имеющихся в машине:

```
100 '"определение активного адаптера
110 DEF SEG = &H40          'указывает на область данных BIOS
120 X = PEEK(&H87)          'проверка на наличие EGA
130 IF X = 0 THEN 200       'EGA отсутствует, продолжает дальше
140 IF X AND 8 = 0 THEN... 'активный монитор EGA
 .
 .
200 X = PEEK(&H10)          'читает байт статуса оборудования
210 Y = X AND 48            'выделяет биты 4 и 5
220 IF Y = 48 THEN ...      '... тогда монохромный (00110000)
230 IF Y = 32 THEN ...      '... тогда цветной 80*25 (00100000)
240 IF Y = 16 THEN ...      '... тогда цветной 40*25 (00010000)
```

Следующий пример проверяет наличие монохромной карты, когда активной является карта EGA или цветная. Тот же пример можно использовать для проверки наличия цветной карты, если использовать адреса портов &H3D4 и &H3D5.

```
100 '"проверка наличия монохромной карты
110 OUT &H3B4,&HF           'адрес регистра курсора
120 X = INP(&H3B5)          'чтение и сохранение значения
130 OUT &H3B5,100           'посылает в регистр любое значение
140 IF INP(&H3B5)<>100 THEN... 'если карта есть – вернется то же
150 OUT &H3B5,X             'восстанавливает значение регистра
```

Низкий уровень

Приведенные примеры соответствуют примерам на Бейсике.

```
; --- Определение активного адаптера:
  MOV   AX,40H        ; указывает ES на область данных BIOS
  MOV   ES,AX         ;
  MOV   AL,ES:[87H]   ; проверяет наличие EGA
  CMP   AL,0          ;
```

```
  JE    NO_EGA       ; если 0040:0087 = 0, то EGA нет
  TEST  AL,00001000B  ; EGA есть, проверяет бит 3
  JNZ   EGA_NOT_ACTIVE; если бит 3=1, то EGA неактивен
   .
   .
EGA_NOT_ACTIVE:
  MOV   AL,ES:[10H]   ; проверяет байт статуса дисплея
  AND   AL,00110000B  ; выделяет биты 4 и 5
  CMP   AL,48         ; это монохромная карта?
  JE    MONOCHROME    ; переход, если да
```

Предполагая наличие монохромной карты, проверим, установлена ли цветная карта (неактивная):

```
; --- Установлена ли неактивная цветная карта?
   MOV   DX,3D4H       ; указывает на регистр адреса 6845
   MOV   AL,0FH        ; запрашивает регистр курсора
   OUT   DX,AL         ; указывает на регистр
   INC   DX           ; указывает на регистр данных
   IN    AL,DX         ; получает текущее значение
   XCNG  AH,AL         ; сохраняет значение
   MOV   AL,100        ; тестовое значение 100
   OUT   DX,AL         ; посылает его
   IN    AL,DX        ; считывает его снова
   CMP   AL,100       ; сравнивает значения
   JNE   NO_CARD       ; переход, если нет карты
   XCNG  AH,AL         ; иначе – есть цветная карта
   OUT   DX,AL         ; тогда восстанавливает значение
```

Определение числа и типа дисковых накопителей

На всех машинах, кроме AT (который будет обсуждаться ниже), регистры микросхемы 8255 интерфейса с периферией содержат информацию о том, сколько НГМД имеет машина. Информация, определяющая тип диска, содержится в таблице размещения файлов (FAT) диска, которая следит за использованием дискового пространства. Первый байт FAT содержит один из следующих кодов:

Код	Тип диска
FF	двухсторонний, 8 секторов
FE	односторонний, 8 секторов

FD	двухсторонний, 9 секторов
FC	односторонний, 9 секторов
F9	двухсторонний, 15 секторов
F8	фиксированный диск

Сама таблица размещения файлов не является файлом. Она может быть считана при помощи функций DOS или BIOS, непосредственно читающих определенные сектора диска. Операционная система обеспечивает функцию, которая возвращает идентификационный байт диска.

Данные BIOS не показывают число жестких дисков в системе, так как переключатели предназначены только для гибких дисков. Однако можно использовать указанную функцию операционной системы для поиска накопителей. Она возвращает значение 0CDH вместо одного из упомянутых кодов, когда накопители отсутствуют. Надо просто проверять все большие и большие номера накопителей, до тех пор, пока не будет обнаружено указанное значение.

AT уникален в том смысле, что его информация о конфигурации говорит, какой тип накопителя используется. Эту информацию можно получить из порта с адресом 71H, предварительно послав номер регистра в порт 70H. Для НГМД номер регистра равен 10H. Информация о первом накопителе содержится в битах 7 – 4, а о втором – в битах 3 – 0. В обоих случаях цепочка битов 0000 говорит об отсутствии накопителя, 0001 – о двухстороннем накопителе с плотностью 48 дорожек на дюйм, а 0010 – о накопителе большой емкости (96 дорожек на дюйм). Информация о фиксированном диске содержится в регистре 12H. И снова биты 7 – 4 и 3 – 0 соответствуют первому и второму накопителям. 0000 указывает на отсутствие накопителя. Другие 15 возможных значений описывают емкость и конструкцию накопителя.

Эти коды сложные и если потребуется данная информация, то необходимо обратиться к техническому руководству по AT.

Средний уровень

Функция 1CH прерывания 21H возвращает информацию об указанном накопителе. Необходимо поместить номер накопителя в DL, причем 0 = накопитель по умолчанию, 1 = A, и т.д. При возвращении DX содержит число кластеров в FAT, AL – число секторов в кластере, а CX – число байтов в секторе. DS:BX указывает на байт,

содержащий код идентификации диска из FAT. В следующем примере определяется тип накопителя A:

```
; ---определение типа диска
  MOV  AH,1CH      ; функция MS DOS
  MOV  DL,1        ; выбор накопителя A
  INT  21H         ; получение информации
  MOV  DL,[BX]     ; получение типа накопителя
  CMP  DL,0FDH     ; двухсторонний, 9 секторов?
  JE   DBL_9       ; и т.д.
```

BIOS AT имеет функцию, сообщающую общие параметры накопителей.

Это функция 8 прерывания 13H. Она возвращает число накопителей в DL, максимальное число сторон накопителя в DH, максимальное число секторов в CL и дорожек в CH, а код статуса ошибки накопителя – в AH.

Другая функция BIOS AT возвращает тип накопителя. Это функция 15H прерывания 13H, которая требует номера накопителя в DL. В AH возвращается код, причем 0 = нет накопителя, 1 = дискета без обнаружения изменений, 2 = дискета с обнаружением изменений и 3 = фиксированный диск. В случае фиксированного диска в CX:DX возвращается число секторов по 512 байт.

Определение числа и типа периферийных устройств

При старте ROM-BIOS проверяет присоединенное оборудование, сообщая о результатах своей проверки в регистр статуса. Этот регистр занимает два байта, начиная с 0040:0010. Нижеприведенные значения битов относятся ко всем машинам, пока не оговорено обратное:

```
бит 0    если 1, то присутствует НГМД
1        XT,AT:1 = есть мат. сопроцессор (PC,PCjr:не использ.)
2-3      11 = базовая память 64K (AT:не используется)
4-5      активный видеоадаптер (11 = монохромный,
         10 = цветной 80*25, 01 = цветной 40*25)
6-7      число НГМД (если бит 0 = 1)
8        PCjr:0 = есть DMA (PC,XT,AT:не используется)
9-11     число адаптеров коммуникации
12       1 = есть игровой порт (AT:не используется)
```

13 PCjr:есть серийный принтер (PC,XT,AT:не использ.)
14-15 число присоединенных принтеров

Большая часть информация расшифровывается примитивно. Но информация о дисковых накопителях распределена между битами 0 и 6-7. Значение 0 в битах 6-7 указывает, что имеется один дисковый накопитель; чтобы узнать об отсутствии накопителей, надо проверить бит 0.

Число портов коммуникации может быть получено из области данных BIOS. BIOS отводит четыре 2-байтных поля для хранения базовых адресов вплоть до четырех COM портов (MS DOS использует только два из них). Базовый адрес – это младший из адресов портов, относящихся к группе портов, имеющих доступ к данному каналу коммуникации. Эти четыре поля начинаются с адреса 0040:0008. Порту COM1 соответствует адрес: 0008, а COM2 – 000A. Если это поле содержит 0, то соответствующий порт отсутствует. Таким образом, если слово по адресу 0008 отлично от нуля, а по адресу 000A – нулевое, то имеется один порт коммуникации.

AT хранит информацию о периферии в регистре 14H микросхемы конфигурации. Сначала необходимо записать 14H в порт с адресом 70H, а затем прочитать содержимое регистра через порт 71H. Вот значение битов этого регистра:

биты 7-6 00 = 1 НГМД, 01 = 2 НГМД
5-4 01 = вывод на цветной дисплей, 40 строк
10 = вывод на цветной дисплей, 80 строк
11 = вывод на монохромный дисплей
3-2 не используется
1 1 = имеется мат. сопроцессор
0 0 = нет НГМД, 1 = имеется НГМД

Высокий уровень

В Бейсике нужно просто прочитать байты статуса из области данных BIOS. В приведенном примере проверка наличия дисковых накопителей достигается проверкой четности младшего байта статусного регистра (четный – нет накопителей).

```
100 DEF SEG = 0          'указывыает на дно памяти
110 X = PEEK(&H410)      'получает младший байт регистра
```

```
120 IF X MOD 2 = 0 THEN 140 'он четный – нет накопителей
130 PRINT "имеется диск" ' или иначе имеется накопитель
140 GOTO 160          'переход ко второму сообщению
150 PRINT "нет накопителей"  'второе сообщение
160 ...               'продолжение...
```

Проверка наличия COM1:

```
100 DEF SEG = 40H       'указывает на область данных BIOS
110 PORT = PEEK(0) + 256*PEEK(1) 'получает слово со смещением 0
120 IF PORT = 0 THEN...  '... то нет адаптера COM1
```

Средний уровень

Прерывание 11H BIOS возвращает байт статуса оборудования в AX.

На входе ничего подавать не надо. В примере определяется число дисковых накопителей.

```
;  ---получение числа дисковых накопителей:
   INT   11H         ; получает байт статуса
   TEST  AL,0        ; имеются накопители?
   JZ    NO_DRIVES   ; переход, если нет
   AND   AL,1100000B ; выделяет биты 5-6
   MOV   CL,5        ; подготовка к сдвигу регистра
   SHR   AL,CL       ; сдвиг вправо на 5 битов
   INC   AL          ; добавляет 1, т.к. отсчет идет с 1
```

Низкий уровень

Ассемблерная программа работает так же, как и программа на Бейсике. В примере читается информация о конфигурации для AT, определяя, установлен ли математический сопроцессор:

```
   MOV   AL,14H      ; номер регистра
   OUT   70H,AL      ; посылаем запрос
   IN    AL,71H      ; читает регистр
   TEST  AL,10B      ; проверяет бит 1
   JZ    NO_COPROCESSOR ; если не установлен, то сопроцессора нет
```

Ревизия количества памяти

Вопрос: «Сколько имеется памяти?» может иметь три смысла. О каком количестве памяти сообщают переключатели, установленные на системной плате? Сколько микросхем памяти реально установлено в машине? И, наконец, сколько остается свободной памяти, которую DOS может использовать для выполнения программ? Машина может иметь 10 банков памяти по 64K, но переключатели могут указывать на наличие только 320K, оставляя половину памяти для каких-либо специальных целей. А как может программа узнать, сколько из доступных 320K она может использовать, учитывая, что другое программное обеспечение может быть загружено резидентным в верхнюю или нижнюю часть памяти?

Ответ на каждый вопрос можно получить своим способом. Для PC и XT установка переключателей может быть просто прочитана через порт B микросхемы интерфейса с периферией 8255. BIOS хранит двухбайтную переменную по адресу 0040:0013, которая сообщает число килобайт используемой памяти. Для PCjr бит 3 порта 62H (порт C микросхемы 8255) равен нулю, когда машина имеет добавочные 64K памяти. AT дает особо полную информацию о памяти. Регистры 15H (младший) и 16H (старший) микросхемы информации о конфигурации говорят, сколько памяти установлено на системной плате (возможны три значения: 0100H – для 256K, 0200H – для 512K и 0280H – для 512K плюс 128K на плате расширения). Память канала ввода/вывода для AT сообщается регистрами 17H и 18H (с инкрементом 512K). Память сверх 1-го мегабайта доступна через регистры 30H и 31H (опять с инкрементом 512K, вплоть до 15 мегабайт). Если AT имеет 128K на плате расширения, то установлен бит 7 регистра 33. Во всех случаях надо сначала послать номер регистра в порт 70H, а затем прочитать значение из порта 71H.

Легко написать программу, которая прямо тестирует наличие памяти через определенные интервалы адресного пространства. Поскольку минимальная порция памяти 16 килобайт, то достаточно проверить одну ячейку памяти в каждом 16-килобайтном сегменте, чтобы убедиться, что все 16K присутствуют. Когда данная ячейка памяти отсутствует, то при чтении из нее получают значение 233. Для проверки можно записать в ячейку произвольное число, отличное от 233, и сразу же считать его. Если вместо посланного числа возвращается 233, то соответствующий банк памяти отсутствует. Не стоит применять этот способ на AT, где при попытке писать

в несуществующую память вступает в действие встроенная обработка несуществующей памяти. Диагностика AT настолько хороша, что можно целиком положиться на системную информацию о конфигурации.

Память постоянно занимается частями операционной системы, драйверами устройств, резидентными программами обработки прерываний и управляющими блоками MS DOS. При проверке банков памяти нельзя вносить необратимых изменений в содержимое памяти.

Сначала надо сохранить значение, хранящееся в тестируемой ячейке, затем проверить ее и восстановить первоначальное значение.

Имеется еще одна проблема. Если процедура хотя бы временно модифицирует свой код, то это может привести к краху. Поэтому для проверки надо выбирать такую ячейку из блока 64K, которая не будет занята текстом данной процедуры. Для этого нужно поместить процедуру тестирования впереди программы, а для тестирования выбрать ячейку со смещением, равным смещению для кодового сегмента. Например, если регистр кодового сегмента содержит 13E2, то сегмент начинается со смещения 13E2 во втором 64K-байтном блоке памяти.

Поскольку подпрограмма проверки не может находиться по этому адресу, то можно безопасно проверять значение 3E2 в каждом блоке. Запрет прерываний позволяет не беспокоиться о модификации кода из-за аппаратных прерываний, которые могут происходить во время проверки.

Определение количества памяти для реально доступной операционной системы также требует некоторого фокуса. Когда программа первый раз получает управление, то DOS отводит ей всю доступную память, включая верхнюю область памяти, содержащую нерезидентную часть DOS (которая автоматически перезагружается, если она была модифицирована). Для запуска другой программы из текущей или для того, чтобы сделать программу подходящей для многопользовательсой системы, необходимо урезать программу до требуемого размера.

Эта же функция 4AH прерывания 21H может быть использована для расширения отведенной памяти. Поскольку программе отводится вся доступная память при загрузке, то такое расширение невозможно при старте. Если попробовать сделать это, то будет установлен флаг переноса, в регистре AX появится код ошибки 8, а в ре-

гистр BX будет возвращено максимальное число доступных 16-байтных параграфов. Такая информация как раз и нужна. Значит, надо выдать запрос со слишком большим значением в регистр BX (скажем, F000H параграфов), а затем выполнить прерывание. Необходимо позаботиться о том, чтобы выполнить эту функцию в самом начале программы, пока регистр ES еще имеет начальное значение.

Высокий уровень

Интерпретатор Бейсика использует только 64K (хотя операторы PEEK и POKE позволяют доступ к памяти за пределами 64K). Доля памяти, доступная в настоящий момент, возвращается функцией FRE.

Эта функция имеет фиктивный аргумент, который может быть числовым или символьной строкой. BYTES = FRE(x) передает в BYTES число свободных байтов. BYTES = FRE(x$) делает то же самое. Но строковый аргумент вынуждает очистить область данных перед тем, как возвратить число байтов. Заметим, что если размер рабочей области устанавливается с помощью оператора CLEAR, то количество памяти, сообщаемое функцией FRE, будет на 2,5 – 4 килобайта меньше из-за потребностей рабочей области интерпретатора.

Транслятор Бейсика не накладывает ограничение 64K на суммарный объем кода и данных. Но сам компилятор ограничен тем количеством памяти, которое он может использовать при компиляции. Если этого пространства недостаточно, то нужно уничтожить все ненужные номера строк при помощи ключа компиляции /N. Можно также использовать более короткие имена переменных.

Средний уровень

Прерывание 12H BIOS проверяет установку переключателей и возвращает в AX количество килобайт памяти в системе. Эта величина вычисляется из установки регистров микросхемы 8255 или, для AT, микросхемы конфигурации/часов. Входных регистров нет. Имеется в виду, что установка переключателей может быть неверной, что ограничивает достоверность такого подхода. Для определения числа 16-байтных параграфов, доступных для DOS, используется функция 4AH прерывания 21H. ES должен иметь то же значение, что при старте задачи:

```
; ---определение числа параграфов, доступных для DOS
```

```
  MOV   AH,4AH        ; указывает нужную функцию
  MOV   BX,0FFFFH     ; требует слишком большую память
  INT   21H           ; BX содержит число доступных параграфов
```

AT использует функцию 88H прерывания 15H для проверки наличия расширенной памяти, которая ищет память вне адресного пространства процессора в обычном режиме адресации. Говорят, что она ищет память за отметкой одного мегабайта. При этом на системной плате должно быть от 512 до 640 килобайт памяти, чтобы эта функция работала. Число килобайтных блоков расширенной памяти возвращается в AX.

Низкий уровень

Первый пример проверяет число банков памяти по 64K в первых десяти 64-килобайтных сегментах памяти. Если проверять старшие 6 банков памяти, то нужно иметь в виду, что имеются видеобуфер, начиная с B000:0000 (и, возможно, A000:0000) и ПЗУ, начиная с F000:0000 (и, возможно, C000:0000).

```
; ---проверка каждого банка памяти:
  CLI               ; запрет аппаратных прерываний
  MOV   AX,CS       ; получает значение кодового сегмента
  AND   AX,0FFFH    ; сбрасывает старшие 4 бита
  MOV   ES,AX       ; помещает указатель в ES
  MOV   DI,0        ; DI считает число банков памяти
  MOV   CX,10       ; будет проверяться 10 банков
  MOV   BL,'X'      ; для проверки использует 'X'
NEXT:
  MOV   DL,ES:[0]   ; сохраняет значение тестируемой ячейки
  MOV   ES:[0],BL   ; помещает 'X' в эту ячейку
  MOV   DH,ES:[0]   ; читает тестируемую ячейку
  MOV   ES:[0],DL   ; восстанавливает значение
  CMP   DH,'X'      ; совпадает с тем, что писали?
  JNE   GO_AHEAD    ; если нет, то банк отсутствует
  INC   DI          ; увеличивает число банков
GO_AHEAD:
  MOV   AX,ES       ; готовит увеличение указателя
  ADD   AX,1000H    ; указывает на следующие 64K
  MOV   ES,AX       ; возвращает указатель в ES
```

```
LOOP  NEXT          ; обрабатывает следующий банк
STI                 ; разрешает аппаратные прерывания
```

1.3. Управление прерываниями

Прерывания – это готовые процедуры, которые компьютер вызывает для выполнения определенной задачи. Существуют аппаратные и программные прерывания. Аппаратные прерывания инициируются аппаратурой либо с системной платы, либо с карты расширения. Они могут быть вызваны сигналом микросхемы таймера, сигналом от принтера, нажатием клавиши на клавиатуре и множеством других причин. Аппаратные прерывания не координируются с работой программного обеспечения. Когда вызывается прерывание, то процессор оставляет свою работу, выполняет прерывание, а затем возвращается на прежнее место. Для того чтобы иметь возможность вернуться точно в нужное место программы, адрес этого места (CS:IP) запоминается на стеке вместе с регистром флагов. Затем в CS:IP загружается адрес программы обработки прерывания и ей передается управление. Программы обработки прерываний иногда называют драйверами прерываний. Они всегда завершаются инструкцией IRET (возврат из прерывания), которая завершает процесс, начатый прерыванием, возвращая старые значения CS:IP и регистра флагов, тем самым давая программе возможность продолжить выполнение из того же состояния.

С другой стороны, программные прерывания фактически ничего не прерывают. На самом деле это обычные процедуры, которые вызываются пользовательскими программами для выполнения рутинной работы, такой, как прием нажатия клавиши на клавиатуре или вывод на экран. Однако эти подпрограммы содержатся не внутри программы пользователя, а в операционной системе, и механизм прерываний дает возможность обратиться к ним. Программные прерывания могут вызываться друг из друга. Например, все прерывания обработки ввода с клавиатуры DOS используют прерывания обработки ввода с клавиатуры BIOS для получения символа из буфера клавиатуры. Отметим, что аппаратное прерывание может получить управление при выполнении программного прерывания. При этом не возникает конфликтов, так как каждая подпрограмма обработки прерывания

сохраняет значения всех используемых ею регистров и затем восстанавливает их при выходе, тем самым не оставляя следов того, что она занимала процессор.

Адреса программ прерываний называют векторами. Каждый вектор имеет длину четыре байта. В первом слове хранится значение IP, а во втором – CS. Младшие 1024 байта памяти содержат вектора прерываний, таким образом имеется место для 256 векторов. Вместе взятые они называются таблицей векторов. Вектор для прерывания 0 начинается с ячейки 0000:0000, прерывания 1 – с 0000:0004, 2 – с 0000:0008 и т.д. Если посмотреть на четыре байта, начиная с адреса 0000:0020, в которых содержится вектор прерывания 8H (прерывание времени суток), то можно обнаружить там A5FE00F0. Имея в виду, что младший байт слова расположен в начале и что порядок IP:CS, это 4-байтное значение переводится в F000:FEA5. Это стартовый адрес программы ПЗУ, выполняющей прерывание 8H.

Программирование контроллера прерываний 8259

Для управления аппаратными прерываниями во всех типах IBM PC используется микросхема программируемого контроллера прерываний Intel 8259. Поскольку в каждый момент времени может поступить не один запрос, микросхема имеет схему приоритетов. Имеется 8 уровней приоритетов, кроме AT, у которого их 16, и обращения к соответствующим уровням обозначаются сокращениями от IRQ0 до IRQ7 (от IRQ0 до IRQ15), что означает запрос на прерывание. Максимальный приоритет соответствует уровню 0. Добавочные 8 уровней для AT обрабатываются второй микросхемой 8259; этот второй набор уровней имеет приоритет между IRQ2 и IRQ3. Запросы на прерывание 0 – 7 соответствуют векторам прерываний от 8H до 0FH; для AT запросы на прерывания 8 – 15 обслуживаются векторами от 70H до 77H. Ниже приведены назначения этих прерываний.

Аппаратные прерывания в порядке приоритета:

IRQ 0 таймер
1 клавиатура
2 канал ввода/вывода
8 часы реального времени (только AT)
9 программно переводятся в IRQ2 (только AT)
10 резерв
11 резерв

12 резерв
13 мат. сопроцессор (только AT)
14 контроллер фиксированного диска (только AT)
15 резерв
3 COM1 (COM2 для AT)
4 COM2 (модем для PCjr, COM1 для AT)
5 фиксированный диск (LPT2 для AT)
6 контроллер дискет
7 LPT1

Прерыванию времени суток дан максимальный приоритет, поскольку если оно будет постоянно теряться, то будут неверными показания системных часов. Прерывание от клавиатуры вызывается при нажатии или отпускании клавиши; оно вызывает цепь событий, которая обычно заканчивается тем, что код клавиши помещается в буфер клавиатуры (откуда он затем может быть получен программными прерываниями).

Микросхема 8259 имеет три однобайтных регистра, которые управляют восемью линиями аппаратных прерываний. Регистр запроса на прерывание (IRR) устанавливает соответствующий бит, когда линия прерывания сигнализирует о запросе. Затем микросхема автоматически проверяет, не обрабатывается ли другое прерывание. При этом она запрашивает информацию регистра обслуживания (ISR). Дополнительная цепь отвечает за схему приоритетов. Наконец, перед вызовом прерывания проверяется регистр маски прерываний (IMR), чтобы узнать, разрешено ли в данный момент прерывание данного уровня.

Как правило, программисты обращаются только к регистру маски прерываний через порт 21H и командному регистру прерываний через порт 20H.

Запрет/разрешение отдельных аппаратных прерываний

Программы на аасемблере могут запретить аппаратные прерывания.

Это маскируемые прерывания; другие аппаратные прерывания, возникающие при некоторых ошибках (таких, как деление на ноль) не могут быть маскированы. Имеются две причины для запрета аппаратных прерываний. В первом случае все прерывания блокируются, с тем чтобы критическая часть кода была выполнена целиком, прежде чем машина произведет какое-либо другое действие.

Например, прерывания запрещают при изменении вектора аппаратного прерывания, избегая выполнения прерывания, когда вектор изменен только наполовину.

Во втором случае маскируются только определенные аппаратные прерывания. Это делается, когда некоторые определенные прерывания могут взаимодействовать с операциями, критичными к временам. Например, точно рассчитанная по времени процедура ввода/вывода не может себе позволить быть прерванной длительным дисковым прерыванием.

Низкий уровень

Выполнение прерываний зависит от значения флага прерывания (бит 9) в регистре флагов. Когда этот бит равен 0, то разрешены все прерывания, которые разрешает маска. Когда он равен 1, то все аппаратные прерывания запрещены. Чтобы запретить прерывания, установив этот флаг в 1, используется инструкция CLI. Для очистки этого флага и восстановления прерываний – инструкция STI. Необходимо избегать отключения прерываний на длительный период. Прерывание времени суток происходит 18,2 раза в секунду и если к этому прерыванию был более чем один запрос в то время, когда аппаратные прерывания были запрещены, то лишние запросы будут отброшены и системное время будет определяться неправильно.

Следует иметь в виду, что машина автоматически запрещает аппаратные прерывания при вызове программных прерываний и автоматически разрешает их при возврате. Когда пишутся программные прерывания, то можно начать программу с инструкции STI, если допускаются аппаратные прерывания. Отметим также, что если за инструкцией CLI не следует STI, то это приведет к остановке машины, так как ввод с клавиатуры будет заморожен.

Для маскирования определенных аппаратных прерываний нужно просто послать требуемую цепочку битов в порт с адресом 21H, который соответствует регистру маски прерываний (IMR). Регистр маски на второй микросхеме 8259 для AT (IRQ8-15) имеет адрес порта A1H. Нужно установить биты регистра, соответствующие номерам прерываний, которые необходимо маскировать. В этот регистр можно только записывать. Нижеприведенный пример блокирует дисковое прерывание. Не стоит забывать очистить регистр в конце программы, иначе обращение к дискам будет запрещено и после завершения программы.

```
; ---маскирование 6-го бита регистра маски прерываний
  MOV   AL,01000000B   ; маскируется бит 6
  OUT   21H,AL         ; посылается в регистр маски прерываний
  .
  MOV   AL,0           ;
  OUT   21H,AL         ; очищается IMR в конце программы
```

Написание собственного прерывания

Имеется несколько причин для написания собственного прерывания. Во-первых, большинство из готовых прерываний, обеспечиваемых операционной системой, – не что иное, как обычные процедуры, доступные для всех программ, и можно пожелать добавить их в эту библиотеку. Например, многие программы могут использовать процедуру, выводящую строки на экран вертикально. Вместо того чтобы включать ее в каждую программу в качестве процедуры, можно установить ее как прерывание, написав программу, которая останется резидентной в памяти после завершения . Тогда можно использовать INT 80H вместо WRITE_VERTICALLY (нужно иметь в виду, что вызов прерывания происходит несколько медленнее, чем вызов процедуры).

Второй причиной написани прерывания может быть использование какого-либо отдельного аппаратного прерывания. Это прерывание автоматически вызывается при возникновении определенных условий. В некоторых случаях BIOS инициализирует вектор этого прерывания так, что он указывает на процедуру, которая вообще ничего не делает (она содержит один оператор IRET). Можно написать собственную процедуру и изменить вектор прерываний, чтобы он указывал на нее. Тогда при возникновении аппаратного прерывания будет выполняться процедура пользователя. Одна из таких процедур – это прерывание времени суток, которое автоматически вызывается 18,2 раза в секунду. Обычно это прерывание только обновляет показание часов, но можно добавить к нему любой код, который пожелается. Если код проверяет показания часов и вступает в игру в определенные моменты времени, то возможны операции в реальном времени.

Другие возможности – это написание процедур обработки Ctrl-Break, PrtSC и возникновения ошибочных ситуаций. Прерывания принтера и коммуникационные позволяют компьютеру быстро переключаться между операциями ввода/вывода и другой обработкой.

Наконец, можно захотеть написать прерывание, приспособленное к программным нуждам, которое полностью заменит одну из процедур операционной системы.

Средний уровень

Функция 25H прерывания 21H устанавливает вектор прерывания на указанный адрес. Адреса имеют размер в два слова. Старшее слово содержит значение сегмента (CS), младшее – смещение (IP). Чтобы установить вектор, указывающий на одну из процедур, нужно поместить сегмент процедуры в DS, а смещение – в DX (следуя порядку нижеприведенного примера). Затем поместить номер прерывания в AL и вызвать функцию. Любая процедура прерывания должна завершаться не обычной инструкцией RET, а IRET. (IRET выталкивает из стека три слова, включая регистр флагов, в то время как RET помещает на стек только два. Если попытаться тестировать такую процедуру как обычную процедуру, но кончающуюся IRET, то стек исчерпается). Отметим, что функция 25H автоматически запрещает аппаратные прерывания в процессе изменения вектора, поэтому не существует опасности, что посреди дороги произойдет аппаратное прерывание, использующее данный вектор.

```
; ---установка прерывания
  PUSH  DS          ; сохраняется DS
  MOV   DX,OFFSET ROUT ; смещение для процедуры в DX
  MOV   AX,SEG ROUT    ; сегмент процедуры
  MOV   DS,AX          ; помещается в DS
  MOV   AH,25H         ; функция установки вектора
  MOV   AL,60H         ; номер вектора
  INT   21H          ; меняется прерывание
  POP   DS           ; восстанавливается DS

; ---процедура прерывания
ROUT  PROC  FAR
    PUSH  AX         ; сохраняются все изменяемые регистры
     .
     .
    POP   AX         ; восстанавливаются регистры
    MOV   AL,20H     ; эти две строки надо использовать
    OUT   20H,AL     ; только для аппаратных прерываний
    IRET
ROUT  ENDP
```

В конце кода каждого из аппаратных прерываний нужно включить следующие 2 строчки кода:

```
MOV  AL,20H
OUT  20H,AL
```

Это просто совпадение, что числа (20H) одни и те же в обеих строках. Если аппаратное прерывание не заканчивается этими строками, то микросхема 8259 не очистит информацию регистра обслуживания, с тем чтобы была разрешена обработка прерываний с более низкими уровнями, чем только что обработанное. Отсутствие этих строк легко может привести к краху программы, так как прерывания от клавиатуры скорее всего окажутся замороженными и даже Ctrl-Alt-Del окажется бесполезным. Отметим, что эта добавка не нужна для тех векторов прерываний, которые являются расширениями существующих прерываний, таким, как прерывание 1CH, которое добавляет код к прерыванию времени суток.

Когда программа завершается, то должны быть восстановлены оригинальные вектора прерываний. В противном случае последующая программа может вызвать данное прерывание и передать управление на то место в памяти, в котором заданной процедуры уже нет. Функция 35 прерывания 21H возвращает текущее значение вектора прерывания, помещая значение сегмента в ES, а смещение – в BX. Перед установкой своего прерывания нужно получить текущее значение вектора; используя эту функцию, сохранить эти значения, и затем восстановить их с помощью функции 25H (как выше) перед завершением своей программы.

Например:

```
; ---в сегменте данных:
  KEEP_CS  DW   0       ; хранит сегмент заменяемого прерывания
  KEEP_IP  DW   0       ; хранит смещение прерывания
; ---в начале программы
        MOV  AH,25H     ; функция получения вектора
        MOV  AL,1CH     ; номер вектора
        INT  21H        ; теперь сегмент в ES, смещение в BX
        MOV  KEEP_IP,BX ; запоминается смещение
        MOV  KEEP_CS,ES ; запоминается сегмент
```

```
; ---в конце программы
        CLI
        PUSH  DS        ;  DS будет разрушен
        MOV   DX,KEEP_IP ;  подготовка к восстановлению
        MOV   AX,KEEP_CS ;
        MOV   DS,AX      ; подготовка к восстановлению
        MOV   AH,25H     ; функция установки вектора
        MOV   AL,1CH     ; номер вектора
        INT   21H        ; восстанавливается вектор
        POP   DS         ; восстанавливается DS
        STI
```

Имеется пара ловушек, которых следует избегать при написании прерывания. Если новая процедура прерывания должна иметь доступ к данным, то необходимо позаботиться, чтобы DS был правильно установлен (обычно прерывание может использовать стек вызывающей программы). Другая неприятность может заключаться в том, что при завершении программы по Ctrl-Break вектор прерывания не будет восстановлен, если только не будет предусмотрено, чтобы программа реакции на Ctrl-Break выполняла эту процедуру.

Низкий уровень

Описанные выше функции MS DOS просто получают или изменяют пару слов в младших ячейках памяти. Смещение вектора может быть вычислено простым умножением номера вектора на 4. Например, чтобы получить адрес прерывания 16H в ES:BX:

```
; ---получение адреса прерывания 16H
  SUB   AX,AX        ; устанавливается ES на начало памяти
  MOV   ES,AX        ;
  MOV   DI,16H       ; номер прерывания в DI
  SHL   DI,1         ; умножается на 2
  SHL   DI,1         ; умножается на 2
  MOV   BX,ES:[DI]    ; берется младший байт в BX
  MOV   AX,ES:[DI]+2  ; берется старший байт в ES
  MOV   ES,AX        ;
```

Не рекомендуется прямо устанавливать вектор прерываний, обходя функцию DOS. В частности, в многозадачной среде операцион-

ная система может поддерживать несколько таблиц векторов прерываний и реальный физический адрес таблицы может быть известен только DOS.

Дополнение к существующему прерыванию

Хотя и не часто, но иногда бывает полезно добавить код к существующему прерыванию. В качестве примера рассмотрим программы, которые преобразуют одно нажатие клавиши в длиные, определяемые пользователем символьные строки (макроопределения клавиатуры). Эти программы используют тот факт, что весь ввод с клавиатуры поступает через функцию 0 прерывания 16H BIOS. Все прерывания ввода с клавиатуры DOS вызывают прерывание BIOS для получения символа из буфера клавиатуры. Поэтому необходимо модифицировать лишь прерывание 16H, таким образом, чтобы оно служило шлагбаумом для макроопределений, после чего любая программа будет получать макроопределения, независимо от того, какое прерывание ввода с клавиатуры она использует.

Конечно, модифицировать прерывания BIOS и DOS непросто, поскольку BIOS расположена в ПЗУ, а DOS поступает без листинга и обе они ограничены размерами отведенной для них памяти. Но можно написать процедуру, которая предшествует и/или следует за соответствующим прерыванием, и эта процедура может вызываться при вызове прерывания DOS или BIOS. Например, в случае прерывания 16H нужно написать процедуру и указать на нее вектором прерывания для 16H. Оригинальное значение вектора 16H тем временем переносится в какой-либо неиспользуемый вектор, скажем, 60H.

Новая процедура просто вызывает прерывание 60H, чтобы использовать оригинальное прерывание 16H; поэтому когда программа вызывает прерывание 16H, управление передается процедуре, которая затем вызывает оригинальное прерывание 16H и по завершении опять возвращает управление процедуре, а из нее уже возвращается в то место программы, из которого был вызов прерывания 16H. После того как это сделано, в новой процедуре может содержаться любой код, как до, так и после вызова прерывания 60H. Перечень необходимых действий:

1. Создать новую процедуру, вызывающую прерывание 60H.
2. Перенести вектор прерывания для 16H в 60H.
3. Изменить вектор 16H, чтобы он указывал на новую процедуру.
4. Завершить программу, оставляя ее резидентной.

1.4. Управление программами

Большинство программ загружаются в память, запускаются, а затем удаляются операционной системой при завершении. Языки высокого уровня обычно не имеют альтернативы. Но для программистов на ассемблере имеется другая возможность, и данный подпункт демонстрирует ее. Некоторые программы действуют как драйверы устройств или драйверы прерываний и они должны быть сохранены в памяти («резидентными») даже после их завершения (вектора прерываний обеспечивают механизм, посредством которого последующие программы могут обращаться к резидентным процедурам). Иногда программе необходимо запустить из себя другую программу. На самом деле DOS позволяет программе загрузить в память вторую копию COMMAND.COM, которая может быть использована как средство интерфейса с пользователем или выполнения команд типа COPY или DIR.

Программы могут быть представлены в двух форматах: *.EXE или *.COM. Размер файла с программами первого типа может быть больше 64K, но такие программы требуют некоторой обработки перед тем, как DOS загрузит их в память. С другой стороны, программы второго типа особенно полезны для коротких утилит и существуют прямо в том формате, который нужен для загрузки в память. В обоих случаях код, составляющий программу, предваряется в памяти префиксом программного сегмента (PSP). Это область размером 100H байт, которая содержит информацию, необходимую DOS для работы программы; PSP также обеспечивает место для файловых операций ввода/вывода. При загрузке *.EXE файла и DS, и ES указывают на PSP. Для *.COM файлов CS также сначала указывает на PSP.

1.5. Манипуляции с памятью

Когда MS DOS загружает программу, то помещает ее в младшую область памяти, сразу же за COMMAND.COM и установленными драйверами устройств или другими утилитами, которые резидентны в памяти. В этот момент времени вся память за программой отведена этой программе. Если программе нужна память для создания области данных, то она может приближенно вычислить, где в памяти кончается ее код, и затем поместить требуемую область данных в любое ме-

сто за концом кода. Для определения адреса конца программы нужно поместить в конце программы псевдосегмент типа:

```
ZSEG   SEGMENT
       ;
ZSEG   ENDS
```

В ассемблере IBM PC ZSEG будет последним сегментом, так как сегменты располагаются в алфавитном порядке. С другими ассемблерами нужно действительно поместить эти строки в конце программы.

В самой программе достаточно поставить оператор MOV AX, ZSEG и AX будет указывать на первый свободный сегмент памяти за программой.

Такой подход работает до тех пор, пока программа предполагает наличие памяти, которой на самом деле нет. Но не будет также работать в многопользовательской среде, когда несколько программ могут делить между собой одну и ту же область адресов. Для решения этой проблемы MS DOS имеет возможность отслеживать 640K системной памяти и отводить по требованию программы блоки памяти любого размера. Блок памяти – это просто непрерывная область памяти, его максимальный размер определяется размером доступной памяти, в частности, он может быть больше одного сегмента (64K). Если затребован слишком большой блок, то DOS выдает сообщение об ошибке. Любая возможность перекрытия блоков исключена. Кроме того, MS DOS может освобождать, урезать или расширять существующие блоки. Хотя программа не обязана использовать эти средства, но удобно и предусмотрительно делать это. Некоторые функции DOS требуют, чтобы были использованы средства управления памятью DOS, например, завершение резидентной программы или вызов другой программы из данной.

Прежде чем отвести память, существующий блок (вся память от начала программы до конца) должен быть обрезан до размера программы. Затем, при создании блока, DOS создает 16-байтный управляющий блок памяти, который расположен непосредственно перед блоком памяти. Первые 5 байтов этого блока имеют следующее значение:

байт 0 ASCII 90 – если последний блок в цепочке, иначе ASCII 77.

байты 1-2 0 – если блок освобожден
байты 3-4 – размер блока в 16-байтных параграфах

DOS обращается к блокам по цепочке. Адрес первого блока хранится во внутренней переменной. Значение этой переменной позволяет DOS определить положение первого отведенного блока, а из информации, содержащейся в нем, может быть найден следующий блок и т.д.

MS DOS обеспечивает три функции распределения памяти, номера от 48H до 4AH прерывания 21H. Функция 48H отводит блок памяти, а 49H – освобождает блок памяти. Третья функция («SETBLOCK») меняет размер памяти, отведенной для программы; эта функция должна быть использована перед двумя остальными. После ее выполнения можно спокойно отводить и освобождать блоки памяти. Программа должна освободить все отведенные ею блоки перед завершением.

Иначе эта память будет недоступной для последующего использования.

Использование команд интерфейса с пользователем из программы

Программа может иметь в своем распоряжении полный набор команд интерфейса с пользователем DOS, таких, как DIR или CHKDSK. Когда эти команды используются из программы, загружается и запускается вторая копия COMMAND.COM. Хотя такой подход может сэкономить много усилий при программировании, для его успешной реализации требуется достаточное количество памяти для этой второй копии, но программа пользователя может попасть в ловушку, если памяти недостаточно.

Загрузка и запуск программных оверлеев

Оверлеи – это части программы, которые остаются на диске, в то время как тело программы резидентно в памяти. Когда требуется функция, выполняемая каким-либо оверлеем, то он загружается в память и программа вызывает его как процедуру. Различные оверлеи могут загружаться в одно и то же место памяти, перекрывая предыдущий код. Например, программа ведения базы данных может загрузить процедуру сортировки, а затем перекрыть ее процедурой генерации отчетов. Эта техника используется для экономии памяти.

Но она хороша только для тех процедур, которые не используются постоянно, иначе частые обращения к диску приведут к тому, что программа будет выполняться слишком медленно.

Средний уровень

MS DOS использует функцию EXEC для загрузки оверлеев. Эта функция, номер 4BH прерывания 21H, используется также для загрузки и запуска одной программы из другой, если поместить код 0 в AL. Если в AL поместить код 3, то тогда будет загружен оверлей. В этом случае не создается PSP, поэтому оверлей не устанавливается как независимая программа. Такая процедура просто загружает оверлей, не передавая ему управления.

Имеются два способа обеспечить память для оверлея. Может быть использована либо область внутри тела программы, либо специально отведена область памяти за пределами головной программы. Функции EXEC передается только сегментный адрес, в качестве позиции, куда будет загружен оверлей. Когда оверлей загружается в тело головной программы, то программа должна вычислить номер параграфа, куда будет загружаться оверлей, сама. С другой стороны, при загрузке в специально отведенную память MS DOS обеспечивает программу номером параграфа.

В нижеприведенном примере используется загрузка в отведенную память. Поскольку DOS отводит программе всю доступную память, то сначала необходимо освободить память с помощью функции 4AH. Функция 48H отводит достаточно большой блок памяти, чтобы он мог принять самый большой из оверлеев. Эта функция возвращает значение сегмента блока в AX, и этот номер параграфа определяет, куда будет загружен оверлей, а также по какому адресу оверлей будет вызываться головной программой. Кроме кода 3, засылаемого в AL, необходимо установить для этой функции еще два параметра. DS:DX должны указывать на строку, дающую путь к файлу оверлея, завершаемую байтом ASCII 0. Необходимо указывать полное имя файла, включая расширение .COM или .EXE, поскольку DOS в данном случае не считает, что он ищет программный файл.

Наконец, ES:BX должны указывать на 4-байтный блок параметров, который содержит (1) 2-байтный номер параграфа, куда будет загружаться оверлей, и (2) 2-байтный фактор привязки, который будет использоваться для привязки адресов в оверлее. В качестве номера параграфа надо использовать число, возвращаемое в AX, для

номера параграфа отведенного блока памяти. Фактор привязки дает смещение, по которому могут быть вычислены адреса требующих привязки параметров в оверлее. Необходимо использовать номер параграфа, куда загружается оверлей. После того, как он установлен, нужно вызвать функцию и оверлей будет загружен. Просто изменяя путь к оверлейному файлу, можно вновь и вновь вызывать эту функцию, загружая все новые и новые оверлеи. Если при возврате установлен флаг переноса, то была ошибка и ее код будет возвращен в AX. Код равен единице, если указан неверный номер функции, 2 – если файл не найден, 5 – при дисковых ошибках и 8 – при отсутствии достаточной памяти.

После того, как оверлей загружен в память, к нему можно получить доступ как к далекой (far) процедуре. В сегменте данных должен быть установлен двухсловный указатель, определяющий этот вызов. Сегментная часть указателя просто равна текущему кодовому сегменту. Смещение оверлея должно быть вычислено нахождением разницы между сегментами кода и оверлея и умножением результата на 16 (переводя величину из параграфов в байты). В нижеприведенном примере две переменные OVERLAY_OFFSET и CODE_SEG помещены одна за другой для правильной установки указателя. Однажды загруженный, оверелей затем может вызываться инструкцией CALL DWORD

PTR OVERLAY_OFFSET.

Оверлей может быть полной программой со своими сегментами данных и стека, хотя, как правило, используется стековый сегмент вызывающей программы. При вызове оверлея значение сегмента его собственного сегмента данных должно быть помещено в DS.

```
; ---завершает программу фиктивным сегментом
ZSEG        SEGMENT
ZSEG        ENDS

; ---в сегменте данных
OVERLAY_SEG    DW    ?
OVERLAY_OFFSET DW    ?              ; смещение оверлея
CODE_SEG       DW    ?              ; сегмент оверлея – должен
PATH           DB    'A:OVERLAY.EXE' ; следовать за смещением
0BLOCK         DD    0              ; 4-байтный блок параметров
```

```
; ---освобождает память
  MOV  CODE_SEG,CS    ; создание копии CS
  MOV  AX,ES          ; копирование значения сегмента PSP
  MOV  BX,ZSEG        ; адрес сегмента конца программы
  SUB  BX,AX          ; вычисление разности
  MOV  AH,4AH         ; номер функции SETBLOCK
  INT  21H            ; освобождает память
  JC   SETBLK_ERR     ; флаг переноса говорит об ошибке
; ---отводит память для оверлея
  MOV  BX,100H        ; отводит для оверлея 1000H байт
  MOV  AH,48H         ; функция отведения памяти
  INT  21H            ; теперь AX:0 указывает на блок
  JC   ALLOCATION_ERR ; флаг переноса говорит об ошибке
  MOV  OVERLAY_SEG,AX ; запасает адрес сегмента оверлея
; ---вычисление смещения оверлея в кодовом сегменте
  MOV  AX,CODE_SEG    ; вычитает значение сегмента оверлея
  MOV  BX,OVERLAY_SEG ; из значения сегмента кода
  SUB  BX,AX          ; BX содержит число параграфов
  MOV  CL,4           ; сдвигает это число на 4 бита влево
  SHL  BX,CL          ; чтобы получить величину в байтах
  MOV  OVERLAY_OFFSET,BX ; запоминает смещение
; ---загрузка первого оверлея
  MOV  AX,SEG BLOCK ; ES:BX указывает на блок параметров
  MOV  ES,AX        ;
  MOV  BX,OFFSET BLOCK ;
  MOV AX,OVERLAY_SEG ; помещает адрес сегмента оверлея
  MOV  [BX],AX        ; в первое слово блока параметров
  MOV  [BX]+2,AX      ; сегмент оверлея – фактор привязки
  LEA  DX,PATH        ; DS:DX указывает на путь к файлу
  MOV  AH,48H         ; номер функции EXEC
  MOV  AL,3           ; код загрузки оверлея
  INT  21H            ; загружается оверлей
  JC   LOAD_ERROR     ; флаг переноса говорит об ошибке
; ---теперь программа занимается своими делами
   .
   .
  CALL  DWORD PTR OVERLAY_OFFSET  ; вызов оверлея
   .    ; нужно указывать DWORD PTR, так как оверлей –
   .    ; далекая процедура
```

```
;---необходимо посмотреть эту структуру при написании оверлея
DSEG SEGMENT ; как обычно, устанавливается сегмент данных
        .       ; опускается стековый сегмент (используется
        .       ; стек вызывающей программы)
DSEG    ENDS

CSEG    SEGMENT    PARA PUBLIC 'CODE'
OVERLAY  PROC FAR       ; всегда «далекая» процедура
        ASSUME CS:CSEG,DS:DSEG
        PUSH   DS     ; хранит DS вызывающей программы
        MOV    AX,DSEG; устанавливается DS оверлея
        MOV    DS,AX
         .
         .
        POP    DS     ; восстанавливается DS при завершении
        RET
OVERLAY  ENDP
CSEG    ENDS
        END
```

Преобразование программ из типа .EXE в тип .COM

Программисты на ассемблере имеют возможность преобразовать свои программы из обычного формата EXE в формат COM. Файлы EXE имеют заголовок, содержащий информацию для привязки; DOS привязывает некоторые адреса программы при загрузке. С другой стороны, файлы COM существуют в таком виде, что привязка не требуется – они хранятся уже в том виде, в котором загружаемая программа должна быть в памяти машины. По этой причине файлы EXE, по меньшей мере, на 768 байтов больше на диске, чем их COM эквиваленты (хотя при загрузке в память они будут занимать одинаковое место). Файлы COM также быстрее загружаются, поскольку не требуется привязки.

Других преимуществ у них нет, а некоторые программы слишком сложны и слишком велики, чтобы их можно было преобразовать в тип COM.

Привязка – это процесс установки адресов, связанных с сегментным регистром. Например, программа может указывать на начало области данных следующим кодом:

```
MOV   DX,OFFSET DATA_AREA
```

```
MOV   AX,SEG DATA_AREA
MOV   DS,AX
```

Смещение в DX связано с установкой сегментного регистра DS. Но какое значение должен принимать сам DS? Программа требует абсолютный адрес, но номер параграфа, в котором будет располагаться DATA_AREA, зависит от того, в какое место в памяти будет загружена программа. А это зависит от версии MS DOS, а также от того, какие резидентные программы будут находиться в младших адресах памяти. По этой причине во время компоновки программы можно только установить некоторые сегментные значения через смещения относительно начала программы. Затем, когда DOS осуществляет привязку, значение начального адреса программы прибавляется к сегментным значениям, давая абсолютные адреса, требуемые в сегментном регистре.

Файлы COM не нуждаются в привязке, поскольку хранятся в таком виде, что не нуждаются в фиксации сегмента. Все в программе хранится относительно начала кодового сегмента, включая все данные и стек. По этой причине вся программа не может превышать 65535 байт по длине, что соответствует максимальному смещению, которое существует в используемой схеме адресации (поскольку верхняя часть этого блока занята стеком, то реальное пространство, доступное для кода и данных, немного меньше, чем 65535 байт, хотя стековый сегмент при необходимости может быть вынесен за границу 64K – байтного блока). В файлах COM все сегментные регистры указывают на начало PSP, если сравнить с файлами EXE, где DS и ES инициализируются аналогичным образом, но CS указывает на первый байт, следующий за PSP.

Для представления программы в виде файла COM требуется соблюдение следующих правил:

1. Нельзя оформлять программу в виде процедуры. Вместо этого нужно поместить в самое начало метку, вроде START, и завершить программу оператором END START.

2. Необходимо поместить в начале программы оператор ORG 100H, который указывает начало кода (т.е. устанавливает счетчик комманд).

Программы COM начинаются с 100H, что является первым байтом, следующим за PSP, поскольку CS указывает на начало PSP, которое расположено на 100H байт ниже. Для того чтобы начать выполнение с любого другого места, нужно поместить по адресу 100H инструкцию JMP.

3. Оператор ASSUME должен устанавливать DS, ES и SS таким образом, чтобы они совпадали со значением для кодового сегмента, например, ASSUME CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG.

4. Данные программы могут помещаться в любом месте программы до тех пор, пока они не перемешаны с кодом. Лучше начинать программы с области данных, поскольку макроассемблер может выдавать сообщения об ошибках при первом проходе, если имеются ссылки на идентификатор данных, который еще не обнаружен. Для перехода к началу кода в качестве первой команды программы можно использовать инструкцию JMP.

5. Нельзя использовать фиксацию сегментов типа MOV AX,SEG NEW_DATA. Достаточно указания одного смещения метки. В частности, нужно опускать обычный код, используемый в начале программы для установки сегмента данных MOV AX,DSEG / MOV DS,AX.

6. Стековый сегмент полностью опускается в начальном коде. Указатель стека инициализируется на вершину адресного пространства 64K, используемого программой (напоминаем, что стек растет вниз в памяти). В программах COM он должен быть сделан меньше, чем 64K; SS и SP могут быть изменены. Стоит иметь в виду, что при компоновке программы компоновщик выдаст сообщение об ошибке, указывающее, что сегмент стека отсутствует. Игнорируйте его.

7. Завершается программа либо инструкцией RET, либо прерыванием 20H. Прерывание 20H – это стандартная функция для завершения программы и возврата управления в DOS. Даже когда программа завершается инструкцией RET, на самом деле, используется прерывание 20H. Это происходит потому, что вершина стека первоначально содержит 0. При выполнении завершающей инструкции программы RET 0 выталкивается из стека, переназначая счетчик команд на начало PSP. Находящаяся в этой ячейке функция 20H выполняется как следующая инструкция программы, вызывая передачу управления в DOS.

Все это означает, что при старте программы не надо помещать на стек DS и 0 (PUSH DS / MOV AX,0 / PUSH AX), как это требуется для EXE файлов.

После того как программа сконструирована таким образом, необходимо ассемблировать и компоновать ее как обычно. Затем преобразовать ее в форму COM с помощью утилиты EXE2BIN, имеющейся в MS DOS. Если имя программы, построенной компоновщи-

ком, MYPROG.EXE, то нужно просто ввести команду EXE2BIN MYPROG. В результате получится программный файл с именем MYPROG.BIN. Все, что остается после этого сделать – переименовать этот файл в MYPROG.COM. Можно также сразу использовать команду EXE2BIN MYPROG MYPROG.COM для получения файла с расширением COM.

Контрольные вопросы

1. Что такое программирование?
2. В какой форме составлялись программы для первых ЭВМ?
3. Почему языки автокоды (ассемблеры) называются машинно-ориентированными языками программирования?
4. Перечислите основные процедурные языки программирования в хронологической последовательности их создания.
5. Что такое парадигма программирования?
6. Назовите основные парадигмы программирования и их отличия друг от друга.
7. Что такое структурное программирование?
8. Что такое визуальное программирование?

2. АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ И ПРОГРАММИРОВАНИЕ

2.1. Методика подготовки и решения задачи на ЭВМ

Решение задачи на ЭВМ – сложный и трудоемкий процесс, который начинается с постановки задачи. На основе словесной формулировки задачи выбираются переменные, подлежащие определению, записываются ограничения, связи между переменными, в совокупности образующие математическую модель решаемой проблемы. Анализируется метод решения. На этом этапе необходимо принять очень важное решение – использовать ли имеющееся готовое программное обеспечение или разрабатывать собственную программу. Дешевле и быстрее использовать имеющиеся в наличии готовые разработки. Обновление программного обеспечения – задача программистов. В этом случае традиционно выделяются следующие основные *этапы решения* задачи на ЭВМ:

1) постановка задачи, разработка математической модели;
2) выбор метода численного решения;
3) разработка алгоритма и структуры данных;
4) проектирование программы;
5) производство окончательного программного продукта;
6) решение задачи на ЭВМ.

Постановка задачи – это точное описание исходных данных, условий задачи и целей ее решения. На этом этапе многие из условий задачи, заданные в форме различных словесных описаний, необходимо выразить на точном (формальном) языке математики. Часто задача программирования задается в математической формулировке, поэтому необходимость в выполнении этапов 1 и 2 отпадает. Для решения достаточно сложных задач этап формализации может потребовать значительных усилий и времени. Среди опытных программистов распространено мнение, что выполнить этап формализации – это значит сделать половину всей работы по созданию программы.

Выбор метода решения тесно связан с постановкой задачи. На первом этапе задача сводится к математической модели, для которой известен метод решения. Метод численного решения сводит решение задачи к последовательности арифметических и логических операций. Однако возможно, что для полученной модели известны несколько методов решения и тогда предстоит выбрать лучший. Можно

усовершенствовать существующий или разработать новый метод решения формализованной задачи. Эта работа по своему характеру является научно–исследовательской и может потребовать значительных усилий. Разработкой и изучением таких методов занимается раздел математики, называемый численным анализом.

При выборе метода надо учитывать требования, предъявляемые постановкой задачи, и возможности его реализации на конкретной ЭВМ: точность решения, быстроту получения результата, требуемые затраты оперативной памяти для хранения исходных и промежуточных данных и результатов.

Алгоритм устанавливает последовательность точно определенных действий, приводящих к решению задачи. При этом последовательность действий может задаваться посредством словесного или графического описаний. Если выбранный для решения задачи численный метод реализован в виде стандартной библиотечной подпрограммы, то алгоритм обычно сводится к описанию и вводу исходных данных, вызову стандартной подпрограммы и выводу результатов на экран или на печать. Более характерен случай, когда стандартные подпрограммы решают лишь какую–то часть задачи. Здесь эффективным подходом является разделение сложной исходной задачи на некоторые подзадачи, реализующиеся отдельными модулями. Определяется общая структура алгоритма, взаимодействие между отдельными модулями, детализируется логика. Этот этап тесно связан со следующим этапом проектирования программы.

Проектирование программы включает в себя несколько подзадач. Во-первых, необходимо выбрать язык программирования. Во-вторых, определить, кто будет использовать разработанное программное обеспечение и каким должен быть интефейс (средство общения с пользователем). В-третьих, решить все вопросы по организации данных. В-четвертых, кодирование, т. е. описание алгоритмов с помощью инструкций выбранного языка программирования. Если задача, для которой разрабатывается алгоритм, сложная, то не следует сразу пытаться разрешить все проблемы. Сложившийся в настоящее время подход к разработке сложных программ состоит в последовательном использовании принципов проектирования сверху вниз, модульного и структурного программирования.

Окончательный программный продукт получается после *отладки и испытания* программы. При программировании и вводе данных с клавиатуры могут быть допущены ошибки. Их обнаружение, лока-

лизацию и устранение выполняют на этапе отладки и испытания (тестирования) программы. Причем логические ошибки могут быть допущены и на этапе постановки задачи, и на этапе алгоритмизации. В этом случае необходимо вернуться к предыдущим этапам. Дорабатывать и улучшать программу можно в течение всего жизненного цикла программного продукта.

Решение задачи на ЭВМ – выполнение всех предусмотренных программой вычислений и вывод результатов расчета на экран дисплея или на печать.

2.2. Способы записи алгоритма. Алгоритм и его свойства

В основе решения любой задачи лежит понятие алгоритма. Под алгоритмом принято понимать точное предписание, определяющее вычислительный процесс, ведущий от варьируемых начальных данных к искомому результату.

При составлении алгоритмов следует учитывать ряд требований, выполнение которых приводит к формированию необходимых свойств.

Алгоритм должен быть однозначным, исключающим произвольность толкования любого из предписаний и заданного порядка исполнения. Это свойство алгоритма называется определенностью.

Реализация вычислительного процесса должна через определенное число шагов привести к выдаче результатов или сообщения о невозможности решения задачи. Это свойство алгоритма называется результативностью.

Решение однотипных задач с различными исходными данными можно осуществлять по одному и тому же алгоритму, что дает возможность создавать типовые программы для решения задач при различных вариантах задания значений исходных данных. Это свойство алгоритма называется массовостью.

Предопределенный алгоритмом вычислительный процесс можно расчленить на отдельные этапы, элементарные операции. Это свойство алгоритма называется дискретностью.

Алгоритмизация – это техника составления алгоритмов и программ для решения задач на ЭВМ.

Изобразительные средства для описания алгоритмов

К изобразительным средствам описания алгоритмов относятся следующие основные способы их представления:

а) словесный – запись на естественном языке;

б) структурно-стилизованный – запись на языке псевдокода;

в) программный – тексты на языках программирования;

г) графический – схемы графических символов.

Словесный способ записи алгоритмов представляет собой описание последовательных этапов обработки данных и задается в произвольном изложении на естественном языке. Способ основан на использовании общепринятых средств общения между людьми и, с точки зрения написания, трудностей для авторов алгоритмов не представляет. Однако для «исполнителей» такие описания алгоритмов часто неприемлемы. Они строго не формализуемы, страдают многословностью записей, допускают неоднозначность толкования отдельных предписаний. Поэтому такой способ описания алгоритмов не имеет широкого распространения.

Пример 1. Записать алгоритм нахождения наибольшего общего делителя двух натуральных чисел (m и n) на естественном языке.

При таком словесном способе содержание алгоритма может быть следующим:

1) если числа равны, то необходимо взять любое из них в качестве ответа, в противном случае – продолжить выполнение алгоритма;

2) определить большее из чисел;

3) заменить большее число разностью большего и меньшего чисел;

4) повторить алгоритм с начала.

Структурно-стилизованный способ записи алгоритмов основан на формализованном представлении предписаний, задаваемых путем использования ограниченного набора типовых синтаксических конструкций. Такие средства описания алгоритмов часто называются псевдокодами. Разновидностью структурно-стилизованного способа описания алгоритмов является известный школьникам алгоритмический язык в русской нотации (АЯРН).

Пример 2. Приведем описанный на АЯРН алгоритм решения задачи об определении принадлежности точки D треугольнику ABC.

алг Определение принадлежности точки треугольнику (дейст xA, yA, xB, yB, xC, yC, xD, yD целое z лит a);

```
арг xA, yA, xB, yB, xC, yC, xD, yD;
рез z,a;
нач
действ S1, S2, S3, S4
вычислить значение S1, равное площади тр–ка ABC;
вычислить значение S2, равное площади тр–ка ABD;
вычислить значение S3, равное площади тр–ка ACD;
вычислить значение S4, равное площади тр–ка CDB;
если S1 = S2+S3+S4
то z := 1,
  a := “точка внутри треугольника”,
иначе z := 0,
  a := “точка вне треугольника”,
все
напечатать значение a:
кон
```

Программный способ записи алгоритмов – это алгоритм, записанный на языке программирования, позволяющем на основе строго определенных правил формировать последовательность предписаний, однозначно отражающих смысл и содержание частей алгоритма с целью их последующего исполнения на ЭВМ.

Пример 3. Программа на языке Бейсик перевода температуры из градусов Цельсия в градусы Фаренгейта. PRINT «Перевод температуры из град. Цельсия в град. Фаренгейта»

```
PRINT «Укажите температуру в град. Цельсия»
INPUT C
6 IF C = 9999 THEN 7
F = C*1.8 +32
PRINT C, F
GOTO 6
7 END
```

Для графического изображения алгоритмов используются *графические символы*. Наиболее распространенными являются блочные символы (блоки), соединяемые линиями передач управления. Существует государственный стандарт на выполнение графической записи блок-схемы алгоритма, в котором указаны перечень условных графических символов, их наименования, форма. Такая запись алгоритма является наиболее наглядной.

Схемы алгоритмов

Использование схем позволяет представить алгоритм в наглядной форме, поэтому они используются наиболее часто.

Вычислительный блок представляет собой прямоугольник, в котором записываются расчетные формулы. Причем формула должна быть записана таким образом, что вычисляемая переменная записывается слева, далее идет знак равенства (в данном случае этот знак называется присваиванием), далее - расчетная формула.

Проверка условия изображается ромбом, внутри которого записывается это условие. В результате проверки выбирается один из двух возможных путей вычислительного процесса.

Если условие выполняется, то есть имеет значение ДА, то следующим выполняется этап по стрелке ДА. Если условие не выполняется, то осуществляется переход по стрелке НЕТ.———•

Начало и конец вычислительного процесса изображаются овалом, в котором записываются слова «Начало» или «Останов».

При решении задач на ЭВМ исходные данные задаются при вводе в машину. *Ввод данных* может осуществляться разными способами, например, с клавиатуры, диска и т. д. Задание численных значений исходных данных будем называть вводом, а фиксацию результатов расчета - выводом. Ввод исходных данных и вывод результатов изображаются параллелограммом. Внутри него пишется слово «Ввод» или «Вывод» и перечисляются переменные, подлежащие вводу или выводу.

Размеры и отображаемые функции устанавливаются ГОСТ 19.701-90.

Рассмотрим создание простой программы: таблица умножения. Компьютер может выполнять сложные вычисления, но, для начала, научимся заставлять его выполнять простые. Как выглядит столбик таблицы умножения, знают все школьники. Создадим программу, которая будет такой столбик выводить.

Внимательно посмотрим на табл. 2.1. Чтобы научиться программировать, такой путь придется проходить каждый раз при создании очередного программистского шедевра.

Таблица 2.1

«Скелет» программы на Паскале

Постановка задачи	Блок-схема	Программа	Комментарий
		Var A, B, C : Integer;	Объявление переменных определенного типа.
Начало программы	Начало	Begin	Вывод на экран приглашения для ввода числа.
Ввод первого сомножителя	Ввод A	Write('Введи первый сомножитель: '); Readln(A);	Считывание числа, введенного с клавиатуры в переменную A.
Присвоить второму 1	B=1	B:=1;	Присвоение переменной B значения, равного 1.
Получить произведение	C=A*B	Repeat C:= A * B;	Цикл вычисления произведения.
Вывести его на экран	Вывод C	Writeln(A,' x ',B, ' = ', C);	Вывод значений сомножителей и произведения на экран.
Увеличить второй на 1	B=B+1	B:=B+1;	Увеличение второго сомножителя.
Проверить, не больше ли он 10	B>10 (Нет / Да)	Until B > 10;	Проверка величины сомножителя.
Выход	Конец	Readln; end.	Ожидание нажатия Enter перед выходом из программы

Рассмотрим основные этапы программы.

Слово Var не является обязательным, но серьезных программ без переменных не бывает.

```
Var
{Оператор Var обозначает начало блока для объявления переменных.}
K : Integer;
{Объявление  переменных}
{То, что в фигурных скобках, программа не обрабатывает, это комментарии для пояснения действий программы}
```

```
Begin
Writeln('Введи K');
{Приглашение ввести значение переменной K}
Readln(K);
{Считывание значения, введенного с клавиатуры, в переменную с именем K}
Writeln('Это K = ', K);
{Вывод на экран значения введенной переменной K}
{Любая команда в Паскале завершается точкой с запятой.}
End.
{Begin и End обозначают начало и конец какого-либо программного блока. End с точкой – конец программы.}
```

В программе на Паскале обязательными элементами являются только Begin и End – с точкой, все остальное – творчество составителя программы. Но перед написанием любой программы нужно определиться с ее базовыми составляющими (табл. 2.2).

Таблица 2.2

Базовые составляющие программы

Состав программы	Выполняемые операции	Способы выполнения
Ввод	Нужно решить: какие данные (цифры, текст, изображения и т.д.) и как попадут в программу	- будут заложены в самой программе; - введены с клавиатуры; - взяты из файла; - другие выриантs
Обработка	Что и как необходимо сделать с исходной информацией, чтобы получить результат	- вычисление по формулам; - шифрование, кодирование, сортировка, поиск; - изменение свойств; - другие действия
Вывод	Что и куда выводить	- на экран; - на принтер; - в файл; - другой вариант

Информацию, с которой работают программы, принято называть данными. И, в самом деле, числа, символы, текст, изображения передаются программе для обработки.

Перед использованием данных программа должна отвести для них место в памяти, а значит, программист должен знать возможности различных типов данных, а программа – их потребности. Вот и настало время рассказать о типах данных.

С какой же информацией-данными может работать Паскаль? С объемными изображениями, физическими объектами. Основные, но не все типы данных, которыми может пользоваться язык программирования Pascal, указаны в табл. 2.3.

Таблица 2.3

Основные типы данных Pascal

Идентификатор	Занимаемое место в памяти	Диапазон значений
Целые числа		
integer	2	-32768..32767
byte	1	0..255
longint	4	-2147483648..2147483647
Вещественные числа		
real	6	$2{,}9\times10^{-39}$ - $1{,}7\times10^{38}$
extended	10	$3{,}4\times10^{-4932}$ - $1{,}1\times10^{4932}$
Логический тип данных		
boolean	1	true, false
Символьный тип данных		
char	1	все символы кода ASCII
Структурированные типы данных		
массив - array	Сумма потребностей всех элементов	-
строка - string	Можно указать, например, string[10]	-
файл - file.	Зависит от определения его структуры	-

Переменные нужно объявлять. Хотя данные, которые использует программа, могут быть неизменяющимися, например: имя, адрес, номер квартиры, чаще используют меняющиеся данные, например:

вклад в банке, остаток на складе, скорость автомобиля и т.д. Некоторой порции данных, которую программа использует как единое целое, дали название: ПЕРЕМЕННАЯ. Сказать проще, для работы с данными программы используют переменные, то есть величины, которые могут менять свое значение. Каждая переменная в любом языке имеет три характеристики: имя, тип, и значение.

1. Имя – это название переменной, по которому к ней будет обращаться программа.

2. Тип указывает на то, какие данные хранятся в переменной, например, символьные или числовые.

3. Значение – это то, что конкретно хранится в данной переменной, например, число 10 или 0,55. А если это символы, то, например, буква А или слово РОССИЯ.

Программа должна знать, какие переменные будут в ней использованы. А для этого в программе выполняют объявление переменных. Посмотрим на пример ниже.

```
Var
N : integer;    c : char;
  S1 : string;
  A : array[1..100] of integer;
```

После Var указываются имена переменных и типы данных, которые они будут хранить. В Паскале, в отличие от BASIC-а, все переменные нужно объявлять явно, то есть нельзя использовать переменную, которой нет в блоке объявлений.

Данные могут быть как изменяющимися, так и постоянными. Неизменяющиеся порции данных называются константами и могут быть определены, например, так:

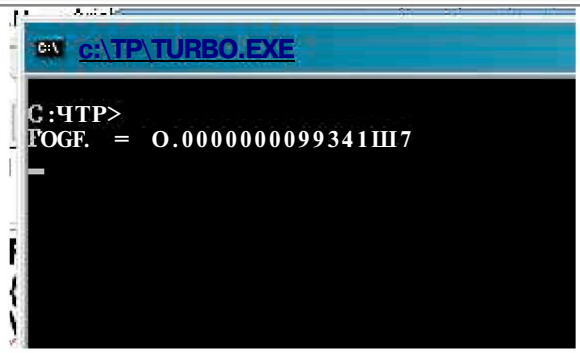
```
Const
Min = 0;
  Max = 100;
  RAZN = (Max - Min) div 2;
 Beta = Chr(255);
 Mess = 'Out of memory';
 Schar = #13;          {Символ без видимого представления}
  Ln10 = 2.30258;
  Number = ['0'..'9'];        {Константа в виде массива}
```

Операция присваивания выглядит так A:=10; Переменной с именем А присваивается значение – число 10.

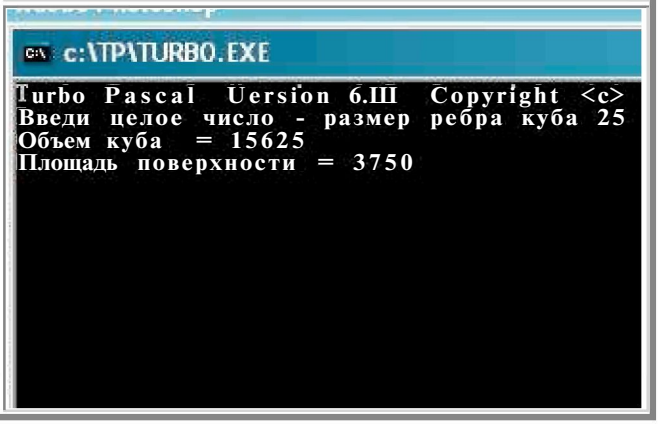
Встретив приведенную выше конструкцию, программа ищет в памяти компьютера отведенный для этой переменной блок, на который будет указывать имя этой переменной – в данном случае, A. В этот блок, размер которого определяется типом этой переменной (в данном случае, числовым) поместит значение этой переменной (в данном случае, число 10). В дальнейшем, когда в программе встретится где-то в выражении или другой конструкции имя этой переменной, будет использоваться не символ A, а значение этой переменной, то есть число 10, на которое и указывает имя этой переменной, а это не что иное, как A.

Простые примеры нужно изучить, чтобы научиться программировать. Рассмотрим первый пример.

```
Program POGRESHNOST;
{N+} {Директива для использования числового сопроцессора}
Var
   x : single;
   y : double;
   POGR : extended;
Begin
  x := 1/3;    y := 1/3;  {x и y, казалось бы, равны, но...}
   POGR := abs(x-y);
  writeln('POGR=',POGR:15:15);
{Вычисляем разность между одинаковыми операциями. Погрешность указывает на разницу в диапазонах разных типов данных}
Readln;  End.
```




```
Program KUB;
Var
A : Integer;   V, S : Longint;
Begin
  Write('Введите целое число – размер ребра куба ');
  Readln(A);
    V:=A*A*A;
```



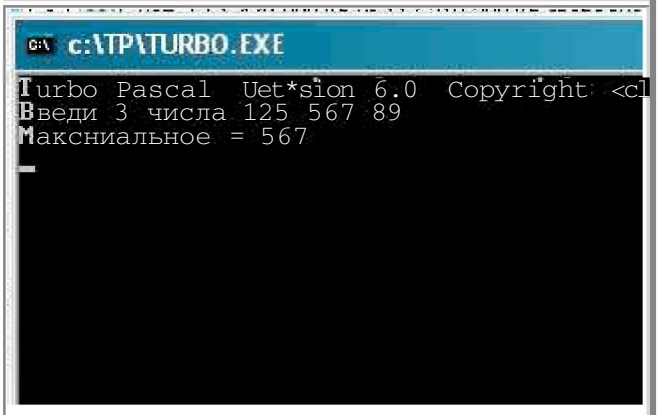
```
   Writeln ('Объем куба  = ',V<);

   S:=6*A*A;
   Writeln ('Площадь поверхности
='S);
Readln;  end. {Конец}
```

```
Uses Crt;  {Вывод звукового сиг-
нала
 Begin
   Sound(600);  {Включение звука,
частота  600Гц}
   Delay(100);  {Задержка 100мс}
   NoSound;  {Выключение звука}
End.
```

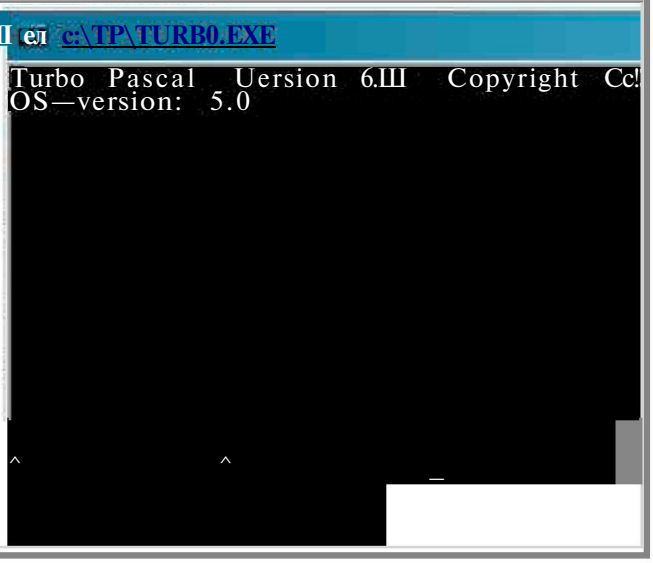
Поэкспериментировав с частотой и длительностью, можно услышать гудки и писки из динамика компьютера

```
Var   {Определение максимально-
го числа}
A, B, C : Integer;
Max : Integer;
Begin
  Write ('Введите 3 числа');
  Readln(A,B,C);
    Max:=A;
      If B>Max then Max:=B;
      If C>Max then Max:=C;
  Writeln('MAX=',Max); <
Readln; End.
```




```
Uses Dos;  {Определить версию
операционной системы}
Var

V:Word;
Begin
V:=DosVersion;
WriteLn('OS-version: ',
Lo(V),'.',Hi(V));
Readln;
End.
```



Как и во всех следующих программах (за очень редким исключением), в рассмотренных выше примерах можно выделить следующие блоки:

– объявление переменных;
– ввод исходных данных;
– обработка или простые вычисления;
– вывод результата на экран.

Так как это простые программы, то после выполнения они исчезают с экрана. Для анализа результатов их работы в конце почти каждой стоит оператор ReadLn; . В работающей программе он просто ждет, что, просмотрев результаты, пользователь нажмет клавишу Enter, и лишь после этого произойдет завершение ее работы.

Построение выражений и встроенные функции. Любая программа что-то вычисляет или что-то обрабатывает. Поэтому без программной конструкции «вычислительное выражение» не обходится практически ни одна из них. Выражения формируются из констант, переменных, функций, знаков операций и круглых скобок по определенным синтаксическим правилам.

Круглые скобки используются для изменения порядка вычисления частей выражения. Выражения без скобок вычисляются в порядке, соогласно следующему приоритету операций:

– вычисление значений функций;
– унарные операции (not,+,-);
– операции типа умножения (*,/,div,mod,and);
– операции типа сложения (+,-, or, xor);
– операции отношения (=, <>, <, >, <=, >=).

Построение вычислительных выражений рассмотрим на примере использования встроенных функций языка Pascal.

```
VAR
X : Integer;
Y : Real;
Begin
  Write('Введите число ');  Readln(X);
  Writeln('Модуль X =', ABS(X));
  Writeln('Синус X=',SIN(X));
  Write('Введите вещественное ');  Readln(Y);
  Writeln('Целая часть Y =', INT(Y));
  Writeln('Корень из Y =', SQRT(Y):4:4);
  Writeln('Случайное число =', RANDOM);
  Writeln('Логарифм Y =', LN(Y));
  Writeln(('Округленное  Y =', ROUND(Y));
  Writeln('Дробная часть Y =', FRAC(Y));
  Writeln('Y без дробной части =', TRUNC(Y));
```

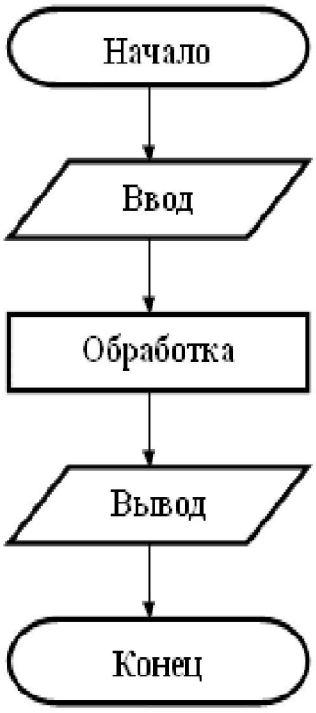
```
Inc(X);
Writeln('X+1=',X);
Dec(X);
Writeln('X-1=',X);
readln;
End.
```

В программе показано использование часто употребляемых функций Паскаля. Все остальные функции могут быть рассмотрены самостоятельно. Достаточно нажать в оболочке Паскаля комбинацию клавиш Shift+F1 и можно увидеть очень подробную подсказку. Почти по каждой функции имеется демонстрационный пример.

Технология алгоритмического программирования, которая используется в Pascal, базируется на методе последовательной детализации алгоритмов. Это означает, что большой и сложный алгоритм (другими словами, порядок выполнения программы) разбивается на отдельные простые блоки, которые должны представлять собой стандартные алгоритмические структуры: Линейную, Ветвление, Циклическую. Для наглядности их представляют в виде блок-схем. Рассмотрим их и примеры самых простых программ, которые эти структуры используют (табл. 2.4).

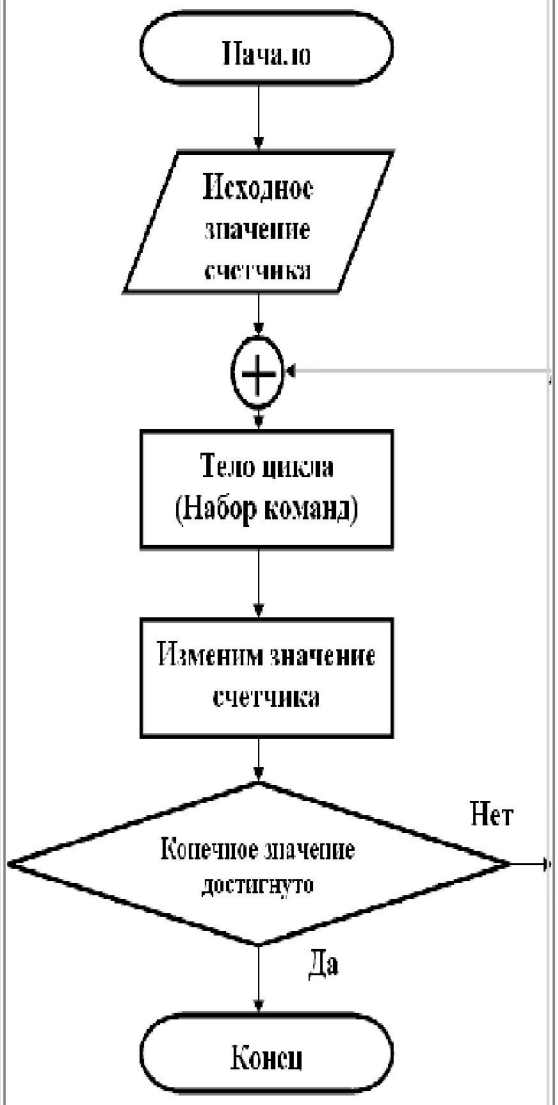
Таблица 2.4

Примеры программной реализации алгоритмических структур

Блок-схема	Алгоритмическая структура
ЛИНЕЙНАЯ (СЛЕДОВАНИЕ)	
	Program Date; {Показ сегодняшней даты и времени} Uses Dos; {Объявление модуля, используемого в программе} Var Year,Month, Day, DayOfWeek : Word; Hour, Minute, Second, Hund : Word; Begin GetDate(Year,Month,Day,DayOfWeek); { Определение даты } Writeln('Сегодня - ', Day, '.', < Month,<'.',Year); {вывод даты } GetTime(Hour,Minute,Second,Hund); {Определение времени } Write('Сейчас - '); WriteLn(Hour,':',Minute,':',Second,'.',Hund); { Вывод времени } End

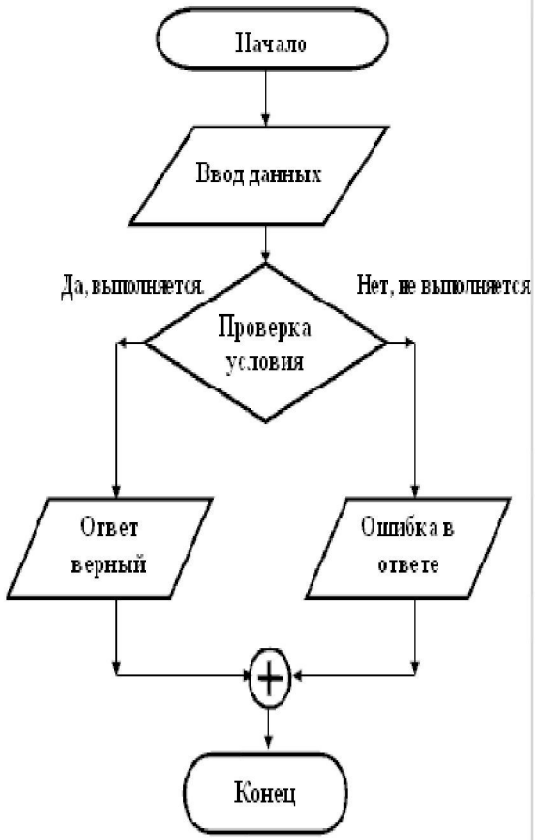
Продолжение табл. 2.4

Блок-схема	Алгоритмическая структура
	Внимание: Write – после вывода не переводит курсор на следующую строку, а Writeln – переводит
ЦИКЛ	

	Program Tumn; {Таблица умножения}
	Var
	
	P1, I : Integer;
	
	Begin
	Write('Введите первый сомножитель: ');
	Readln(P1);
	
	For I:=1 to 10 do {Организация цикла со счетчиком}
	
	Begin
	Writeln(P1,' x ',I,' = ', P1*I);
	end;
	
	Readln;
	end.
	
	Внимание: Внутрь операторных скобок Begin - end; можно помещать несколько выражений. В этом случае все они будут считаться единым блоком.
	В данном случае тело цикла
	Writeln(P1,' x ',I,' = ', P1*I);
	окружено Begin и end; – для демонстрации

Окончание табл. 2.4

Блок-схема	Алгоритмическая структура
ВЕТВЛЕНИЕ	

	Program VETVLENIE; {Программа тестирования знания дат}
	
	Var
	God: Integer;
	
	Begin
	Write('Введите год начала Великой Отечественной войны – ');
	Readln(God);
	
	If God = 1941 then
	Writeln('Ответ верный')
	else
	Writeln('Вы ошиблись');
	
	Readln;
	
	End.

Ветвление в программах. Основные циклические конструкции приведены в табл. 2.5. Формат условного оператора на языке Паскаль:

If <условие> Then <оператор 1> Else <оператор 2>;

При выполнении условия программа обрабатывает первую группу операторов, при невыполнении – вторую:

If <условие> Then Begin <группа операторов 1> end {Внимание! Перед Else точка с запятой не ставятся}

Else Begin < группа операторов 2> end;

Рассмотрим пример. Из двух чисел необходимо выбрать наибольшее.

```
Program Example;
Var A,B,C : Real;  {A,B – для хранения аргументов, C – результат}
```

```
Begin
  Writeln('Введите два числа');
  Readln(A,B);  {Вводим аргументы с клавиатуры}

If A>B Then C:=A Else C:=B;    {Если A>B, то результат – A, иначе
результат – B}

Writeln('Наибольшее = ',C);  {Выводим результат на экран}
End.
```

Таблица 2.5

Использование циклических алгоритмов в программах

Описание конструкции	Пример

Цикл с параметром (со счетчиком) FOR
 	
FOR n:=1 TO 10 DO
BEGIN
ПРЫЖОК (10 РАЗ)
END;
 	
Телом цикла называется последовательность операторов, которая может выполняться многократно (ПРЫЖОК)
	Var n,x,y : longint; {Десять четных чисел и их квадраты}
	Begin
	x:=2;
	
	FOR n:=1 to 10 do
	  Begin
	  y := x*x;
	  writeln (‘x и y = ’, x, ’ – ‘, y);
	  x:=x+2;
	  End<;
	Readln;
	End.

Цикл с предусловием WHILE
 	
WHILE силы есть? DO
BEGIN
ПРЫЖОК
END;
 	
Если условие истинно, то выполняются операторы циклической части. (Прыгаем, пока есть силы)
	Var x,y : longint; {Десять четных чисел и их квадраты}
	Begin
	x:=2;
	WHILE x<=10 do
	  Begin
	  y := x*x;
	  writeln (‘x и y = ’, x, ’ – ‘, y);
	  x:=x+2
	  end;
	Readln;
	end.

Окончание табл. 2.5

Описание конструкции	Пример

Цикл с постусловием REPEAT
 	
REPEAT
 	
ПРЫЖОК
 	
UNTIL УСТАЛ?
 	
Выполняются до тех пор, пока условие ложно. Как только условие становиться истинным, цикл прекращается. (Прыгаем, пока не устанем)
	Var x,y : longint; {Десять четных чисел и их квадраты}
	Begin
	x:=2;
	REPEAT
	y := x*x;
	writeln (‘x и y = ’, x, ’ – ‘, y);
	x:= x+2
	UNTIL x>10;
	Readln;
	end.

В реальных программах такие алгоритмические структуры, как Линейная, Ветвление и Цикл используются очень активно, и без четкого понимания их функциональности сколько-нибудь серьезные программы создать невозможно.

Работа со строками

Объявление переменной строчного типа.

```
Var
S : String [20];  {Квадратные скобки со значением  указывают максимальную длину строки}
```

```
Var st : String;
Begin
  Write('Введите строку');  Readln(st);
  if st[5] = ‘A’ then Writeln('Пятый символ в строке это A');
end. {Обращаемся к символу, как к элементу массива – строка}
```

Символ с индексом 0 является байтом, указывающим длину строки. ORD(st[0]) – это текущая длина строки, но можно получить и так: LENGTH(st).

К строкам можно применять операцию «+» – сцепление, например:
st := ‘a’ + ‘b’; st := st + ‘c’; {st содержит «abc»}

Примеры использования стандартных процедур и функции работы со строками приведены в табл. 2.6; другие примеры использования строк проиллюстрированы в помощи к Pascal (Shift+F1).

Таблица 2.6

Стандартные процедуры и функции работы со строками

C0NCAT(S1 [,S2, ... , SN])	возвращает строку, сцепление строк-параметров S1, S2, ..., SN
COPY(ST, I, C)	копирует из строки ST C символов, начиная с символа с номером I
DELETE (ST, I, C)	процедура; удаляет C символов из строки ST, начиная с символа с I
INSERT (SB, ST, I)	процедура; вставляет подстроку SB в строку ST, начиная с символа I
POS (SB, ST)	отыскивает в строке ST первое вхождение подстроки SB и возвращает номер позиции, если подстрока не найдена, возвращается ноль
Str(X: арифметическое выражение; var st: string)	процедура преобразует численное выражение X в его строковое представление и помещает результат в st
Val(st: string; x: числовая переменная; var code: integer)	процедура преобразует строковую запись числа, содержащуюся в st, в числовое представление
UpCase(Переменная)	позволяет преобразовывать любой символ из строчного в прописной

Процедуры и функции. Вспомогательный алгоритм может быть оформлен как подпрограмма (процедура). Затем эту процедуру можно вызывать по имени из любого места программы.

В общем виде оформление процедуры можно представить так, как это показано в табл. 2.7.

Пример создания и использования функции из помощи к Pascal:

```
Program  Func;
        function IntToStr(i: Longint): string;
         { Функция конвертирует число в строку }
            Var
```

```
            string[11];
      begin
        Str(i, s);

IntToStr := s;
      end;
    WriteLn(IntToStr(-5322));
end.
```

Таблица 2.7

Оформление процедуры в Pascal

	Описание конструкции		Пример

	VAR {Объявление переменных программы}
	x : Integer; s : String;
	
	Procedure MyProc(a : Integer; b : String); {Заголовок Процедуры. a и b – фиктивные параметры, работают внутри процедуры}
	Begin
	. . . . .{Тело Процедуры}
	End;
	
	BEGIN
	. . . . .{x, s – фактические параметры}
	MyProc(x,s); {Вызов Процедуры}
	....
	END.
			Program HelloAll;
			
			Procedure
			Hello(Name: string);
			Begin
			Writeln(‘Привет, ‘, Name, ‘!’);
			Writeln(Name, ‘ как дела?’);
			Writeln;
			End;
			
			Begin
			Hello(‘Катя’);
			Hello(‘Андрей’);
			Hello(‘Лена’);
			End.

Пояснение. Зачем нужно конвертирование? Если в программе

```
VAR S : String;  K : Integer;  N : Integer;
```

то если даже Writeln(S,' – ',K); выдает на экран 5 – 7, то все равно нельзя сделать N := K + S, потому что N и S – разные типы данных, и S нужно сначала попытаться конвертировать в число.

Внимание. Функция внутри себя использует так называемые формальные переменные, которые не видны из основной программы. А вот когда она (функция) вызывается, то получает для обработки значения фактических переменных, которые объявлены и используются вне функции.

Массивы. Если в программе требуется использовать 10 или 100, или 1000 однотипных переменных, то они могут быть определены для использования в виде массива.

Var – имя переменной-массива : Array [Диапазон индексов] Of Тип элементов;

Примеры описания массивов:

```
Var
S, BB : Array [1..40] Of Real;
N : Array ['A'..'Z'] Of Integer;
R : Array [-20..20] Of Word;
T : Array [1..40] Of Real;
```

Массив – это переменная с индексом (номером). Имя у всех переменных одно, но при обращении к каждой отдельной используется соответствующий номер (индекс).

Правила заполнения массива

```
Program M1;
Var
  A : Array [1..10] Of Integer;
Begin
  A[1]:=7;  {Заполняем массив значениями }
  A[2]:=32;
  A[3]:=-70;
  ..... {Трудоемкая задача?}
  A[10]:=56;
  Writeln(A[1],A[2],A[3], ?,A[10])
End.
```

```
Program M2;
Var A : Array [1..10] Of Integer;
  I : Integer;

Begin
  For I:=1 To 10
Do {Организуем цикл }
  Readln(A[I]);  {Вводим A[I] }
  For I:=1 to 10
Do {Выводим массив}
  Writeln(A[I],'-',I,'-й');

End.
```

```
Program M3;
Const N=40; {Константа N будет содержать количество элементов массива}

Var  A : Array [1..N] Of Integer;
  I : Integer;

Begin
  For I:=1 To N Do
    Begin
      A[I]:= Trunc(Random(256));
      Write(A[I],'-  *-');
    End;
End.  {Массив из случайных чисел}
```

Двумерные массивы. Массивы могут быть многомерными. Двумерный массив принято называть МАТРИЦЕЙ.

Пример объявления двумерного массива:

Var A : Array [1..20,1..30] Of Integer;

Сначала указывается номер строки, а затем – номер столбца.

Рассмотрим пример работы с матрицей – программу, вычисляющую сумму элементов, находящихся на диагонали. Здесь значения элементов матрицы задает сам компьютер в виде случайных чисел.

```
Program M4;
 Var A : Array[1..10,1..10] Of Integer;
   I, K : Byte;
   S : Integer;

Begin
 S:=0;
 For I:=1 To 10 Do
   Begin  {Блок заполнения строки<}
    For K:=1 To 10 Do
      Begin  {Блок заполнения строки элементами}
       A[I,K]:=Trunc(Random*100)+1;
       Write(A[I,K]:6);
         If K=I Then S:=S+A[I,K];  {Суммирование диагональных
элементов}
      End;
       Writeln
   End;
 Writeln('Сумма элементов гл. диагонали = ',S)
End.
```

Важно! Матричные вычисления, то есть различные операции с матрицами встречаются в инженерных и научных расчетах очень часто. Трудно назвать сферу, где бы матрицы не использовались так или иначе, начиная с электротехники и электроники и заканчивая статистикой и биологией. Понимание механизмов работы с матрицами многим может пригодиться в жизни.

Если программа имеет сложную структуру, например, много Begin-ов и End-ов, нужно располагать их так, чтобы Begin и End, относящиеся к одному блоку, имели одинаковый отступ. Тогда нельзя запутаться в их расстановке.

Работа с файлами. Данные для долговременного хранения помещаются на диски в виде файлов. Файл, как и любая переменная, должен быть объявлен, например, так:

File of Тип данных.

Это означает, что в файл на диск будут последовательно записываться порции данных указанного типа.

Файл, состоящий из символьных строк, объявляется как File: text; где File – имя файловой переменной.

Рассмотрим пример с подробными комментариями.

Uses Crt;
Var
ch : char;
F : text;
Name, S : String;
 	
Begin
Writeln('‘Список класса.’');
Assign(F, 'spis.txt');
Rewrite(F);
repeat
Write('‘Введите имя ученика: "); Readln(Name);
Writeln(F, Name);
Writeln('Дальше – любая клавиша. Esc- Выход.');
ch:=readkey;
until ch=#27;
 	
Close(F);
 	
Assign(F, 'spis.txt');
Reset(F);
while not Eof(F) do
begin
Readln(F,S);
Writeln(S);
end;
Close(F);
readln;
End.
	Дополнительные функции, расширяющие возможности Паскаля, располагаются в модулях, которые перед использованием нужно объявлять.
	Назначается имя файлу, которое будет являться именем файла на диске.
	Перезаписывается дисковый файл. Те данные, что хранились на диске под этим именем, перестают существовать.
	Файл заполняется списком класса, пока не будет нажата клавиша Esc.
	Записывается имя ученика в файл.
	
	Закрывается запись в файл. Операционная система фиксирует размер файла и разрешает работать с ним другим программам.
	
	Открываем доступ к файлу для чтения.
	Операционная система предупреждается, что из данного файла будут читать, пока не встретим конец – End of file (Eof).
	
	Вывод на экран.
	
	Закрывать доступ при чтении не обязательно

Важно! Когда в программе собираются читать файл, то должны точно знать, данные какого типа там хранятся.

Графика на Pascal. Овладение основами работы с графикой на Паскале нельзя недооценивать. Рассмотрим только самые азы применения графических возможностей в программах.

Перед использованием программной графики нужно перевести работу видеосистемы (видеоконтроллера и монитора) в соответствующий графический режим.

Группа команд включения графического режима имеет вид:

```
Gd := Detect;  InitGraph(Gd, Gm, '');
if GraphResult <> grOk then Halt(1);
Использование графического режима ...
CloseGraph;
```

Графические режимы различаются разрешением (максимально отображаемым количеством точек по горизонтали и вертикали) экрана и количеством возможных цветов.

Рассмотрим полный пример программы вывода цветных точек до нажатия любой клавиши:

```
uses
Crt, Graph;  {Объявление дополнительных модулей, которые будет
использовать программа}

Var                              {Объявление необходимых переменных}
Gd, Gm : Integer;
Color : Word;

Begin {Начало программы}
  Gd := Detect;  InitGraph(Gd, Gm, '');  {Определение и включение гра-
фического режима}
  if GraphResult <> grOk then Halt(1);

  Randomize;              {Перенастройка генератора случайных чисел}
repeat
  Color := Random(15);              {Выбор случайного цвета}
  PutPixel(Random(640), Random(480), Color);  { Цветное «звездное»
небо}
  Delay(300);                       {Задержка времени}
until KeyPressed;                   {Ожидание нажатия любой клавиши}
```

```
CloseGraph;                    {Закрытие графического режима}
end.                           {Закрытие программы}
```

Научимся рисовать различные геометрические фигуры.

```
      Uses Crt, Graph;
const
Gray50 : FillPatternType = ($AA, $55, $AA, $55, $AA, $55, $AA, $55);
var
  Gd, Gm : Integer;
  Color : Word;
begin
  Gd := Detect;  InitGraph(Gd, Gm, '');
  if GraphResult <> grOk then Halt(1);
  Randomize;

repeat
  SetColor(Random(GetMaxColor)+1);
  Line(Random(400), Random(400), Random(400), Random(400));
       {Линии}

  Circle(Random(500), Random(500), Random(200));
                  {Окружности}

  SetFillPattern(Gray50, Random(15));
  Bar(Random(400), Random(400), Random(400), Random(400));
       {Закрашенные прямоугольники}

  delay(9000);
until  KeyPressed;
CloseGraph;
end.
```

Внимание! Примеры создания графики построены по принципу программ хранителей экрана – выводят меняющиеся изображения до нажатия любой клавиши.

Примеры программ на Паскале

Ниже представлены примеры простых программ на языке программирования Pascal. В процессе их разбора те, кто начинает учить-

ся программировать, смогут усвоить некоторое количество простых приемов, которые можно использовать в дальнейшем.

Описание к приведенным программам дается после демонстрации каждого исходника.

1. Из простой в заглавную.
2. Число в степени.
3. Число пробелов в строке.
4. Цикл с предусловием.
5. Нахождение минимального числа.
6. Заполнение и вывод двумерного массива.
7. Лохотрон – найди шарик.
8. Корни квадратного уравнения – 1.
9. Корни квадратного уравнения – 2.
10. Количество цифр в числе.
11. ASCII – код нажатой клавиши.
12. Контроль ввода нуля.
13. Запись в файл.
14. Поиск файла на диске.
15. Контроль ввода.
16. Размер диска.
17. Десятки и единицы.
18. Количество положительных чисел.
19. Оператор Case.
20. Подача сигнала.
21. Догнал ли Ахилл черепаху.

1. Из простой в заглавную. Программа демонстрирует перевод символа строки из обычного представления в заглавное. Часто такой прием требуется для анализа введенного символа, если важно проверить именно символ, а не его регистр.

```
Program Str1;
VAR Words: STRING;
BEGIN
  Words := 'microsoft';
   Words[1] := UpCase (Words[1]);
   WriteLn(Words);  {Выводится Microsoft}
END.
```

2. Число в степени. В Паскале нет встроенной функции возведения в степень. Программа показывает, как это можно исправить.

Можно попробовать реализовать такую функцию для случая, когда показатель степени не является целым числом.

```
        Program  Stepen_chisla;
        Var
  Z, A : Real;  M : integer;
        Function Step (N: integer;  X:real): real;
        Var
  I: integer;  Y: Real;
        Begin
    I:=1; Y:=1;
            While I<=N do
           Begin
              Y:=Y*X;  I:=I+1;
            End;
             Step:=Y;
           End;  {Конец функции}
        Begin
   Write('Введите степень и возводимое число');  Readln(Z,M);
          F:=Step(M,Z);
          Writeln(Z, ' в степени', M, '=',F);
        End.
```

3. Число пробелов в строке. Программа демонстрирует приемы работы со строками: поиск символа в строке, контроль длины введенной строки. Здесь же реализована структура цикла с постусловием и применение функции, созданной пользователем.

```
         program  CountSpacesInString;
        var
 str: string;
        function CountSpaces (s: string): integer;
        var
   i, count: integer;
        begin
  count := 0;
          for i:=1 to length(s) do
          if s[i]=' ' then
          count := count+1;
```

```
        CountSpaces := count;
      end;
{основная программа}
begin
  writeln ('Программа подсчитывает количество пробелов '+'во введенной строке');
   repeat
          writeln('Введите исходную непустую строку:');
          readln(str);
           if (length(str)<1) then
             writeln('Исходная строка должна быть непустой');
        until length(str)>0;
      writeln('В строке "',str,'" ',CountSpaces(str),' пробелов.');
      readln;
      end.
```

4. Цикл с предусловием. В приведенной программе циклически будет вычисляться произведение двух введенных чисел. Для продолжения работы программы необходимо периодически на ее вопрос отвечать утвердительным символом Д или д. Паскаль выводит вещественные числа в так называемой экспоненцмальной форме. Для их обычного представления необходимо после выводимой переменной указать параметры вывода в виде двух чисел после двоеточий.

```
      program  cycle_while;
      Var
 x,y,sum:real;   otv:char;
      begin
 sum:=0;
       otv:='Д';
         while (otv='Д') or (otv='д') do
          begin
            write('Введите числа x,y > 0 ');
           readln(x,y);
           writeln('Их произведение = ',x*y:8:3);
         sum:=sum+x+y;
          write('Завершить программу (Д/Н)? ');
         readln(otv);
         end;
```

```
    writeln('Общая сумма = ',sum:8:3);
    readln
  end.
```

5. Нахождение минимального числа из введенных. Простая программа демонстрирует цикл со счетчиком и работу условных операторов. В начале анализа в качестве минимального задано максимально возможное из целых положительных целочисленного типа.

```
Program MIN;
Var
  A, I : Integer;
  Min : Integer;

Begin
  Min:=32767;
    For I:=1 to 10 do
      Begin
        Write('Vvedi chislo ');
        Readln(A);
          If A < Min then Min:=A;
      end;
  Writeln('MIN=',Min);
  Readln;
end.
```

Попробуйте реализовать этот алгоритм по-другому.

6. Заполнение и вывод двумерного массива. Демонстрация ручного заполнения двумерного массива с последующим выводом его на экран.

```
Program MASS-DV;
var
 mas:array[1..5,1..5] of integer;  {Объявление двумерного массива}
 i ,j:integer;
  begin
{Ввод значений элементов массива}
  for i:=1 to 5 do

 for j:=1 to 5 do readln(mas[i,j]);
```

```
{Вывод значений элементов массива}
  for i:=1 to 5 do
    begin
      for j:=1 to 5 do
      write(' ',mas[i,j]);
      writeln;
    end;
end.
```

7. Лохотрон – найди шарик. Программа демонстрирует вывод в определенную точку экрана, перемещая в нее курсор процедурой GoToXY из модуля CRT.

```
Uses Crt;
Var
   A, C, B : Integer;

Begin
  clrscr;
  Randomize;
  gotoXY(20,10);  Writeln('Где шарик? Введите номер стакана...');
  A:=Random(99);
     If A<=33 then B:=1 Else If A>66 then B:=3 else B:=2;
       GotoXY(20,11);  Writeln(' _ _ _');
       GotoXY(20,12); Writeln('/\/\/\');
       GotoXY(20,13);  Writeln(' 1 2 3');
       Readln(C);
If C=B then Write('Вы угадали!!!') else Write('Вы ошиблись!');
  GotoXY(20,11);  Writeln(' ');
  GotoXY(20,12);   Writeln('\_/ \_/ \_/');
  GotoXY(17+4*B,12);  Write('O');

Readln; end.
```

8. Корни квадратного уравнения – 1. Классическая программа вычисления корней квадратного уравнения с контролем дискриминанта. Необходимо обратить внимание, как в Паскале формируются вычислительные выражения.

```
Program Sq1;

Var A, B, C, D, X1, X2 : Real;

Begin
  Writeln ('Введите коэффициенты квадратного уравнения');
  Readln (A,B,C);
  D:=B*B-4*A*C;
    If D<0 Then Writeln ('Корней нет! ')
    Else
      Begin
        X1:=(-B+SQRT(D))/2/A;
        X2:=(-B-SQRT(D))/2/A;
        Writeln ('X1=', X1:8:3, ' X2=',X2:8:3)
      End;
End.
```

9. Корни квадратного уравнения – 2. В программе проанализированы ситуации, когда какой-либо из коэффициентов равен 0. Необходимо обратить внимание на использование пустого оператора Begin End; иногда без него никак не обойтись.

```
Program Sq2;
Var A, B, C, D, X, X1, X2 : Real;
Begin
  Writeln ('Введите коэффициенты уравнения (A, B, C) ');
    If A=0 Then
    If B=0 Then
    If C=0 Then Writeln('X – любое число')
      Else Writeln('Корней нет! ')
      Else Begin X:=-C/B;  Writeln('X=',X:8:3) End
      Else
        Begin End;

End.
```

10. Количество цифр в числе. Простая программа определяет количество цифр в числе. Используется алгоритмическая структура «цикл с предусловием».

```
Program  KolCifr;
Var
  I, S, N : Longint;
Begin
  Writeln('Введите целое');  Readln(I);
  N:=1;
     While I > 10 DO
       BEGIN
         I:=I DIV 10;
         Inc(N);
       end;
Write('Количество цифр = ',N);
Readln;  end.
```

11. ASCII – код нажатой клавиши. Какому числу в реальности соответствует каждый из символов клавиатуры во время кодирования? В программах довольно часто приходится анализировать нажатие определенных клавиш. Данная программа позволяет выяснить, что чему соответствует.

```
Uses crt;
Var ch : char;
Begin
  clrscr;  {Очистка экрана}
  repeat
     ch:=readkey;  {Ожидание нажатия клавиши}
     WriteLn('Клавиша и ее ASCII-код: ', ch, '-', ord(ch));  {Вывод ASCII
– кода}
  until ch=#27;  {27 - ASCII – код клавиши Esc}
End.
```

12. Контроль ввода нуля. Программа закончит свою работу, если специально или по ошибре ввели нуль в качестве требуемого числа. Здесь продемонстрировано назначение меток (Label) и применение оператора GoTo.

```
Program KONTROL;
Label  A;
```

```
Var I : Integer;

Begin
 A:
  Write(‘Введите число. 0 – выход из программы’);  Readln(I);
  Writeln(‘Вы ввели число = ’, I);
  IF I <> 0 then Goto A;
End.
```

13. Запись в файл. В реальных программах сохранение результатов в файл происходит очень часто. В данной программе продемонстрирован этот механизм. Нужно поэкспериментировать с записью чисел, а затем попробовать их прочитать и использовать в программе. Практически все школьные олимпиады по программированию используют именно такой способ ввода и вывода данных.

```
Program File;
var
  f: file; {Файловая переменная f}
begin
  assign(f,'test.txt');  {Назначение файловой переменной f имени файла
test.txt}
  rewrite(f);  {Создание файла и открытие его для записи}
  writeln(f,'Запись');  {Запись информации в файл}
  close(f);  {Закрытие файла}
end.
```

14. Поиск файла на диске. А есть ли вообще такой файл на этом диске? Это отнюдь не праздный вопрос, которым задаются многие пользователи. Программа демонстрирует поиск группы файлов с расширением *.exe (программ). Стоит обратить внимание, как происходит обращение к полю переменной, имеющей тип «запись» (через точку).

```
Uses Dos;
Var
  S:SearchRec;
Begin
   FindFirst('*.exe',AnyFile,S);
```

```
    While DosError=0 do
      Begin
        WriteLn(S.Name);
        FindNext(S);
      End;
  End.
```

15. Контроль ввода. В простых программах ошибки при вводе данных могут приводить к неожиданному результату в работе. Все зависит от того, как на эту ошибку прореагирует операционная система. Можно проконтролировать появление ошибок самим, предупредив об этом операционную систему.

```
Program Err;
uses crt;
 var i:integer;   flag:boolean;
begin
 {$I-}{Отключение контроля ошибок ввода – вывода}
  repeat
   readln(i); {Ввод значения}
    if IoResult=0 then flag:=true {Если нет ошибки, то выходим из
цикла}
    else
      begin
        gotoxy(wherex,wherey-1); {Перевод курсора на строку вверх}
        delline;  {Удаление строки}
     end;
  until flag;
{$I+}{Включение контроля ошибок ввода – вывода}
end.
```

16. Размер диска. Раньше именно с помощью таких простых функций можно было все узнать о диске. В современных компьютерах это может получиться некорректно. Необходимо вспомнить о количестве байт пространства диска и диапазоне чисел среды Паскаль.

```
Program DiskSize;
Uses Dos;
Begin
```

```
  WriteLn('DiskSize = ',DiskSize(0) div 1024 div 1024,' Mb');  { Объем диска }
  WriteLn('DiskFree = ',DiskFree(0) div 1024 div 1024,' Mb');  { Объем свободного пространства на диске }
End.
```

17. Десятки и единицы. Показано использование операций целочисленного деления и взятия остатка от деления. В практике программистов понимание сути этих действий требуется довольно часто.

```
Program DES-ED;
Var
  K, Des, Ed : Integer;

Begin
  Write('Введите десятичное число ');  Readln(K);

  Des:= K DIV 10;
  Ed:= K MOD 10;

  Writeln('Количество десятков в числе =', Des);
  Writeln('Количество единиц в числе =', Ed);

Readln; End.
```

18. Количество положительных чисел. Подсчет количества элементов с определенным признаком – довольно частая программистская задача.

```
program  cycle_for;
var i,kn : byte;  x:real;
begin
  kn:=0;
    for i:=1 to 10 do
      begin
        writeln('Введите ',i,' число: ');
        readln(x);
      if x>0 then kn:=kn+1 {Увеличение кол-ва на 1}
end;
```

```
  writeln('Вы ввели ',kn,' положительных чисел.');
readln
end.
```

19. Оператор Case. Если выбор способа реакции программы на определенное условие очень разнообразен, применяется оператор Case (выбор). Реализовать такое ветвление оператором IF было бы затруднительно. Следует обратить внимание, что условия выбора можно задавать через запятую или диапазоном. А заканчиваться Case должен End-ом.

```
Program Vibor;
var  i:integer;
begin
  write('Введите целое число: ');
  readln(i);
    case i of
     0,2,4,6,8 : writeln('Четная цифра');
     1,3,5,7,9 : writeln('Нечетная цифра');
     10...100 : writeln('Число от 10 до 100);
    else writeln('Число либо отрицательное, либо > 100');
end;
  readln;
end.
```

20. Подача сигнала. Программа для напоминания звуком об ошибке.

```
program BipProc;
Var N : Integer;
  Procedure Bip;
    Begin
      Sound(600);  {Включение звука, частота 600Гц}
      Delay(100);  {Задержка 100мс}
      NoSound;  {Выключение звука}
    End.
Begin
  Write('Введите ненулевое число');  Readln(N);
    If N=0 then
```

```
            Begin Writeln('Вы ошиблись'); Bip; end;
    End.
```

21. Догнал ли Ахилл черепаху. Реализация греческого философского опуса о том, догонит ли Ахилл черепаху, если она находится от него в 100 шагах, а скорость Ахилла в 10 раз больше.

```
Program AHIL;
CONST
  Vch=100/3600;   {Скорость черепахи}
   Va=1000/3600;  {Скорость Ахилла}

VAR
  Sa, Sch : Real;

Begin
  Sa:=0; Sch:=100;
    repeat
       Sch:=Sch+Vch*1;
       Sa:=Sa+Va*1;
  Until Sa > Sch;  {Проверка пути, пройденного Ахиллом}

  Writeln('Ахилл догнал чрепаху на расстоянии =', Sch:8:7, ' - шагов.');
Readln;
end.
```

2.3. Языки семейства Си

Сравнительный анализ компиляторов C++

К сожалению, выбор компилятора часто обусловлен, опять-таки, идеологией и соображениями вроде «его все используют». Конечно, среда разработки Microsoft Visual C++ несколько более удобна, чем у портированного gcc – но это ведь вовсе не значит, что релиз своего продукта нужно компилировать с использованием MSVC++. Можно использовать оболочку, компилировать промежуточные версии на MSVC++ (кстати, время компиляции у него гораздо меньше, чем у gcc), но релиз можно собрать с использованием другого компилятора, например, от Intel. И, в зависимости от компилятора, можно полу-

чить прирост в производительности на 10 %. Но какой «правильный» компилятор выбрать, чтобы он сгенерировал максимально быстрый код? К сожалению, однозначного ответа на этот вопрос нет – одни компиляторы лучше оптимизируют виртуальные вызовы, другие – лучше работают с памятью.

Попробуем определить, кто в чем силен среди компиляторов для платформы Wintel (x86-процессор + Win32 ОС). Сравним компиляторы Microsoft Visual C++ 6.0, Intel C++ Compiler 4.5, Borland Builder 6.0, MinGW (портированный gcc) 3.2.

Порядок тестирования

Как проверить, насколько эффективен код, генерирующий компилятор? Очень просто: нужно выбрать несколько наиболее часто употребляемых конструкций языка и алгоритмов и измерить время их выполнения после компиляции различными компиляторами. Для более точного определения времени необходимо набрать статистику и выполнить каждую конструкцию некоторое количество раз.

Вроде бы все просто, но тут начинают возникать определенные проблемы. Провести тестирование некоторых конструкций (например, обращение к полю объекта) не удастся из-за оптимизации на уровне компилятора: строки типа

```
for (unsigned i=0; i<10000000; i++) dummy = obj->dummyField;
```

все компиляторы просто выбросили из конечного бинарного кода.

Вторым неприятным моментом является то, что в результаты всех тестов неявно вошло время выполнения самого цикла FOR, в котором происходит набор статистики. В некоторых реализациях оно может быть очень даже существенным (например, два такта на одну итерацию пустого for для gcc). Измерить «чистое» время выполнения пустого цикла удалось не для всех компиляторов – VC++ и Intel Compiler выполняют достаточно хорошую «раскрутку» кода и исключают из конечного кода все пустые циклы, inline-вызовы пустых методов и т.д. Даже конструкцию вида

```
for (unsigned i=0; i<16; i++) dummy++;
```

VC++ реализовал как dummy += 16; .

Наличие такой нетривиальной низкоуровневой оптимизации наводит на мысль о необходимости анализа сгенерированного кода на уровне ассемблера. Во-первых, это позволит убедиться в том, что действительно измерили то, что хотели измерить (а не оптимизированный компилятором пустой цикл, из которого он выбросил все

«лишние» вызовы). Во-вторых – более точно определить, чей код наиболее оптимален, что существенно дополнит картину тестирования.

Было проведено тестирование [13] времени компиляции работающего исходника с целью определения, у какого же из компиляторов время компиляции наименьшее.

Для измерения времени выполнения тестов использовался счетчик машинных тактов, доступный по команде процессора RDTSC, что позволило не только сравнить время выполнения большого количества однотипных операций, но и получить приближенное время выполнения операции в тактах (вторая величина является более показательной и удобной для сравнения). Все тесты проводились на Pentium III (700 МГц), параметры компиляции были установлены в «-O2 -6» (оптимизация по скорости + оптимизация под набор команд Pentium Pro). Кроме того, для Borland Builder была добавлена опция --fast-call – передача параметров через регистры (Intel Compiler, MSVC++ и gcc автоматически используют передачу параметров через регистры при использовании оптимизации по скорости).

Тестирование было разделено на несколько независимых частей. Первая – тестирование скорости работы основных конструкций языка (виртуальные вызовы, прямые вызовы и т.д.). Вторая – тестирование скорости работы STL. Третья – тестирование менеджера памяти, поставляемого вместе с компилятором. Четвертая – разбор ассемблерного кода таких базовых операций, как вызов функции и построения цикла. Пятая – сравнение времени компиляции и размера выполняемого файла.

Тестирование скорости работы основных конструкций языка. Первый тест очень прост, он заключается в измерении скорости прямого вызова (member call), виртуального вызова (virtual call), вызова статик-метода (данная операция полностью аналогична вызову обыкновенной функции), создания объекта и удаления объекта с виртуальным деструктором (create object), создания/удаления объекта с inline-конструктором и деструктором (create inline object), создания template'ного объекта (create template object). Результаты теста приведены в табл. 2.8.

Таблица 2.8

Результаты тестирования скорости работы основных конструкций языка

	VC++	Intel Compiler	Bulder C++	MinGW (gcc)
virtual call	140 (9)	134 (9)	139 (9)	183 (12)
member call	124 (8)	134 (9)	103 (7)	154 (10)
static call	121 (8)	113 (7)	109 (7)	118 (8)
create object	606 (424)	663 (443)	459 (321)	619 (433)
create inline object	579 (405)	600 (420)	343 (240)	590 (413)
create temlate object	580 (405)	599 (419)	349 (244)	579 (405)

Первая цифра – это полное время, затраченное на тест (в миллисекундах); цифра в скобках – количество тактов на одну команду.

Результаты получились очень интересными: первое место занял Borland Builder, а вот gcc на вызове методов, особенно виртуальных, показал существенное отставание. По всей видимости – из-за бурного развития COM'a, где все вызовы виртуальные, разработчикам «родных» компиляторов под Win32 пришлось максимально оптимизировать эти типы вызовов. Другим интересным фактом является то, что хорошо оптимизировать создание объекта с inline-конструктором и деструктором смог, опять-таки, только Builder.

Конечно, у MSVC++ также наблюдается небольшой прирост производительности, но объясняется это тем, что MSVC++ очень хорошо «раскручивает» код и все заглушки просто выбрасывает. То есть в тесте с inline-вызовами MSVC++ определил, что вызываемый метод является пустым, и исключил его вызов. После исключения вызова пустого метода у него остался пустой цикл, который компилятор также выбросил.

Borland же в случае использования inline-конструктора делает inline не только вызов метода «Конструктор», но и выделение памяти под объект. То же самое делает Builder относительно деструктора. Любопытно отметить, что с шаблонами Builder работает точно так же, как с inline-методами, чего совершенно не скажешь о других компиляторах.

Тестирование STL. STL, как известно, входит в ISO стандарт C++ и содержит очень много полезного и превосходно реализованного кода, использование которого существенно облегчает жизнь программистам. Конечно, MCVC++, gcc и Builder используют различные реа-

лизации STL – и результаты тестирования будут сильно зависеть от эффективности реализации тех или иных алгоритмов, а не от качества самого компилятора. Но, так как STL входит в ISO-стандарт, тестирование этой библиотеки просто неотделимо от тестирования самого компилятора.

Проводилось тестирование только наиболее часто используемых классов STL: string, vector, map, sort. При тестировании string'а измерялась скорость конкатенации, для vector'а – время добавления элемента (удаление не тестировалось, так как это просто тестирование realloc'а, которое будет проведено ниже), для map'а измерялось время добавления элемента и скорость поиска необходимого ключа, для sort'а – время сортировки. Так как Microsoft не рекомендует использовать STL в VC++, для сравнения было добавлено тестирование конкатенации строк на основе родного класса VC++ для работы со строками CString и родного класса Builder'а - AnsiString. Результаты оказались очень интересными (табл. 2.9).

Таблица 2.9

Результаты тестирования STL

	VC++	Intel Compiler	Bulder C++	MinGW (gcc)
string add	8 (572)	11 (837)	3 (244)	2 (199)
AnsiString	-	-	11 (832)	-
C string	106 (7476)	104 (7331)	-	-
sort	157 (10994)	156 (10943)	387 (27132)	226 (15848)
vector insert	110 (77)	96 (67)	63 (44)	56 (39)
map insert	1311 (1836)	1455 (2037)	848 (1148)	448 (627)
map find	181 (127)	4 (3)	418 (293)	199 (139)

Согласно результатам, не рекомендованный STL string работает в 12 раз быстрее, чем родной CString Microsoft. Не менее интересен результат gcc – во всех тестах, связанных с выделением памяти, gcc оказался на первом месте.

Тестирование менеджера памяти. При выделении памяти malloc редко обращается напрямую к системе, а использует вместо этого свою внутреннюю структуру для динамического выделения памяти и изменения размера уже выделенного блока. Скорость работы этого внутреннего менеджера может весьма существенно влиять на ско-

рость работы всего приложения. Тестирование менеджера памяти было разбито на две части: в первой измерялась скорость работы пары malloc/free, а во второй – malloc/realloc, причем realloc должен был выделить вдвое больший объем памяти, чем malloc. Результаты тестирования в табл. 2.10.

Таблица 2.10

Результаты тестирования менеджера памяти

	VC++	Intel Compiler	Bulder C++	MinGW (gcc)
malloc	905 (6336)	902 (6317)	24 (174)	882 (6178)
realloc	30 (718)	30 (716)	12 (295)	30 (719)

И снова быстрее всех был Borland Builder C++. Благодаря такой быстрой реализации malloc'a он находится на первом месте по скорости создания/удаления объектов, а также на тестах STL, связанных с изменением размера блока памяти.

Разбор ассемблерного кода неких базовых операций. Для анализа использовался достаточно простой код на C++:

```
void dummyFn1(unsigned);
void dummyFn2(unsigned aa) {
for (unsigned i=0; i<16; i++) dummyFn1(aa);
}
```

Теперь посмотрим, во что этот кусок кода компилирует MSVC++ (приводится только текст необходимой функции):

```
?dummyFn2@@YAXI@Z  PROC  NEAR
push esi
push edi
mov edi, DWORD PTR _aa$[esp+4]
mov esi, 16
$L271:
push edi
call?dummyFn1@@YAXI@Z
add esp, 4
dec esi
jne SHORT $L271
```

```
pop edi
pop esi
ret 0
?dummyFn@@YAXI@Z    ENDP
```

Как видно, MSVC++ инвертировал цикл и for (unsigned i=0; i<16; i++) у него превратился в unsigned i=16; while (i--); , что очень правильно с точки зрения оптимизации – экономим на одной операции сравнения (см. следующий листинг), которая занимает, как минимум, 5 байт, и нарушает выравнивание. Конечно, компилятор по своему усмотрению поменял порядок изменения переменной i, но в данном примере ее используют просто как счетчик цикла, поэтому такая замена вполне допустима.

А вот что выдал Intel Compiler (сначала он полностью развернул цикл, но после увеличения количества итераций на порядок прекратил заниматься такой самодеятельностью):

```
?dummyFn2@@YAXI@Z   PROC   NEAR
$B1$1:
push ebp
push ebx
mov ebp, DWORD PTR [esp+12]
sub esp, 20
xor ebx, ebx
$B1$2:
mov DWORD PTR [esp], ebp
call?dummyFn1@@YAXI@Z
$B1$3:
inc ebx
cmp ebx, 16
jb $B1$2
$B1$4:
add esp, 20
pop ebx
pop ebp
ret
?dummyFn2@@YAXI@Z    ENDP
```

Во-первых, используется прямой порядок цикла for, поэтому появилась дополнительная команда сравнения «cmp ebx, 16». А вот

и очень интересный момент: перед началом цикла, выделив на стеке необходимое количество памяти плюс некий запас («sub esp, 20»), а потом вместо пары push reg; ..; add esp, 4; , как это делает MSVC++, использовали одну команду копирования. Кроме того, использование регистра общего назначения ebx для счетчика цикла вместо индексного esi, как в MSVC++, дополнительно уменьшает время выполнения и размер кода.

Borland Builder сгенерировал следующую конструкцию:

```
@@dummyFn2$qui proc near
?live16385@0:
@1:
push ebp
mov ebp,esp
push ebx
push esi
mov esi,dword ptr [ebp+8]
?live16385@16:
@2:
xor ebx,ebx
@3:
push esi
call @@dummyFn1$qui
pop ecx
@5:
inc ebx
cmp ebx,16
jb short @3
?live16385@32:
@7:
pop esi
pop ebx
pop ebp
ret
@@dummyFn2$qui endp
```

Если не считать большего количества подготовительных операций, то блок вызова собственно функции является чем-то средним между MSVC++ и Intel Compiler: цикл используется прямой и пере-

дача параметров осуществляется с помощью push reg; . Правда, есть интересный момент: вместо add esp, 4 используется pop ecx; что экономит, как минимум, 4 байта, – правда, из-за дополнительного обращения к памяти команда «pop» может работать медленнее, чем сложение.

Ну и, наконец, gcc (стоит обратить внимание на то, что gcc для ассемблера использует синтаксис AT&T):

```
__Z7dummy2Fnj:
LFB1:
pushl %ebp
LCFI0:
movl %esp, %ebp
LCFI1:
pushl %esi
LCFI2:
pushl %ebx
LCFI3:
xorl %ebx, %ebx
movl 8(%ebp), %esi
.p2align 4,,7
L6:
subl $12, %esp
incl %ebx
pushl %esi
LCFI4:
call __Z2dummyFn1j
addl $16, %esp
cmpl $15, %ebx
jbe L6
leal -8(%ebp), %esp
popl %ebx
popl %esi
popl %ebp
ret
```

Данный код является самым плохим из всех приведенных выше – gcc использует прямой цикл плюс пару push esi; ..; add esp, 4 (это происходит неявно в команде «addl $16, %esp») для передачи параметров; а также резервирует место на стеке прямо в цикле, а не вне его, как это делает Intel Compiler. Кроме того, совершенно непонятно, зачем резервировать место на стеке, а потом использовать команду push reg; . Единственный приятный момент – это явное выравнивание начала цикла по границе, чего не делают остальные компиляторы. Поскольку линейка кэша сегмента кода достигает 32-х байт, то метки начала циклов должны быть выровнены по границе 16 байт. На каждый байт, выходящий за пределы кэша, процессор семейства P2 тратит 9 – 12 тактов.

Сравнение времени компиляции и размера выполняемого файла. Для выполнения этого теста используется все тот же исходный код, из которого были удалены все compiler-specific тесты. Тестирование выполнялось отдельно для компиляции релиза и для отладочной версии, размер бинарного файла указан только для релиза (табл. 2.11). Чтобы исключить влияние файлового кэша, проводятся две одинаковые компиляции подряд – время измеряется по второй с помощью команды «date» (исключение составил только Builder – он сам измеряет время компиляции).

Таблица 2.11

Результаты сравнения времени компиляции и размера выполняемого файла

	VC++	Intel Compiler	Bulder C++	MinGW (gcc)
release build time, sec	3	5	2.35	6
release size, Kb	56	72	77	214
debug build time, sec	3	5	3	7

Первое место поделили Borland Builder и MSVC++, а вот gcc – опять на последнем месте, как по скорости компиляции, так и по размеру бинарного файла. Интересным моментом является тот факт, что время компиляции отладочной версии у gcc и Builder'a выше времени компиляции релиза. Объясняется это тем, что при компиляции отла-

дочной версии компилятору необходимо добавить отладочную информацию, что существенно увеличивает размер объектного файла – и, как следствие, время работы линковщика.

Результаты. Казалось бы, вывод о самом эффективном компиляторе напрашивается сам собой – это Borland Builder C++. Но не стоит спешить. Многие разработчики указывают на ошибки при формировании кода у Borland Builder (в частности, при использовании ссылок его поведение становится непредсказуемым). Кроме того, Borland Builder C++ явно наследует многое от Delphi (например, модификатор вызова метода DYNAMIC), в результате чего при компилировании абсолютно правильного C++ кода могут возникать ошибки (например, отсутствие множественного наследования для VCL-классов, а все потомки от TObject являются VCL-классами).

С другой стороны, самым стабильным и «вылизанным» компилятором можно назвать gcc. Но скорость выполнения откомпилированного кода на нем будет не слишком высокой. Причиной тому, вероятно, является существование gcc на многих платформах и, как следствие, необходимость компилирования под эти платформы.

MSVC++ или Intel Compiler не имеют явно выраженных недостатков, так что их позиции примерно равны.

В общем, однозначно ответить, «какой компилятор наилучший ?», невозможно. Результаты вышеприведенных тестов помогут пользователям сделать правильный выбор.

2.4. Алгоритмы и структуры данных в C++

Этот подпункт посвящен языку программирования C, на русском произносится как «Си». Данный язык является прародителем языка C++. Если хорошо знать язык Си, то без труда можно изучить C++, обратное тоже верно. Подробнее про язык Си можно почитать в википедии, основные понятия языка C в табл. 2.12., C++ – в табл. 2.13

Таблица 2.12

Основы языка программирования Си

Изучаемый вопрос	Описание
Основы программирования на Си	Первая программа на языке программирования С. http://cppstudio.com/post/241/
Оператор выбора if в языке С	Управление программным потоком, конструкция if else, условные выражения. http://cppstudio.com/post/6449/
Циклы for, while и do while в языке С	Зачем нужны циклы, примеры использования циклов в программах. Введение в операторы break, continue. http://cppstudio.com/post/6458/
Оператор выбора switch в С(Си)	Что такое оператор множественного выбора switch и в каких случаях его следует использовать. http://cppstudio.com/post/6691/
Функции в языке программирования С	Введение в функции в языке С, для чего они нужны и как их запрограммировать. http://cppstudio.com/post/6471/
Указатели в С	Пример использования указателей, как их объявлять и для чего они нужны. http://cppstudio.com/post/5828/
Динамическое выделение памяти в С	Выделение и высвобождение памяти с помощью функций malloc и free. http://cppstudio.com/post/9088/
Структуры в языке С	Структуры и объединения в языке программирования Си. http://cppstudio.com/post/9172/
Массивы в С (часть 1)	Введение в массивы в языке Си. http://cppstudio.com/post/9244/
Массивы в С (часть 2): многомерные массивы	Учимся работать с двумерными массивами в языке С. http://cppstudio.com/post/9407/
Си-строки	Введение в строки в языке программирования Си. http://cppstudio.com/post/9567/

Таблица 2.13

Основы языка программирования С++

Изучаемый вопрос	Описание
Введение в С++	Установка IDE, введение в язык С++, объявление и использование переменных и многое другое. http://cppstudio.com/post/213/
Структура программы в С++	Структура программ – это разметка рабочей области (области кода) с целью чёткого определения основных блоков программ и синтаксиса. http://cppstudio.com/post/248/
Первая программа на С++	Самая простая программа на С++ – это программа, выводящая на экран в консоли текстовое сообщение. http://cppstudio.com/post/250/
Управляющие символы С++	Символы, которые выталкиваются в поток вывода с целью форматирования вывода или печати некоторых управляющих знаков С++. http://cppstudio.com/post/256/
Арифметические операции С++	Операции, которые присутствуют во всех программах, сложнее «Hello world». Любые манипуляции с переменными выполняются именно благодаря арифметическим операциям. http://cppstudio.com/post/259/
Типы данных С++	Огромную роль в программировании играет процесс отладки программ. Если при отладке программы возникла ошибка, нужно знать, как её исправить. http://cppstudio.com/post/271/
Таблица ASCII	Американский стандартный код для обмена информацией в ОС Windows. http://cppstudio.com/post/276/
Операции присваивания в С++	Для сокращённой записи выражений в языке программирования С++ есть специальные операции, которые называются операциями присваивания. http://cppstudio.com/post/279/
Операции инкремента и декремента в С++	Инкремент ++ – это увеличение на единицу. Декремент -- – это уменьшение на единицу. http://cppstudio.com/post/282/

Продолжение табл. 2.13

Изучаемый вопрос	Описание
Оператор выбора if	Операторы выбора позволяют принять программе решение, основываясь на истинности или ложности условия. http://cppstudio.com/post/286/
Оператор выбора if else	Оператор if else позволяет определить программисту действие, когда условие истинно, и альтернативное действие, когда условие ложно. http://cppstudio.com/post/291/
Логические операции в C++	Логические операции образуют сложное (составное) условие из нескольких простых (два или более) условий. Эти операции упрощают структуру программного кода в несколько раз. http://cppstudio.com/post/297/
Поразрядные логические операции C++	Данные операции работают с битами ячеек памяти и применяются в бинарной арифметике. http://cppstudio.com/post/300/
Приоритет операций в C++	Очерёдность выполнения операций в выражении. http://cppstudio.com/post/302/
Условная операция (операция выбора) в C++	Единственная в C++ трехместная (тернарная) операция используется вместо оператора выбора if else. http://cppstudio.com/post/304/
Оператор множественного выбора switch	Если в программе требуется рассмотреть более чем два варианта ветвления, используется оператор switch. http://cppstudio.com/post/306/
Явное и неявное преобразование типов данных C++	Неявное преобразование типов данных выполняет компилятор C++, а явное преобразование данных выполняет сам программист. http://cppstudio.com/post/310/
Форматированный ввод/вывод в C++	Возможность управлять вводом-выводом в C++ обеспечивают форматирующие функции-члены, флаги и манипуляторы. http://cppstudio.com/post/319/
Генератор случайных чисел rand () в C++	Программа для генерации псевдослучайных чисел. http://cppstudio.com/post/339/

Продолжение табл. 2.13

Изучаемый вопрос	Описание
Цикл for в C++	Многократное прохождение по одному и тому же коду программы. http://cppstudio.com/post/348/
Цикл while в C++	Цикл, повторяющий одно и то же действие, пока условие продолжения цикла while остаётся истинным. http://cppstudio.com/post/352/
Цикл do while в C++	В do while сначала выполняется тело цикла, а затем проверяется условие продолжения цикла. http://cppstudio.com/post/361/
Оператор break	Операторы break применяются для изменения управления в программе. http://cppstudio.com/post/364/
Оператор continue	Оператор continue выполняет пропуск оставшейся части кода тела цикла и переходит к следующей итерации цикла. http://cppstudio.com/post/4271/
Исключения в C++ (exception)	Исключения в языке C++ – это хороший инструмент для обработки нестандартных ситуаций, возникающих в результате работы программы. http://cppstudio.com/post/9773/
Указатели, массивы и строки	
Массивы в C++	Массивы используются для обработки большого количества однотипных данных. http://cppstudio.com/post/389/
Как найти время работы программы на C++	Показаны примеры программ, в которых определено время работы. http://cppstudio.com/post/468/
Квалификатор const в C++	Константы или константные переменные, то есть переменные, значения которых после объявления модифицировать нельзя. http://cppstudio.com/post/394/
Указатели в C++	Специальные переменные, которые ссылаются на блок данных из области памяти, причём на самое его начало. http://cppstudio.com/post/423/
Указатель на указатель + динамическое выделение памяти	Указатели на указатели – эффективный способ организации хранения данных в памяти. http://cppstudio.com/post/9555/

Продолжение табл. 2.13

Изучаемый вопрос	Описание
(часть 1)	
Указатель на указатель + динамическое выделение памяти (часть 2)	Вставка и удаление элементов динамического массива в C++ с помощью указателей. http://cppstudio.com/post/9605/
Ссылки в C++	Особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разименовывается. http://cppstudio.com/post/429/
Динамический массив в C++	Одномерные и двумерные динамические массивы, выделение и высвобождение памяти. http://cppstudio.com/post/432/
Литералы в программировании	Литералы используются в тексте программы для обозначения числовых значений, строк, символов или логических констант. http://cppstudio.com/post/235/
Символы и строки в C++	Введение в работу со строками в C++, основные функции для работы со строками и символами. http://cppstudio.com/post/437/
Кириллица в консоли	Корректное отображение русских букв (кириллицы) в консольном приложении. http://cppstudio.com/post/435/
Структуры и файлы	
Структуры в C++	Использование структуры в программировании, способы ее объявления и инициализации, примеры программ с использованием структур. http://cppstudio.com/post/7008/
Перечисления в C++ (enum)	Назначение перечислений, их использование в своих программах, использование enum. http://cppstudio.com/post/8106/
Работа с файлами в C++	Большинство компьютерных программ работает с файлами, и поэтому возникает необходимость их создавать, удалять, записывать читать, открывать. http://cppstudio.com/post/446/

Продолжение табл. 2.13

Изучаемый вопрос	Описание
Функции, локальные и глобальные переменные, рекурсия	
Функции в C++	Опыт показывает, что для написания больших программ, лучше пользоваться функциями. http://cppstudio.com/post/396/
Прототипы функций в C++	Прототип и описание функции используются компилятором для того, чтобы вызов функции происходил правильным образом. http://cppstudio.com/post/404/
Перегрузка функций в C++	Перегрузка функции – это определение нескольких функций (две или больше) с одинаковым именем, но различными параметрами. http://cppstudio.com/post/406/
Аргументы функций по умолчанию	Значения параметров в функции по умолчанию называют аргументами функций по умолчанию. http://cppstudio.com/post/409/
Встроенные функции в C++	Основная идея заключается в том, чтобы ускорить программу ценой занимаемого места. http://cppstudio.com/post/2676/
Математические функции в C++	В заголовочном файле <cmath> определены функции, выполняющие некоторые часто используемые математические задачи. http://cppstudio.com/post/413/
Локальные и глобальные переменные в C++	Каждая переменная имеет свою область видимости, то есть такую область, в которой можно работать с переменной. http://cppstudio.com/post/415/
Рекурсия в C++	Функция, которая вызывает саму себя, непосредственно (в своём теле) или косвенно (через другую функцию). http://cppstudio.com/post/418/
Параметры функции main (argc, argv)	Если программу запускать через командную строку, то ей можно передать информацию, для этого и существуют параметры argc и argv[]. http://cppstudio.com/post/421/
Передача строки в функцию	Способы передачи строк как параметров функции. Примеры программ, исходный код. http://cppstudio.com/post/7216/

Продолжение табл. 2.13

Изучаемый вопрос	Описание
Введение в объектно-ориентированное программирование	
Классы в C++	Классы и объекты в C++ являются основными концепциями объектно-ориентированного программирования (ООП). http://cppstudio.com/post/439/
Конструктор и деструктор классов в C++	Специальные методы класса – конструктор и деструктор. http://cppstudio.com/post/6964/
Директивы #ifndef и #endif	Данная препроцессорная обёртка предотвращает попытку многократного включения заголовочных файлов. http://cppstudio.com/post/443/
Static: Многоцелевое ключевое слово	Ключевое слово static, хотя и означает «неизменный», имеет несколько способов использования. http://cppstudio.com/post/3298/
Перегрузка операторов в C++	Перегрузив оператор +, можно складывать массивы строки и даже целые объекты простой записью вида a+b. http://cppstudio.com/post/7958/
Перегрузка операторов в C++ (часть 2)	Перегрузка операций = (присваивание), == (равенство) и [ ] (индексация). http://cppstudio.com/post/10058/
Разработка интерфейсов классов в C++	Что такое интерфейсы классов и для чего они нужны? http://cppstudio.com/post/2992/
Дружественные функции C++	Хотя дружественные функции и нарушают целостность классов, бывают такие случаи, когда необходимо дать доступ к закрытым свойствам классов, и без дружественных функций никак не обойтись. http://cppstudio.com/post/8423/
Указатель this C++	Что такое указатель this, как его использовать и зачем он нужен. http://cppstudio.com/post/8712/
Дружественные классы C++	Пример использования дружественных классов. http://cppstudio.com/post/9091/
Конструктор копирования в C++	Конструктор копирования нужен для того, чтобы создавать «реальные» копии объектов класса, а не побитовую копию объекта. http://cppstudio.com/post/9903/

Окончание табл. 2.13

Изучаемый вопрос	Описание
Наследование классов	Введение в одну из главных концепций ООП – наследование. http://cppstudio.com/post/10103/
Шаблоны в C++ (template)	
Шаблоны функций в C++	Объявление и использование шаблонов для создания шаблонов функций. http://cppstudio.com/post/5165/
Шаблоны классов в C++	Объявление и использование шаблонов классов, пример создания шаблона класса Стек. http://cppstudio.com/post/5188/
Стандартная библиотека шаблонов (STL)	
string: шаблонный строковый класс STL	Введение в основы шаблона класса string стандартной библиотеки шаблонов C++. http://cppstudio.com/post/6110/

С момента возникновения программирования стало актуальным разрабатывать как можно более эффективные алгоритмы (табл. 2.14), которые решают общие проблемы в программировании: сортировка, поиск и т.д. Благодаря таким алгоритмам тратится меньше ресурсов компьютера и, соответственно, программа работает быстрее. Не стоит изобретать велосипед, тратить время на разработку своих собственных алгоритмов, все уже написано ранее. Остается лишь только уметь воспользоваться этими алгоритмами в своих целях.

Таблица 2.14

Алгоритмы и структуры данных в C++

Изучаемый вопрос	Описание
Поиск в массивах	
Линейный поиск (поиск в лоб) в массивах в C++	Алгоритм и программа на C++, реализующая линейный поиск. http://cppstudio.com/post/452/
Двоичный (бинарный) поиск в массивах в C++	Алгоритм бинарного поиска и программа, показывающая результат работы этого алгоритма. http://cppstudio.com/post/2996/

Окончание табл. 2.14

Сортировки массивов	
Сортировка пузырьком	Алгоритм сортировки пузырьком и пример программы, которая сортирует элементы массива. http://cppstudio.com/post/457/
Сортировка выбором	Программа, выполняющая сортировку выбором. http://cppstudio.com/post/459/
Сортировка вставками	Наглядный пример работы сортировки вставками. http://cppstudio.com/post/462/
Структуры данных	
Структура данных: очередь	Всем хорошо известно, что такое очередь. В программировании тоже есть такое понятие, оно реализовано как структура данных. http://cppstudio.com/post/5159/
Структура данных: стеки	Очень интересная структура данных. Как известно, она чаще других используется ОС для реализации своих внутренних процессов. http://cppstudio.com/post/5155/
Другие алгоритмы	
Длиная арифметика в C++	Иногда приходится оперировать громадными числами, делать разные вычисления, в таких случаях простых типов данных языка C++ не хватает. Но все же из такой ситуации есть выход. http://cppstudio.com/post/5036/

Линейный поиск в массивах, или как его ещё называют, поиск в ЛОБ, эффективен в массивах с небольшим количеством элементов, причём элементы в таких массивах никак не отсортированы и не упорядочены. Алгоритм линейного поиска в массивах последовательно проверяет все элементы массива и сравнивает их с ключевым значением. Таким образом, в среднем необходимо проверить половину значений в массиве, чтобы найти искомое значение. Чтобы убедиться в отсутствии искомого значения, необходимо проверить все элементы массива. Разработаем программу, которая ищет минимальное значение в массиве. Поиск в программе реализован согласно алгоритму линейного поиска в массиве.

```
    // min_max.cpp: определение точки входа для консольного
    //приложения.
1   // поиск минимального значения в одномерном масиве
2   #include  "stdafx.h"
3   #include <iostream>
4   #include <ctime>
5   using namespace std;
6   int main(int argc, char* argv[])
7   {
8      srand(time(0));
9      const int array_size = 25;  // размер одномерного массива
10     int array1[array_size];  // объявление одномерного массива
11     for (int counter = 0;  counter < array_size;  counter++)
12     {
13     array1[counter] = rand() % 50 - rand() % 50;  // заполнение
14  //массива случайными значениями в диапазоне
15  //от -49 до 49 включительно
16       cout << array1[counter] << " ";  // печать элементов
17  // одномерного  массива array1
18     }
19     int min = array1[0];  // переменная для хранения
20  //минимального значения
21     for (int counter = 1;  counter < array_size;  counter++)
22     {
23     if ( min > array1[counter] ) // поиск минимального значения
24   в одномерном массиве
25         min = array1[counter];
26     }
27     cout << "nmin = " << min << endl;
28     system("pause");
       return 0;
    }
```

В строке 12 объявлена переменная с квалификатором const. В цикле for (строки 14 – 18) заполняется массив array1 и сразу же печатаются значения элементов массива. Два действия объединены в один цикл, таким образом не надо объявлять отдельный цикл для вывода значений массива. В строке 19 объявлена переменная min для хранения минимального значения, которое будет найдено в ходе ли-

нейного поиска в массиве. Причём переменная min инициализирована первым значением массива, так как перед сравнением необходимо сначала инициализировать переменную. В строках 20 – 24 объявлен цикл for, в котором будет выполнятся алгоритм линейного поиска в массивах. Переменная-счётчик в цикле for инициализирована единицей, то есть обработка массива начнётся со второго элемента, ведь первый элемент массива уже содержится в переменной min В теле цикла объявлен оператор условного выбора if, который будет последовательно сравнивать значение в переменной min со значениями элементов массива array1. В каждой итерации цикла будет выполнятся проверка условия в строке 22, и если условие истинно, то в переменную min будет записываться всё меньшее и меньшее значение, если таковое окажется в массиве. В противном случае значение в переменной min меняться не будет. Результат работы алгоритма линейного поиска в массиве отражен на рис. 2.1.

Рис. 2.1. Поиск в массивах C++

Как показала программа, значение -26 – это минимальное значение массива array1.

Известно, как искать минимальное значение в массиве, используя алгоритм линейного поиска в массиве. Нужно переделаеть эту программу так, чтобы она искала максимальное значение в массиве. Всё, что нужно изменить, так это знак строгого отношения в операторе условного выбора if – на противоположный, строка 22.

```
    // фрагмент кода (линейный поиск в массиве)
1    if ( max < array1[counter] ) // поиск максимального значения в од-
2   номерном массиве
```

Теперь известно, как быстро и легко переделать алгоритм линейного поиска в массивах минимального значения на алгоритм линейного поиска в массивах максимального значения. Организуется линейный поиск в одномерном массиве, в двумерных массивах алго-

ритм линейного поиска не изменится. Рассмотрим фрагмент кода – алгоритм линейного поиска в двумерном массиве.

```
    int min = array1[counter_string][counter_column];  // переменная для
    // хранения минимального значения
1   for (int counter_string = 0;  counter_string < counter_string;
2   counter_string++)
3       {
4           for (int counter_column = 0;  counter_column < counter_column;
5  counter_column++)
6                   {
7                 if ( min > array1[counter_string][counter_column] ) // поиск
8   минимального значения в двумерном массиве
9             min = array1[counter_string][counter_column];
                    }
       }
```

В строке 1 инициализируется переменная min первым значением двумерного массива array1, для того чтобы избежать логических ошибок. Фактически, переменную min можно не инициализировать, и программа будет работать правильно. Но существует вероятность возникновения логической ошибки, то есть программа сработает, но сработает неправильно. Суть в том, что при объявлении переменной в ней изначально будет содержаться какое-то значение (это значение называют мусор), и в зависимости от того, какое значение будет содержаться в переменной изначально, без принудительной инициализации этой переменной программа будет работать неправильно, так как будет сравнивать значения массива с мусором. Обычно мусор всегда положительный, а значит, минимальное значение в массиве будет искаться безошибочно. В случае с алгоритмом линейного поиска в массиве максимального значения программа всегда будет срабатывать неправильно. Именно поэтому нужно принудительно инициализировать переменную min или max начальным значением. Тогда возникает вопрос: «Каким значением инициализировать эту переменную?». Инициализируем переменную min любым значением из массива. Не обязательно выполнять инициализацию первым значением из массива, можно взять любое. Но для простоты следует брать первое значение массива. Алгоритм линейного поиска в двумерном мас-

сиве не сильно изменился, а точнее, вообще не изменился. Добавился ещё один цикл for, ведь поиск выполняется в двумерном массиве.

2.5. Алгоритмы сортировки в массивах в C++

Двоичный (бинарный) поиск – это алгоритм поиска элемента в отсортированном массиве. Бинарный поиск нашел себе применение в математике и информатике. Можно не пользоваться алгоритмом двоичного поиска, но знать принцип его работы – обязательно. Двоичный поиск можно использовать только в том случае, если есть массив, все элементы которого упорядочены (отсортированы). Бинарный поиск не используется для поиска максимального или минимального элементов, так как в отсортированном массиве эти элементы содержатся в начале и в конце массива соответственно, в зависимости от того, как отсортирован массив, по возрастанию или по убыванию. Поэтому алгоритм бинарного поиска применим, если необходимо найти некоторый ключевой элемент в массиве. То есть организовать поиск по ключу, где ключ – это определённое значение в массиве. Разработаем программу, в которой объявим одномерный массив, и организуем двоичный поиск. Объявленный массив нужно инициализировать некоторыми значениями, причём так, чтобы эти значения были упорядочены.

```
1   // binary_search.cpp: определение точки входа для консольного
2   // приложения
3   #include  "stdafx.h"
4   #include <iostream>
5   using namespace std;
6   int main(int argc, char* argv[])
7   {
8      const int size_array = 10;
9      int array_[size_array] = {-8, -7, -6, -6, -4, 2, 6, 7, 8, 15 };
10  // объявление одномерного массива
11     cout << "array[" << size_array << "] = { ";
12     for (int counter = 0;  counter < size_array;  counter++)
13     {
14      cout << array_[counter] << " ";
15  // печать элементов одномерного массива array1
```

```
16      }
17      cout << " }";
18      int average_index = 0, // переменная для хранения
19    //индекса среднего элемента массива
20          first_index   = 0, // индекс первого элемента в массиве
21          last_index    = size_array - 1 , // индекс последнего элемента
22    // в массиве
23    // ---------------------------------------------
24          search_value  = 15;  // искомое (ключевое) значение
25    // ---------------------------------------------
26      if (last_index == -1) cout << "\narray is empty" << endl;
27    // массив пуст
28
29      while  (first_index  <  last_index)
30       {
31          average_index = first_index + (last_index - first_index) / 2;
32    // меняем индекс среднего значения
33          search_value <= array_[average_index] ? last_index =
34     average_index : first_index = average_index  +  1;
35      // найден ключевой элемент или нет
36       }
37      if ( array_[last_index] == search_value)
38          cout << "\nvalue is found" << "\nindex = " << last_index << endl;
39      else
40          cout << "\nvalue is not found" << endl;
41       system("pause");
42      return 0;
43    }
```

Итак, в строке 8 объявлена целочисленная переменная константа, которой присвоено значение 10 – размер одномерного массива. Согласно тону хорошего программирования, размер статического массива должен задаваться в отдельной переменной, с квалификатором const. В строке 9 объявлен одномерный массив соответствующего размера. Строки 11 – 16 выводят на экран элементы массива с некоторым оформлением. В строках 18 – 21 объявляются переменные, которые будут использоваться в алгоритме двоичного поиска. В строке 24 объявлена переменная, значение в которой будет искомым. В строках 26 – 36 находится алгоритм двоичного поиска в мас-

сивах. Сначала нужно проверить размер массива, в котором будет искаться ключевое значение, строка 26. Массив может быть нулевого размера, если размер массива больше или равен 1, тогда нужно начать искать ключевое значение. Объявление цикла while, строки 29 – 36, в цикле организован поиск значения таким образом, что после выхода из цикла индекс найденного значения сохранится в переменной last_index. В теле цикла, строка 28, объявлена условная операция ? : , хотя можно было воспользоваться оператором выбора if else. И наконец, в строках 37 – 42 объявлен оператор условного выбора if else, который определяет, говорить ли о том, что было найдено искомое значение, или нет.

Рассмотрим пошагово, как именно работает алгоритм двоичного поиска в массивах на псевдокоде:

Шаг первый:
проверка условия цикла: 0 < 9 – true
average_index = 0 + (9 – 0) / 2 = 4, значит, средний элемент -4
проверка в условной операции 15 <= (-4) – false
значит, first_index = 4 + 1 = 5

Шаг второй:
проверка условия цикла: 5 < 9 – true
average_index = 5 + (9 – 5) / 2 = 7, значит, средний элемент 7
проверка в условной операции 15 <= 7 – false
значит, first_index = 7 + 1 = 8

Шаг третий:
проверка условия цикла: 8 < 9 – true
average_index = 8 + (9 – 8) / 2 = 8 // значит, средний элемент 8
проверка в условной операции 15 <= 8 – false
значит, first_index = 8 + 1 = 9

Шаг четвёртый:
проверка условия цикла: 9 < 9 – false // выходим из цикла при этом значение в переменной last_index не менялось, так как искомое значение в последнем элементе массива

Так как алгоритмы сортировки пока не известны, то массив инициализируем вручную, причём обязательно упорядоченно. В строке 24 нужно указать искомый элемент массива и запустить программу (рис. 2.2).

Сортировка пузырьком – это простейший алгоритм сортировки, применяемый чисто для учебных целей. Практического применения этому алгоритму нет, так как он не эффективен, особенно если необходимо отсортировать массив большого размера. К плюсам сортировки пузырьком относится простота реализации алгоритма.

Рис. 2.2. Поиск в массивах C++

Алгоритм сортировки пузырьком сводится к повторению проходов по элементам сортируемого массива. Проход по элементам массива выполняет внутренний цикл. За каждый проход сравниваются два соседних элемента, и если порядок неверный, элементы меняются местами. Внешний цикл будет работать до тех пор, пока массив не будет отсортирован. Таким образом внешний цикл контролирует количество срабатываний внутреннего цикла. Когда при очередном проходе по элементам массива не будет совершено ни одной перестановки, то массив будет считаться отсортированным. Чтобы хорошо понять алгоритм, отсортируем методом пузырька массив, состоящий, к примеру, из семи чисел (табл. 2.15).

Исходный массив: 3 3 7 1 2 5 0.

Таблица 2.15

Сортировка пузырьком

№ операции	Элементы массива							Перестановки
Исходный массив	3	3	7	1	2	5	0	
0	3	3						false
1		3	7					false
2			1	7				7 > 1, true
3				2	7			7 > 2, true

Продолжение табл. 2.15

№ операции	Элементы массива							Перестановки
4					5	7		7 > 5, true
5						0	7	7 > 0, true
Текущий массив	3	3	1	2	5	0	7	
0	3	3						false
1		1	3					3 > 1, true
2			2	3				3 > 2, true
3				0	3			3 > 0, true
4					3	5		false
5						5	7	false
Текущий массив	3	1	2	0	3	5	7	
0	1	3						3 > 1, true
1		2	3					3 > 2, true
2			0	3				3 > 0, true
3				3	3			false
4					3	5		false
5						5	7	false
Текущий массив	1	2	0	3	3	5	7	
	1	2						false
		0	2					2 > 0, true
			2	3				false
				3	3			false
					3	5		false
						5	7	false

Окончание табл. 2.15

№ операции	Элементы массива							Перестановки
Текущий массив	1	0	2	3	3	5	7	
	0	1						1 > 0, true
		1	2					false
			2	3				false
				3	3			false
					3	5		false
						5	7	false
Конечный массив	0	1	2	3	3	5	7	
Конец сортировки								

Для того чтобы отсортировать массив, хватило пяти запусков внутреннего цикла, for. Запустившись, цикл for срабатывал шесть раз, так как элементов в массиве семь, то итераций (повторений) цикла for должно быть на одно меньше. На каждой итерации сравниваются два соседних элемента массива. Если текущий элемент массива больше следующего, то их меняют местами. Таким образом, пока массив не будет отсортирован, будет запускаться внутренний цикл и выполняться операция сравнения. Необходимо обратить внимание на то, что за каждое полное выполнение цикла for как минимум один элемент массива находит своё место. В худшем случае, понадобится n-2 запуска внутреннего цикла, где n – количество элементов массива. Это говорит о том, что сортировка пузырьком крайне неэффективна для больших массивов.

Разработаем программу, в которой сначала необходимо ввести размер одномерного массива, после чего массив заполняется случайными числами и сортируется методом пузырька.

```
1   // bu_sort.cpp: определяет точку входа для консольного приложения.
2
3   #include  "stdafx.h"
4   #include <iostream>
5   #include <iomanip>
6   #include <ctime>
```

```
7    using namespace std;
8
9    void bubbleSort(int *, int);  // прототип функции сортировки пузырьком
10
11   int main(int argc, char* argv[])
12   {
13      srand(time(NULL));
14      setlocale(LC_ALL,  "rus");
15      cout << "Введите размер массива: ";
16      int size_array;  // длина массива
17      cin >> size_array;
18
19      int *sorted_array = new int [size_array];  // одномерный динамический массив
20      for (int counter = 0;  counter < size_array;  counter++)
21      {
22         sorted_array[counter] = rand() % 100;  // массив заполняется случайными числами
23         cout << setw(2) << sorted_array[counter] << "  ";  // вывод массива на экран
24      }
25      cout << "\n\n";
26
27      bubbleSort(sorted_array, size_array);  // вызов функции сортировки пузырьком
28
29      for (int counter = 0;  counter < size_array;  counter++)
30      {
31         cout << setw(2) << sorted_array[counter] << "  ";  // печать отсортированного массива
32      }
33      cout << "\n";
34
35      system("pause");
36      return 0;
37   }
38
```

```
39  void bubbleSort(int* arrayPtr, int length_array) // сортировка
    пузырьком
40   {
41   int temp = 0;  // временная переменная для хранения элемента
    массива
42   bool exit = false;  // булева переменная для выхода из цикла, если
    массив отсортирован
43
44   while (!exit) // пока массив не отсортирован
45   {
46   exit = true;
47   for (int int_counter = 0;  int_counter < (length_array - 1);
48  int_counter++) // внутренний цикл
49      //сортировка пузырьком по возрастанию – знак >
50      //сортировка пузырьком по убыванию – знак <
51      if (arrayPtr[int_counter] > arrayPtr[int_counter + 1])
52  // сравниваются два соседних элемента
53      {
54      // выполняется перестановка элементов массива
55      temp = arrayPtr[int_counter];
56      arrayPtr[int_counter] = arrayPtr[int_counter + 1];
57      arrayPtr[int_counter + 1] = temp;
58      exit = false;  // на очередной итерации произведена переста-
    новка элементов
59      }
60   }
61  }
```

Результат работы программы показан на рис. 2.3

Рис. 2.3. Сортировка пузырьком

Сортировка выбором – самый простой алгоритм сортировки. Судя по названию сортировки, необходимо что-то выбирать (макси-

мальный или минимальный элементы массива). Алгоритм сортировки выбором находит в исходном массиве максимальный или минимальный элементы – в зависимости от того, как необходимо сортировать массив: по возрастанию или по убыванию. Если массив должен быть отсортирован по возрастанию, то из исходного массива необходимо выбирать минимальные элементы. Если же массив необходимо отсортировать по убыванию, то выбирать следует максимальные элементы.

Допустим, необходимо отсортировать массив по возрастанию. В исходном массиве находим минимальный элемент, меняем его местами с первым элементом массива. Из всех элементов массива только один элемент стоит на своём месте. Теперь будем рассматривать неотсортированную часть массива, то есть все элементы массива, кроме первого. В неотсортированной части массива опять ищем минимальный элемент. Найденный минимальный элемент меняем местами со вторым элементом массива и т. д. Таким образом, суть алгоритма сортировки выбором сводится к многократному поиску минимального (максимального) элемента в неотсортированной части массива. Отсортируем массив из семи чисел согласно алгоритму «Сортировка выбором».

Исходный массив: 3 3 7 1 2 5 0:

1) находим минимальный элемент в массиве. 0 – минимальный элемент;

2) меняем местами минимальный и первый элементы массива – текущий массив: 0 3 7 1 2 5 3;

3) находим минимальный элемент в неотсортированной части массива. 1 – минимальный элемент;

4) меняем местами минимальный и первый элементы массива – текущий массив: 0 1 7 3 2 5 3;

5) min = 2;

6) текущий массив: 0 1 2 3 7 5 3;

7) min = 3;

8) текущий массив: 0 1 2 3 7 5 3 – в массиве ничего не поменялось, так как 3 стоит на своём месте;

9) min = 3;

10) конечный вид массива: 0 1 2 3 3 5 7 – массив отсортирован.

Запрограммируем алгоритм сортировки выбором в C++:

```
I   // sorting_choices.cpp: определяет точку входа для консольного
    приложения.
2
3   #include  "stdafx.h"
4   #include <iostream>
5   #include <ctime>
6   #include <iomanip>
7   using namespace std;
8
9   void choicesSort(int*, int);  // прототип функции сортировки вы-
    бором
10
II  int main(int argc, char* argv[])
12  {
13     srand(time(NULL));
14     setlocale(LC_ALL,  "rus");
15     cout << "Введите размер массива: ";
16     int size_array;  // длина массива
17     cin >> size_array;
18
19     int *sorted_array = new int [size_array];  // одномерный динами-
    ческий массив
20     for (int counter = 0;  counter < size_array;  counter++)
21     {
22        sorted_array[counter] = rand() % 100;  // заполняем массив
    случайными числами
23        cout << setw(2) << sorted_array[counter] << "  ";  // вывод
    массива на экран
24     }
25     cout << "\n\n";
26
27     choicesSort(sorted_array, size_array);  // вызов функции сорти-
    ровки выбором
28
29     for (int counter = 0;  counter < size_array;  counter++)
30     {
31        cout << setw(2) << sorted_array[counter] << "  ";  // печать от-
    сортированного массива
32     }
```

```
33     cout << "\n";
34     delete [] sorted_array;  // высвобождаем память
35     system("pause");
36     return 0;
37  }
38
39  void choicesSort(int* arrayPtr, int length_array) // сортировка выбором
40  {
41     for (int repeat_counter = 0;  repeat_counter < length_array;  repeat_counter++)
42     {
43        int temp = arrayPtr[0];  // временная переменная для хранения значения перестановки
44        for (int element_counter = repeat_counter + 1;  element_counter < length_array;  element_counter++)
45        {
46           if (arrayPtr[repeat_counter] > arrayPtr[element_counter])
47           {
48              temp = arrayPtr[repeat_counter];
49              arrayPtr[repeat_counter]  =  arrayPtr[element_counter];
50              arrayPtr[element_counter] = temp;
51           }
52        }
53     }
54  }
```

Алгоритм сортировки выбором основан на алгоритме поиска максимального (минимального) элемента. Фактически алгоритм поиска является важнейшей частью сортировки выбором. Так как основная задача сортировки – упорядочивание элементов массива, необходимо выполнять перестановки. Обмен значений элементов сортируемого массива происходит в строках 48 – 50. Если поменять знак > в строке 46 на знак <, то сортироваться массив будет по убыванию. Результат работы программы показан на рис. 2.4.

Рис. 2.4. Сортировка выбором

Сортировка вставками – достаточно простой алгоритм. Как и в любом другом алгоритме сортировки, с увеличением размера сортируемого массива увеличивается и время сортировки. Основным преимуществом алгоритма сортировки вставками является возможность сортировать массив по мере его получения. То есть имея часть массива, можно начинать его сортировать. В параллельном программировании такая особенность играет немаловажную роль.

Сортируемый массив можно разделить на две части – отсортированная часть и неотсортированная. В начале сортировки первый элемент массива считается отсортированным, все остальные – неотсортированными. Начиная со второго элемента массива и заканчивая последним, алгоритм вставляет неотсортированный элемент массива в нужную позицию в отсортированной части массива. Таким образом, за один шаг сортировки отсортированная часть массива увеличивается на один элемент, а неотсортированная – уменьшается на один элемент. Рассмотрим пример сортировки по возрастанию массива из семи чисел (табл. 2.16):

Исходный массив: 3 3 7 1 2 5 0.

Таблица 2.16

Сортировка вставками

Шаг	Отсортированная часть массива							Текущий элемент	Вставка
1	3							3	false
2	3	3						7	false
3	3	3	7					1	true
4	1	3	3	7				2	true
5	1	2	3	3	7			5	true
6	1	2	3	3	5	7		0	true
-	0	1	2	3	3	5	7	-	-

На каждом шаге сортировки текущий элемент сравнивается со всеми элементами в отсортированной части. На первом шаге сравнивается тройка с тройкой, они равны и поэтому их не меняют местами. На втором шаге сравниваем семерку с двумя тройками, 7 > 3, а так как сортировка по возрастанию, то опять элементы массива остаются на своих местах. На третьем шаге единица сравнивается с тремя элементами, и все они больше единицы, значит, единицу вставляют на первое место, в начало массива. На четвёртом шаге текущий элемент – 2 – сравнивается с элементами 1, 3, 3, 7. Получается, что 1 < 2 < 3 < < 7, поэтому двойку вставляем между единицей и тройкой. Пятый и шестой шаги выполняются точно так же. В итоге, на шестом шаге мы получаем отсортированный по возрастанию массив. Запрограммируем алгоритм сортировки вставками на C++:

```
I  // insertion_sort.cpp: определяет точку входа для консольного
   приложения.
2
3  #include  "stdafx.h"
4  #include <iostream>
5  #include <ctime>
6  #include <iomanip>
7  using namespace std;
8
9  void insertionSort(int *, int);  // прототип функции сортировки
   вставками
10
II int main(int argc, char* argv[])
12 {
13    srand(time(NULL));
14    setlocale(LC_ALL,  "rus");
15    cout << "Введите размер массива: ";
16    int size_array;  // длина массива
17    cin >> size_array;
18
19    int *sorted_array = new int [size_array];  // одномерный динами-
   ческий массив
20    for (int counter = 0;  counter < size_array;  counter++)
21     {
```

```
22        sorted_array[counter] = rand() % 100;  // массив заполняется
  случайными числами
23        cout << setw(2) << sorted_array[counter] << "  ";  // вывод мас-
  сива на экран
24     }
25    cout << "n";
26
27    insertionSort(sorted_array, size_array);  // вызов функции сорти-
  ровки вставками
28
29    for (int counter = 0;  counter < size_array;  counter++)
30     {
31        cout << setw(2) << sorted_array[counter] << "  ";  // печать от-
  сортированного массива
32     }
33    cout << "n";
34    delete [] sorted_array;  // высвобождается память
35     system("pause");
36    return 0;
37 }
38
39 void insertionSort(int *arrayPtr, int length) // сортировка вставками
40 {
41    int temp, // временная переменная для хранения значения эле-
  мента сортируемого массива
42        item;  // индекс предыдущего элемента
43    for (int counter = 1;  counter < length;  counter++)
44     {
45        temp = arrayPtr[counter]; // инициализация временной пере-
  менной текущим значением элемента массива
46        item = counter-1;  // запоминание индекса предыдущего эле-
  мента массива
47        while(item >= 0 && arrayPtr[item] > temp) // пока индекс не
  равен 0 и предыдущий элемент массива больше текущего
48        {
49           arrayPtr[item + 1] = arrayPtr[item];  // перестановка элемен-
  тов массива
50           arrayPtr[item] = temp;
51           item--;
```

```
52        }
53    }
54 }
```

Программа сортирует одномерный массив по возрастанию. Изменив знак > на знак < в строке 47, массив будет сортироваться по убыванию. Результат работы программы показан на рис. 2.5.

Рис. 2.5. Сортировка вставками

2.6. Структура данных: очередь и стек

Очередь – это структура данных, которая, как и стек, имеет ограничения по добавлению и удалению элементов. Чтобы понять смысл очереди в программировании, можно представить очередь в магазин: человек в её начале (тот, кто пришел первый) будет обслуживаться первым, вновь пришедшие люди будут добавляться в конец очереди. Таким образом, первый человек в очереди обслуживается первым, последний в ней – последним. Сокращенно очереди обозначаются как: FIFO - First In, First Out (первым пришел, первым ушел).

Очереди часто используются в программировании сетей, операционных систем и других ситуациях, в которых различные процессы должны разделять такие ресурсы, как процессорное время.

Немного терминологии:

– enqueue – добавление элемента в очередь;

– dequeue – удаления элемента из очереди.

Хотя концепция очереди может быть простой, в программировании очередь не так проста, как стек. Вернемся к примеру с очередью в магазин. Что произойдет, когда один человек покидает очередь? Все в очереди должны пройти вперед, верно? Теперь представьте себе, что одновременно только один человек может двигаться. Таким обра-

зом, второй человек делает шаг вперед, чтобы заполнить место, оставленное от первого лица, а затем третьим лицом делается шаг вперед, чтобы заполнить место, оставленное после второго человека, и так далее. Теперь представьте, что никто не может уйти или быть добавленным в очередь, пока все не шагнули вперед. Понятно, что очередь будет двигаться очень медленно. Конечно, нетрудно запрограммировать очередь, и она будет работать, но очень непросто сделать очередь, которая бы работала очень быстро!

Есть несколько основных способов реализации очереди. Во-первых, просто создать массив и переложить все элементы, чтобы поместить элемент в очередь или извлечь его. Данный способ является очень медленным, так как чем больше элементов в очереди, тем больше времени занимает перемещение.

Есть еще один метод, когда, не смещая элементы в очереди, выполняются функции постановки и удаления элементов из очереди. Можно представить себе, что не очередь в магазин движется к продавцу, а продавец движется к очереди. Таким образом, позиция начала очереди постоянно изменяется, стремится к ее концу. Люди, которые стоят в очереди, не делают шагов вперед или назад, что экономит время.

К сожалению, этот метод является гораздо более сложной задачей, чем очередь из массива. Вместо того чтобы отслеживать только «конец» очереди, нужно следить также за началом очереди. Все это еще более усложняется, когда понимают, что после того, как в очередь добавляется и извлекается элемент, очередь будет нужно обернуть вокруг конца массива. Пока люди входят и выходят, очередь движется все дальше и дальше назад, и в конце концов очередь окружит магазин, а в конечном итоге – вернется на прежнее место.

Лучший способ понять, как организована работа очереди – это разобраться в исходном коде. В приведенной далее программе написан шаблон класса для структуры данных «очередь». Шаблон помещен в отдельный заголовочный файл, что является приемлемым способом утилизации кода. Интерфейс шаблонов классов нельзя помещать в файл без реализации класса, так как это связано с особенностью шаблонов классов в C++.

Код программы приведен ниже:

```
1   #ifndef QUEUE_H
2   #define QUEUE_H
3    #include <cassert>
4    template<typename T>
5   class Queue
6   {
7   private:
8      T *queuePtr;      // указатель на очередь
9      const int size;   // максимальное количество элементов
                            в очереди
10     int begin,        // начало очереди
11        end;           // конец очереди
12     int elemCT;       // счетчик элементов
13  public:
14     Queue(int =10);          // конструктор по умолчанию
15     Queue(const Queue<T> &); // конструктор копирования
16     ~Queue();                // деструктор
17      void enqueue(const T &);     // добавить элемент в очередь
18     T dequeue();     // удалить элемент из очереди
19     void printQueue();
20  };
21
22  // реализация методов шаблона класса Queue
23
24  // конструктор по умолчанию
25  template<typename T>
26  Queue<T>::Queue(int sizeQueue) :
27     size(sizeQueue), // инициализация константы
28     begin(0), end(0), elemCT(0)
29  {
30     // дополнительная позиция помогает различать конец
          и начало очереди
31     queuePtr = new T[size + 1];
32  }
33   // конструктор копии
34  template<typename T>
35  Queue<T>::Queue(const Queue &otherQueue) :
```

```
36      size(otherQueue.size) , begin(otherQueue.begin),
37      end(otherQueue.end), elemCT(otherQueue.elemCT),
38      queuePtr(new T[size + 1])
39   {
40      for (int ix = 0;  ix < size;  ix++)
41         queuePtr[ix] = otherQueue.queuePtr[ix];  // копирует очередь
42   }
43    // деструктор класса Queue
44   template<typename T>
45   Queue<T>::~Queue()
46    {
47      delete [] queuePtr;
48    }
49    // функция добавления элемента в очередь
50   template<typename T>
51   void Queue<T>::enqueue(const T &newElem)
52    {
53      // проверяетм , есть ли свободное место в очереди
54      assert( elemCT < size );
55       // обратить внимание на то , что очередь начинает заполнять-
     ся с 0-го индекса
56      queuePtr[end++] = newElem;
57       elemCT++;
58       // проверка кругового заполнения очереди
59      if (end > size)
60         end -= size + 1; // возвращаем end на начало очереди
61   }
62
63   // функция удаления элемента из очереди
64   template<typename T>
65   T Queue<T>::dequeue()
66    {
67      // проверяет, есть ли в очереди элементы
68      assert( elemCT > 0 );
69       T returnValue = queuePtr[begin++];
70      elemCT--;
71       // проверка кругового заполнения очереди
72      if (begin > size)
73         begin -= size + 1;  // возвращает  behin на начало очереди
```

```
        return returnValue;
74  }
75  template<typename T>
76  void Queue<T>::printQueue()
77  {
78     cout << «Очередь: »;
79
80     if (end == 0 && begin == 0)
81        cout << " пустая\n";
82     else
83     {
84        for (int ix = end;  ix >= begin;  ix--)
85           cout << queuePtr[ix] << " ";
86        cout << endl;
87     }
88  }
89  #endif // QUEUE_H
```

Класс реализован с помощью шаблонов, а размер определяется динамически при инициализации (хотя по умолчанию составляет 10 элементов).

В программе выделяется память под очередь. Памяти выделяется на одно место больше для того, чтобы не получилось так, что очередь будет перемещаться по всей памяти, и так удобнее организовывать круговую работу очереди.

Чтобы понять метод круговой реализации очереди, нужно представить массив в виде круга. Когда элемент удаляется из очереди, оставшиеся элементы не сдвигаются вперед к началу очереди. Вместо этого начало очереди сдвигается к элементам. В итоге начало очереди будет сдвинуто так далеко, что очередь будет выходить за пределы конца массива. Когда очередь доходит до конца массива, то обтекает к началу массива.

Чтобы такого не происходило, используют алгоритм круговой организации очереди. Когда очередь подходит к концу массива, она возвращается на его начало.

Для тестирования класса разработана программа-драйвер, код которой можно увидеть ниже:

```
#include <iostream>

using namespace std;
#include "queue.h" // подключается шаблон класса
int main ()
{
   Queue<char> myQueue(14);  // объект класса очередь
   myQueue.printQueue();  // вывод очереди
   int ct = 1;
   char ch;
   // добавление элементов в очередь
   while (ct++ < 14)
   {
      cin >> ch;
      myQueue.enqueue(ch);
   }

   myQueue.printQueue();  // вывод очереди

   // удаление элемента из очереди
   myQueue.dequeue();
   myQueue.dequeue();
   myQueue.dequeue();

   myQueue.printQueue();  // вывод очереди

   cout << "\n\nСработал конструктор копирования:\n";
   Queue<char> newQueue(myQueue);

   newQueue.printQueue();  // вывод очереди

  return 0;
}
```

В этой программе последовательно запускаются различные методы класса и выполняется вывод содержимого очереди для того, чтобы отследить изменения. Результат работы программы показан ниже:

```
CppStu Очередь:  пустая
STREET WORKOUT
Очередь:  T U O K R O W T E E R T S
Очередь:  T U O K R O W T E E

Сработал конструктор копирования:
Очередь:  T U O K R O W T E E
```

dio.com

```
STREET
Очередь: T U O K R O W T E E R T S
Очередь: T U O K R O W T E E
```

Структура данных: стеки

Стек является общей структурой для представления данных, которые должны обрабатываться в определенном порядке. Например, когда функция вызывает другую функцию, которая, в свою очередь, вызывает третью функцию, важно, чтобы третья функция вернулась на вторую функцию, а не на первую.

Один из способов реализации такого порядка обработки данных – это организация своего рода очереди вызовов функций. Последняя добавленная в стек функция будет завершена первой, и наоборот: первая добавленная в стек функция будет завершена последней. Таким образом, сама структура данных обеспечивает надлежащий порядок вызовов.

Концептуально, структура данных «стек» очень проста: она позволяет добавлять или удалять элементы в определенном порядке. Каждый раз, когда добавляется элемент, он попадает на вершину стека, а единственный элемент, который может быть удален из стека – тот, что находится на вершине стека. Таким образом, стек, как принято говорить, «первым пришел, последним ушел – FILO» или «последним пришел, первым ушел – LIFO». Первый элемент, добавленный в стек, будет удален из него в последнюю очередь.

Зачем нужны стеки? Стеки – удобный способ организации вызовов функций. В самом деле, «стек вызовов» – это термин, который часто используется для обозначения списка функций, которые сейчас либо выполняются, либо находятся в режиме ожидания возвращаемого значения других функций.

В некотором смысле стеки являются частью фундаментального языка информатики. Когда нужно реализовать очередь типа «первый пришел, последним ушел», то имеет смысл говорить о стеках с использованием общей терминологии. Кроме того, такие очереди участвуют во многих процессах, начиная от теоретических компьютерных наук (например, функции push-down) и других.

Стеки имеют некоторые ассоциируемые методы:

- Push – добавить элемент в стек;
- Pop – удалить элемент из стека;
- Peek – просмотреть элементы стека;
- LIFO – поведение стека,
- FILO Equivalent to LIFO.

Теперь можно разработать программу, которая будет реализовывать работу структуры данных «стек». Создадим шаблон класса Stack, то есть универсальный стек, который будет соответствовать методике обобщенного программирования.

Этот стек реализован с шаблонами, чтобы его можно было использовать практически для любых типов данных. Причем размер стека определяется динамически, во время выполнения программы. 8 стек добавлена также дополнительная функция peek(), которая показывает n-й элемент от вершины стека.

```
1   #ifndef STACK_H
2   #define STACK_H
3
4   #include <cassert> // для assert
5   #include <iostream>
6
7   #include <iomanip> // для setw
8
9   template <typename T>
10  class Stack
11  {
12  private:
```

```
13      T *stackPtr;                    // указатель на стек
14      const int size;                 // максимальное количество элемен-
    тов в стеке
15      int top;                      // номер текущего элемента стека
16  public:
17      Stack(int = 10);                // по умолчанию, размер стека ра-
    вен 10 элементам
18      Stack(const Stack<T> &);        // конструктор копирования
19      ~Stack();                     // деструктор
20
21      inline void push(const T & );     // поместить элемент в верши-
    ну стека
22      inline T pop();                 // удалить элемент из вершины сте-
    ка и вернуть его
23      inline void printStack();       // вывод стека на экран
24      inline const T &Peek(int ) const;  // n-й элемент от вершины
    стека
25      inline int getStackSize() const;   // получить размер стека
26      inline T *getPtr() const;       // получить указатель на стек
27      inline int getTop() const;      // получить номер текущего эле-
    мента в стеке
28  };
29
30  // реализация методов шаблона класса STack
31
32  // конструктор стека
33  template <typename T>
34  Stack<T>::Stack(int  maxSize) :
35      size(maxSize) // инициализация константы
36  {
37      stackPtr = new T[size];  // выделить память под стек
38      top = 0;  // инициализируется текущий элемент нулем;
39  }
40
41  // конструктор копирования
42  template <typename T>
43  Stack<T>::Stack(const Stack<T> & otherStack) :
44      size(otherStack.getStackSize()) // инициализация константы
45  {
```

```
46      stackPtr = new T[size];  // выделить память под новый стек
47      top = otherStack.getTop();
48
49      for(int ix = 0;  ix < top;  ix++)
50          stackPtr[ix] = otherStack.getPtr()[ix];
51  }
52
53  // функция деструктора стека
54  template <typename T>
55  Stack<T>::~Stack()
56  {
57      delete [] stackPtr;  // удаляется стек
58  }
59
60  // функция добавления элемента в стек
61  template <typename T>
62  inline void Stack<T>::push(const T &value)
63  {
64      // проверяется размер стека
65      assert(top < size);  // номер текущего элемента должен быть
    меньше размера стека
66
67      stackPtr[top++] = value;  // элемент помещается в стек
68  }
69
70  // функция удаления элемента из стека
71  template <typename T>
72  inline T Stack<T>::pop()
73  {
74      // проверяется размер стека
75      assert(top > 0);  // номер текущего элемента должен быть
    больше 0
76
77      stackPtr[--top];  // удаляется элемент из стека
78  }
79
80  // функция возвращает n-й элемент от вершины стека
81  template <class T>
82  inline const T &Stack<T>::Peek(int nom) const
```

```
83   {
84    //
85     assert(nom <= top);
86
87     return stackPtr[top - nom];  // вернуть n-й элемент стека
88   }
89
90   // вывод стека на экран
91   template <typename T>
92   inline void Stack<T>::printStack()
93   {
94      for (int ix = top - 1;  ix >= 0;  ix--)
95          cout << "|" << setw(4) << stackPtr[ix] << endl;
96   }
97
98   // вернуть размер стека
99   template <typename T>
100  inline int Stack<T>::getStackSize() const
101  {
102     return size;
103  }
104
105  // вернуть указатель на стек (для конструктора копирования)
106  template <typename T>
107  inline T *Stack<T>::getPtr() const
108  {
109     return stackPtr;
110  }
111
112  // вернуть размер стека
113  template <typename T>
114  inline int Stack<T>::getTop() const
115  {
116     return top;
117  }
118
119  #endif // STACK_H
```

Шаблон класса Stack реализован в отдельном *.h файле. Все дело в том, что и интерфейс шаблона класса, и реализация должны находиться в одном файле, иначе можно увидеть список ошибок похожего содержания:

```
ошибка undefined reference to «метод шаблона класса»
```

Интерфейс шаблона класса объявлен с 9-й по 28-ю строки. Все методы класса содержат комментарии и описывать их работу отдельно не имеет смысла. Нужно обратить внимание на то, что все методы шаблона класса «стек» объявлены как inline – функции. Это сделано для того, чтобы ускорить работу класса, так как встроенные функции класса работают быстрее, чем внешние.

Сразу после интерфейса шаблона идет реализация методов класса «стек», строки 32 – 117. В реализации методов класса ничего сложного нет, если знать, как устроены стек, шаблоны и классы. В классе есть два конструктора, первый объявлен в строках 32 – 33 – это конструктор по умолчанию. А вот конструктор в строках 41 – 45 – это конструктор копирования. Он нужен для того, чтобы скопировать один объект в другой. Метод Peek, строки 80 – 88, предоставляет возможность просматривать элементы стека. Необходимо просто ввести номер элемента, отсчет идет от вершины стека. Остальные функции являются служебными, то есть предназначены для использования внутри класса, конечно же, кроме функции printStack(), которая выводит элементы стека на экран.

Далее представлен драйвер для стека, то есть программа, в которой тестируется работа класса. Это main функция, в которой будет тестироваться шаблон класса Stack. Смотрим код ниже:

```
1   #include <iostream>
2
3    using namespace std;
4
5   #include "stack.h"
6
7   int main()
```

```
8    {
9       Stack<char> stackSymbol(5);
10      int ct = 0;
11      char ch;
12
13      while (ct++ < 5)
14      {
15         cin >> ch;
16         stackSymbol.push(ch);  // элементы помещаются в стек
17      }
18
19      cout << endl;
20
21      stackSymbol.printStack();  // печать стека
22
23      cout << "\n\nУдалим элемент из стека\n";
24       stackSymbol.pop();
25
26      stackSymbol.printStack();  // печать стека
27
28       Stack<char> newStack(stackSymbol);
29
30      cout << "\n\nСработал конструктор копирования!\n";
31       newStack.printStack();
32
33      cout << "Второй в очереди элемент: "<< newStack.Peek(2) <<
34  endl;
35
36      return 0;
37   }
```

Создали объект стека, строка 9, размер стека при этом равен 5, то есть стек может поместить не более пяти элементов. Заполняем стек в цикле while, строки 13 – 17. В строке 21 выводим стек на экран, после удаляем один элемент из стека (строка 24) и снова выводим содержимое стека, которое изменилось на один элемент.

Смотрим результат работы программы:

CppStudio.com

```
LOTR! Очередь:  пустая !
|  R
|  T
|  O
|  L

Удалим элемент из стека
|  R
|  T
|  O
|  L

Сработал конструктор копирования!
|  R
|  T
|  O
|  L
```

В строке 28 воспользуемся конструктором копирования, а в строке 33 – функцией peek().

Стек получился и исправно работает, можно попробовать его протестировать, например, на типе данных int.

Длиная арифметика в C++

Решая задачи, многим приходится сталкиваться с тем, что просто не хватает размерностей типов для, казалось бы, простейших операций: сложения, вычитания и умножения. Все эти операции знакомы всем с ранних классов. Но что делать, если одну из этих операций необходимо применить для огромных чисел, скажем так, в 1000 или более знаков?

Каждую арифметическую операцию рассмотрим отдельно, предварительно написав на языке C++ код реализации. Прежде всего, необходимо понимать, что бесконечно длинное число можно представить только в виде динамического массива, что предстоит сделать. Но даже если числа представлять в виде динамических массивов, все равно будут накладываться некоторые ограничения. Например, длина такого числа будет ограничена объемом памяти компьютера. Также

следует понимать, что при использовании операций сложения и умножения результат будет занимать больше места в памяти компьютера, нежели операнды.

Сложение длиных чисел

Рассмотрим арифметическую операцию сложения, применяемую в длинной арифметике. Алгоритм этого нехитрого арифметического действия, на удивление, простой. Он выглядит так:

```
1   // определяем длину массива суммы
2   if (size_a > size_b)
3      length = size_a + 1;
4   else
5      length = size_b + 1;
6
7   for (int ix = 0;  ix < length;  ix++)
8    {
9      b[ix] += a[ix];  // сумма последних разрядов чисел
10     b[ix + 1] += (b[ix] / 10);  // если есть разряд для переноса, пе-
    ренос его в следующий разряд
11     b[ix] %= 10;  // если есть разряд для переноса, он отсекается
12  }
13
14  if (b[length - 1] == 0)
15     length--;
```

Числа, которые будем складывать, предположительно записаны в массивы a и b. Необходимо учесть, что они записаны «зеркально», то есть первый элемент массива соответствует последней цифре соответствующего числа, второй элемент – предпоследней, и т.д. Размеры длин чисел хранятся в переменных size_a и size_b, но можно использовать любые другие. Оператор выбора if (строки 2 – 5) нужен для определения максимальной длины числа, полученного в результате суммирования. Чаще всего суммируемые числа разной длины, одно больше, другое меньше, а нужно выделить память так, чтобы каждое число вместилось.

Далее, в алгоритме, делаем так, как на уроках математики: сначала складываем отдельные разряды, начиная с конца, строка 9; делим получившуюся сумму на 10 и получаем целую часть от деления на десять, которую сразу прибавляем к следующему разряду, строка 10.

В строке 11 отсекаем первый разряд полученного числа, если конечно он есть. Главное – не забыть, что число будет храниться в массиве b и выводить его следует с конца.

Вычитание длиных чисел

Вторая, наиболее используемая арифметическая операция – это вычитание. Будем считать, что числа хранятся в массивах a и b, m и n – длины этих чисел соответственно. Следует учесть, что числа записаны «зеркально» (см. выше). Конечно, если знать, какое число больше, то задача упрощается. Но если не знать этого, тогда сначала следует найти, какое из чисел больше. Это понадобится для определения знака получившегося числа, то есть если первое число меньше второго, то в ответе появится минус. Итак, приступим к написанию первой части алгоритма, то есть определению большего числа. Алгоритм выглядит так:

```
1     int k = 3;  // если к == 3, значит, числа одинаковой длины
2     length = size_a;
3     if (size_a > size_b)
4     {
5        length = size_a;
6        k = 1;  // если к == 1, значит, первое число длиннее второго
7     }
8     else
9        if (size_b > size_a)
10       {
11          length = size_b;
12          k = 2;  // если к == 2, значит, второе число длиннее первого
13       }
14       else // если числа одинаковой длины, то необходимо сравнить
      их веса
15          for (int ix = 0;  ix < length; ) // поразрядное сравнение весов
      чисел
16          {
17             if (a[ix] > b[ix]) // если разряд первого числа больше
18             {
19                k = 1;  // значит, первое число длиннее второго
20                break;  // выход из цикла for
21             }
22
```

```
23              if(b[ix] > a[ix]) // если разряд второго числа больше
24              {
25                 k = 2;  // значит, второе число длиннее первого
26                 break;  // выход из цикла for
27              }
28          } // конец for
```

Рассмотрим алгоритм. Сначала можно увидеть, что переменной k придается значение 3. В данной части алгоритма переменная k является флагом результата проверки. Если числа равны, то k останется равно 3-м, если первое больше второго, то k примет значение 1, если второе больше первого, то k примет значение 2. Переменная length примет значение длины большего числа. Теперь перейдем к обоснованию работоспособности этого алгоритма. Сравнение чисел происходит в два этапа. Сначала сравниваются длины чисел: какое число длиннее, то и больше, строки 1 – 11. Если числа одинаковой длины, то можно переходить к поразрядовому сравнению, строки 13 – 26. Начинаем по порядку сравнивать разряды, начиная с самого старшего, так определяется больший вес числа. В этом и заключается суть и сложность первой части. Теперь перейдем ко второй части алгоритма – вычитанию. Оно выглядит так:

```
1     int difference (int *x, int *y, int *z, int length)
2     {
3        for (int ix = 0;  ix < (length - 1);  ix++) // проход по всем
      //разрядам числа, начиная с последнего, не доходя до первого
4        {
5           if (ix < (length - 1)) // если текущий разряд чисел не первый
6           {
7              x[ix + 1]--;  // в следующуем разряде большего числа
      // занимаем 1.
8              z[ix] += 10 + x[ix];  // в ответ записываем сумму
      // значения текущего разряда большего числа и 10-ти
9
10          } else  // если текущий разряд чисел – первый
11                z[ix] += x[ix];  // в ответ суммируем значение текущего
      // разряда большего числа
12
13          z[ix] -= y[ix];  // вычитаем значение текущего разряда
      // меньшего числа
```

```
14
15          if (z[ix] / 10 > 0) // если значение в текущем разряде
        // двухразрядное
16          {
17              z[ix + 1]++; // переносим единицу в старший разряд
18              z[ix] %= 10; // в текущем разряде отсекаем ее
19          }
20      }
21      return 0;
22  }
```

Для самого вычитания удобно написать функцию, ведь тогда не придется писать два алгоритма для двух случаев: когда первое число больше второго, и наоборот. В массиве *x* содержится большее число, в массиве *y* – меньшее, в массиве *z* – результат. Алгоритм довольно простой: для каждого разряда добавляем 10, с учетом вычитания из старшего разряда – 1. Это делается для упрощения вычитания разрядов. Данная операция делается лишь в том случае, когда рассматриваемый разряд не является последним в массиве (первым в числе). После вычитания разрядов проверяем получившееся число в данном разряде в массиве *z*. Ответ запишется в массив *z*, причем «зеркальным» (см. выше) способом. Процедуру следует вызывать следующим образом:

```
1   if (k == 1) difference(a,b,c, length);  – если первое число больше
    второго,
2   if (k == 2) difference(b,a,c, length);  – если второе число больше
    первого.
```

Теперь ответ будет храниться в массиве *c*, все в том же «зеркальном» порядке. Таким вот образом можно вычитать большие и огромные числа.

Умножение длиных чисел

Теперь рассмотрим произведение чисел. Этот алгоритм чаще двух предыдущих можно встретить при решении задач. Перейдем непосредственно к самому алгоритму.

Алгоритм выглядит так:

```
1   length = size_a + size_b + 1;
2
3   for (int ix = 0;  ix < size_a;  ix++)
4      for (int jx = 0;  jx < size_b;  jx++)
5         c[ix + jx - 1] += a[ix] * b[ix];
6
7   for (int ix = 0;  ix < length;  ix++)
8    {
9      c[ix + 1] +=  c[ix] / 10;
10     c[ix] %= 10;
11  }
12
13  while (c[length] == 0)
14     length-- ;
```

Вот так выглядит алгоритм задачи. Теперь попробуем разобраться, как он работает. Сначала имелось два числа в массивах a и b все в том же «зеркальном» (см. выше) виде.

Длины чисел хранятся в переменных size_a и size_b. В переменной length хранится длина результирующего числа.

Она будет равна либо сумме длин первоначальных чисел, либо этой сумме, увеличенной на единицу. Но так как точная длина полученного числа неизвестна, то возьмем длину побольше, то есть второй вариант. Теперь, после этих нехитрых подсчетов, приступим к перемножению чисел. Будем их перемножать так, как учили еще в школе. Для этого запускаем два цикла: один до size_a, другой до size_b. После этих циклов можно увидеть еще один – до length (строка 14). Благодаря этому циклу в записи числа в массиве (в каждой ячейке массива) получается по одной цифре полученного произведения. Последний цикл нужен, чтобы узнать точную длину полученного числа, ведь предположенная длина числа может быть больше действительной. Ответ будет храниться в массиве *c*, все в том же «зеркальном» виде.

Вот и весь алгоритм. Его проще понять, когда он реализован на языке программирования, в данном случае – это C++.

Контрольные тесты

1. Сколько операций отношения содержится в языке C ?

а) 7;

б) 12;
в) 6;
г) 18.

2. Что означает сочетание клавиш != в языке С ?
а) предупреждение;
б) не равно;
в) начало строки;
г) конец строки.

3. Что означает сочетание символов == в языке С ?
а) начало или конец строки;
б) два знака равно;
в) межстрочный интервал;
г) равно.

4. Где используются операции отношения в языке С ?
а) в вычислениях дискриминанта;
б) в условных выражениях;
в) при работе на клавиатуре;
г) при выводе информации.

5. Что представляет собой результат условного выражения в языке С ?
а) истину – true или ложь – false;
б) неравенство;
в) вывод;
г) высказывание.

6. Какому числу соответствует «истина» в языке С ?
а) нулю;
б) единице;
в) отрицательному;
г) случайному.

7. Выберите, пожалуйста, среди всех указанных величин только операции отношения:
а) меньше;
б) или;
в) больше;
г) не;
д) меньше или равно;
е) равно;
ж) не равно;
з) больше или равно.

3. СИ ШАРП (C#)

3.1. Введение в C Sharp и .Net

C# (произносится Си-Шарп) – это новый язык программирования от компании «Microsoft», входящий в новую версию Visual Studio – Visual Studio.NET. Кроме C#, в Visual Studio.NET входят также Visual Basic.NET и Visual C++.

Одна из причин разработки нового языка компанией «Microsoft» – это создание компонентно-ориентированного языка для новой платформы .NET. Другие языки были созданы до появления платформы .NET, язык же C# создавался специально под эту платформу и не несет с собой груза совместимости с предыдущими версиями языков. Хотя это и не означает, что для новой платформы этот язык единственный.

Еще одна из причин разработки компанией «Microsoft» нового языка программирования – это создание альтернативы языку Java. Как известно, реализация Java у «Microsoft» не была лицензионно чистой – «Microsoft» в присущей ей манере существенно изменила свою реализацию. Компания «Sun», владелица Java, подала на «Microsoft» в суд, и «Microsoft» этот суд проиграла. Тогда «Microsoft» решила вообще отказаться от Java и создать свой Java-подобный язык, который и получил название C#.

Если перевести слова NET Runtime на русский язык, то получается что-то вроде «Среда выполнения». Именно в этой среде и выполняется код, получаемый в результате компиляции программы, написанной на C#. NET Runtime, основанный не на ассемблере (т. е. не на коде, родном для процессора), а на некотором промежуточном коде. Отдаленно он напоминает виртуальную Java-машину. Только если в случае Java был только один язык для виртуальной машины, то для NET Runtime таких языков может быть несколько. Теоретически программа для среды NET Runtime может выполняться под любой операционной системой, в которой NET Runtime установлена. Но на практике единственная пока платформа для этого – это Windows.

3.2. Основные понятия

Assembly (сборка) – это базовый строительный блок приложения в .NET Framework. Управляемые модули объединяются в сборки. Сборка является логической группировкой одного или нескольких

управляемых модулей или файлов-ресурсов. Управляемые модули в составе сборок исполняются в Среде Времени Выполнения (CLR). Сборка может быть либо исполняемым приложением (при этом она размещается в файле с расширением .exe), либо библиотечным модулем (в файле с расширением .dll). При этом ничего общего с обычными (старого образца) исполняемыми приложениями и библиотечными модулями сборка не имеет.

Managed Code (управляемый код) – это код, который выполняется в среде CLR. Код строится на основе объявляемых в исходном модуле структур и классов, содержащих объявления методов. Управляемому коду должен соответствовать определенный уровень информации (метаданных) для среды выполнения. Коды C#, Visual Basic и JScript являются управляемыми по умолчанию. Код Visual C++ не является управляемым по умолчанию, но компилятор может создавать управляемый код, для этого нужно указать аргумент в командной строке(/CLR). Одной из особенностей управляемого кода является наличие механизмов, которые позволяют работать с управляемыми данными.

Managed Data (управляемые данные) – это объекты, которые в ходе выполнения кода модуля размещаются в управляемой памяти (в управляемой куче) и уничтожаются сборщиком мусора CLR. Данные C#, Visual Basic и JScript .NET являются управляемыми по умолчанию. Данные C# также могут быть помечены как неуправляемые.

GAC (Global Assembly Cache – общий кэш сборок). Для выполнения .NET-приложения достаточно разместить относящиеся к данному приложению сборки в одном каталоге. Если при этом сборка может быть использована в нескольких приложениях, то она размещается и регистрируется с помощью специальной утилиты в GAC.

CTS – Common Type System (общая система типов). Поддерживается всеми языками платформы. В силу того, что .NET основана на парадигме ООП, речь здесь идет об элементарных типах, классах, структурах, интерфейсах, делегатах и перечислениях. Common Type System является важной частью среды выполнения, определяет структуру синтаксических конструкций, способы объявления, использования и применения общих типов среды выполнения. В CTS сосредоточена основная информация о системе общих предопределенных типов, об их использовании и управлении (правилах преобразования значений). CTS играет важную роль в деле интеграции разноязыких управляемых приложений.

Namespace (пространство имен) – это способ организации системы типов в единую группу. В рамках .NET существует единая (общеязыковая) библиотека базовых классов. Концепция пространства имен обеспечивает эффективную организацию и навигацию по этой библиотеке. Вне зависимости от языка программирования доступ к определенным классам обеспечивается за счет их группировки в рамках общих пространств имен.

NET Runtime состоит из нескольких частей. Одна из них – Common Language Runtime. Это некоторый набор стандартов, котрые должны поддерживать все языки платформы .NET. Например, в предыдущих версиях Visual Studio была такая проблема, что различные языки по-разному хранили данные одного типа. Так, скажем, тип целого в Visual Basic'е занимал два байта, а в Visual C++ – четыре. А это порождало кучу проблем при совместном использовании языков. Так вот, Common Language Runtime как раз, в частности, и определяет стандартные для всех языков .NET типы данных. И уже есть гарантии, что целый тип в одном языке будет в точности соответствовать одноименному типу в другом.

Основные функциональные элементы среды представлены на рис. 3.1.

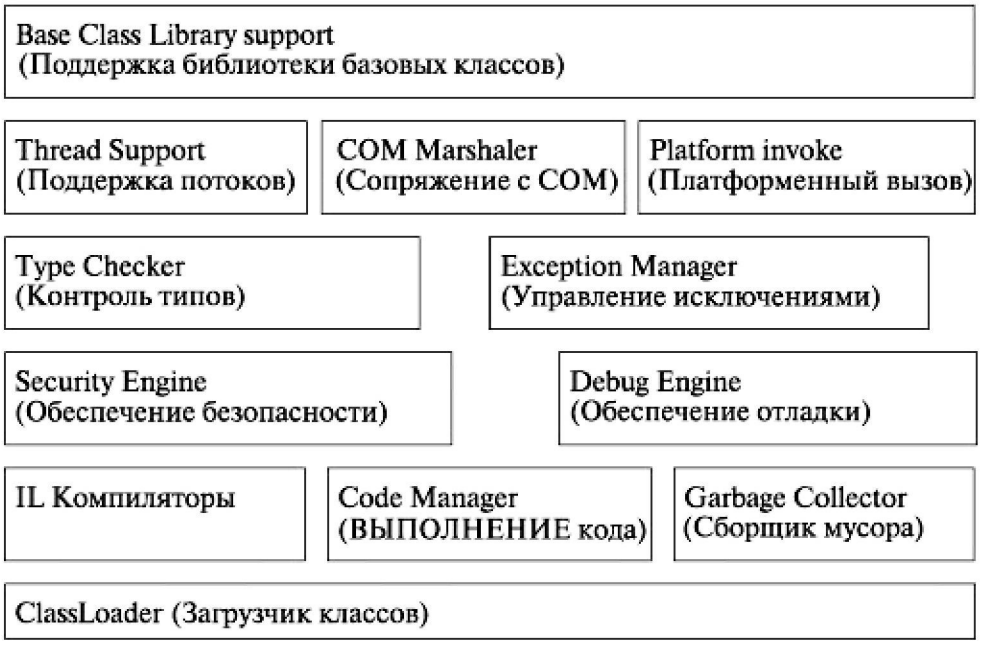


Рис. 3.1. Структура среды выполнения CLR

Строгий контроль типов, в частности, предполагает проверку соответствия типа объекта диапазону значений, которые могут быть присвоены данному объекту.

Защита .NET (безопасность) строится поверх системы защиты операционной системы компьютера. Она не дает пользователю или коду делать то, что делать не позволено, и накладывает ограничения на выполнение кода. Например, можно запретить доступ некоторым секциям кода к определенным файлам.

Функциональные блоки CLR Code Manager и Garbage Collector работают совместно: Code Manager обеспечивает размещение объектов в управляемой памяти, Garbage Collector – освобождает управляемую память.

Exception Manager включает следующие компоненты:

· finally handler (обеспечивает передачу управления в блок finally);

· fault handler (включается при возникновении исключения);

· type-filtered handler (обеспечивает выполнение кода соответствующего блока обработки исключения);

· user-filtered handler (выбор альтернативного блока исключения).

Ниже представлена схема выполнения .NET-приложения в среде CLR (рис. 3.2).

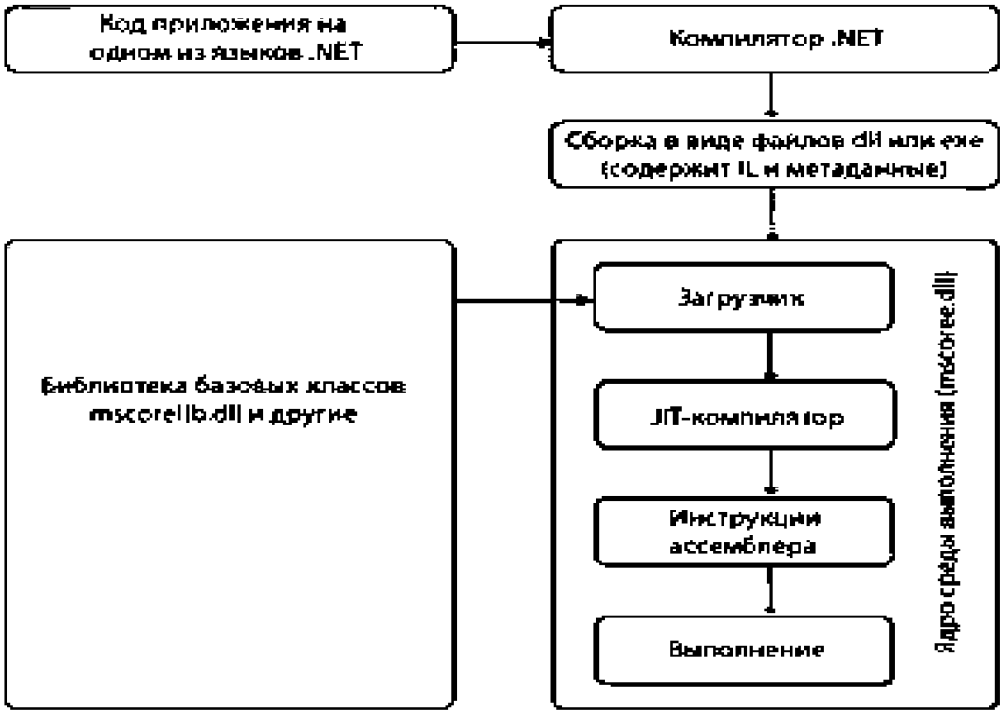


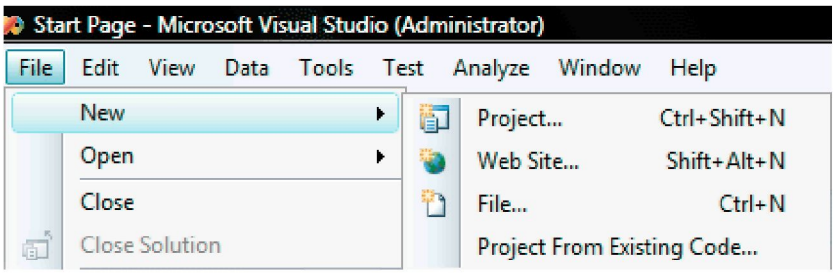
Рис. 3.2. Схема выполнения NET-приложения в среде CLR

Еще одна важная часть NET Runtime – это набор базовых классов. Их очень много (порядка несколько тысяч). Кроме того, эти классы

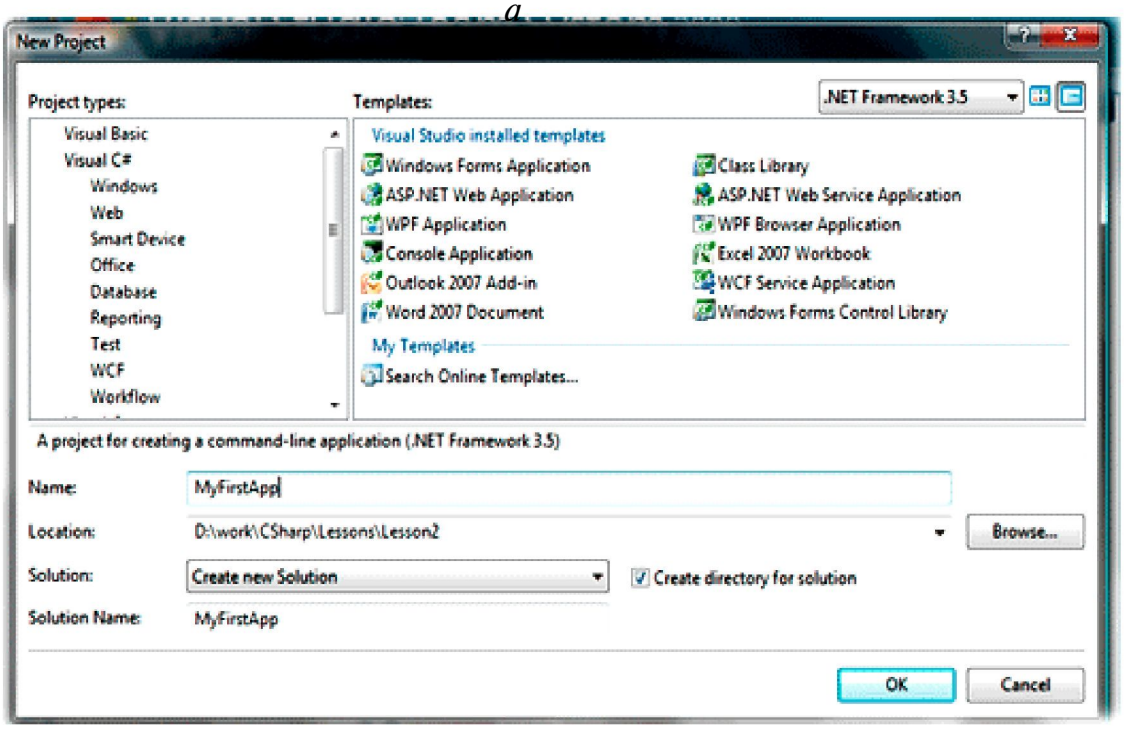
относятся не к конкретному языку, а к NET Runtime. То есть получаем набор классов, общий для всех языков .NET, что достаточно удобно.

Все примеры будут выполняться в среде Microsoft Visual Studio, но это не значит, что нельзя будет использовать какую-либо другую из известных сред. Если нет среды VS.NET, с сайта производителя можно скачать бесплатную редакцию среды разработки.

Пишем первую программу на C#. Для этого запускается Visual Studio.NET. Для создания нового пустого проекта C# выбираем пукт меню New Project или нажимаем комбинацию клавиш Ctrl+Shift+N либо просто заходим в меню File и далее выбираем New и затем Project (рис. 3.3, *а*).



а



б

Рис. 3.3. Создание нового проекта

В появившемся окне New Project слева выбираем, естественно, Visual C#, а справа – тип приложения Console Application (см. рис. 3.3, *б*).

В качестве имени проекта (Name) напечатаем MyFirstApp или что-то в этом роде. Для закрытия данного диалогового окна нажимаем кнопку Ok.

Теперь приступаем к коду. Наша первая программа просто выведет некоторое фиксированное слово в консольное окошко. Вот ее листинг:

```
using System;
namespace first
{
    ///
    /// Summary description for MyFirstClass.
    ///
    class  MyFirstClass
    {
        ///
        /// The main entry point for the application.
        ///
        [STAThread]
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
            //
            Console.WriteLine("Я начинаю изучать C#");

        }
    }
}
```

Запускаем программу, нажав Ctrl+F5. Результат будет таким (рис. 3.4).

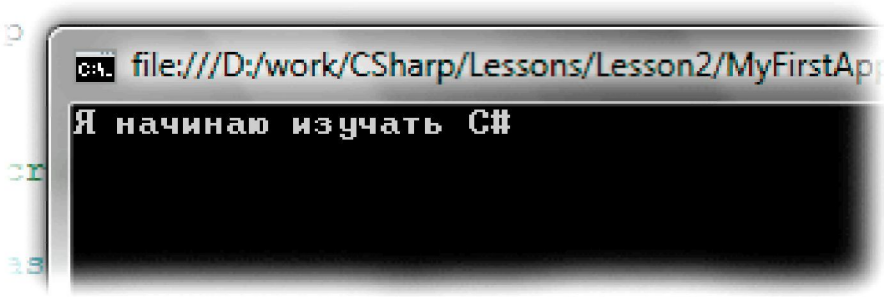


Рис. 3.4. Результат выполнения программы

Разберем текст программы подробнее. Как известно, в .NET Runtime существуют пространства имен. Одно из таких пространств – это System, которое добавляется автоматически в любой проект на C#. В приведенной программе добавлена строчка:

```
...
using System;
...
```

Можно вместо длиных имен использовать более короткие. В частности, вместо System.Console можно писать просто Console. Что и делается в строчке

```
...
Console.WriteLine("Я начинаю изучать C#");
...
```

Далее в программе объявляется класс MyFirstClass. В C# не существует глобальных функций, так что нужно завести сначала класс и затем функцию Main в нем (функция Main обязательно должна быть в каждой программе на C#, именно с этой функции и начинается выполнение программы. Стоит обратить также внимание на то, что эта функция пишется с прописной (большой) буквы. C# различает строчные и прописные буквы, так что это важно). Кроме того, эта функция объявлена с модификатором static. Это означает, что она не относится к конкретному экземпляру класса MyFirstClass, а принадлежит всему классу. В данной функции Main просто выводим на экран некоторую строчку методом *WriteLine*.

Программа выполнилась и быстро закрыла свое окно, для того чтобы такого не произошло, добавим еще одну строку в данную программу:

```
...
Console.WriteLine("Я начинаю изучать C#");
Console.ReadLine();
...
```

ReadLine, как видно из названия, считывает строку.

Теперь, после запуска, окно остается и программа ждет ввода. Для завершения ее работы требуется нажать Enter.

Теперь напишем эту же программу, но для Windows Forms.

Создадим новый проект, но тип проекта выберем «Windows Forms Application». Зададим ему имя MyFirsWinApp, нажав «OK». Появится пустая форма. Поместим на нее кнопку (Button) (рис. 3.5).

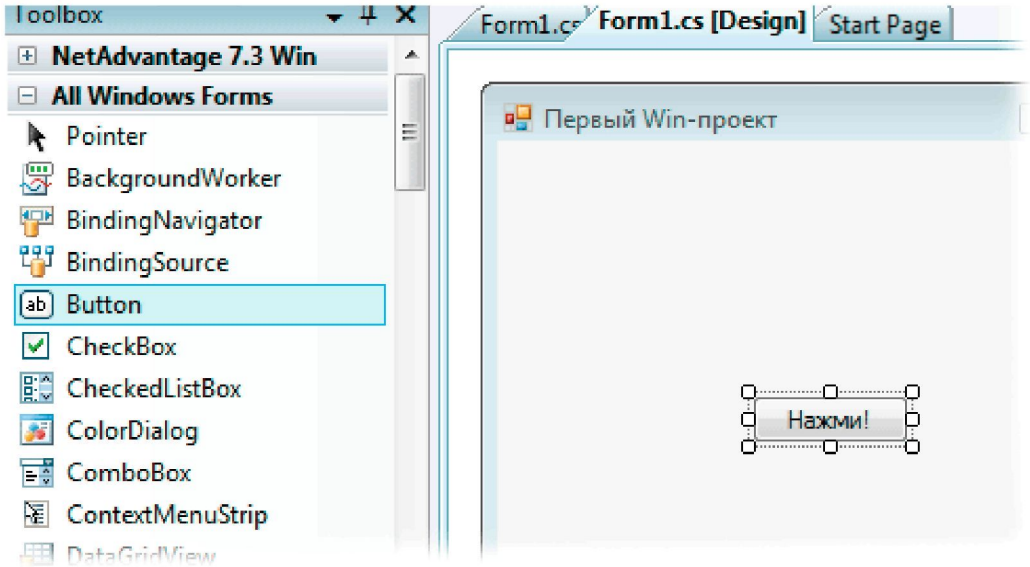


Рис. 3.5. Создание нового проекта MyFirsWinApp

После того, как разместили кнопку, по ней следует кликнуть два раза мышкой, тогда откроется окно редактора кода. Напишем такую строчку:

```
MessageBox.Show ("Я начинаю изучать C#");
```

Можно запускать программу.
Первая программа на C# написана.

3.3. Алфавит и синтаксис C#

АЛФАВИТ C#

Алфавит (или множество литер) языка программирования C# составляют символы таблицы кодов ASCII. Алфавит C# включает:

• строчные и прописные буквы латинского алфавита (будем называть их буквами);

• цифры от 0 до 9 (назовем их буквами-цифрами);

• символ «_» (подчеркивание – также считается буквой);

• набор специальных символов: " { }, 1 [ ] + – %/ \; ' : ? < > = ! & # ~ *_ ;

• прочие символы.

Алфавит C# служит для построения слов, которые в C++ называются лексемами. Различают пять типов лексем:

• идентификаторы;

• ключевые слова;

• знаки (символы) операций;
• литералы;
• разделители.

Почти все типы лексем (кроме ключевых слов и идентификаторов) имеют собственные правила словообразования, включая собственные подмножества алфавита.

Лексемы обособляются разделителями. Этой же цели служит множество пробельных символов, к числу которых относятся пробел, табуляция, символ новой строки и комментарии.

Правила образования идентификаторов

Рассмотрим правила построения идентификаторов из букв алфавита:

1. Первым символом идентификатора C# может быть только буква.

2. Следующими символами идентификатора могут быть буквы, цифры и нижнее подчеркивание.

3. Длина идентификатора не ограничена.

Вопреки правилам словообразования, в C# существуют ограничения относительно применения подчеркивания в качестве самой первой буквы в идентификаторах. Из-за особенностей реализации использование идентификаторов, которые начинаются с этого символа, нежелательно.

Рекомендации по наименованию объектов

Имена – это идентификаторы. Любая случайным образом составленная последовательность букв, цифр и знаков подчеркивания с точки зрения грамматики языка идеально подходит на роль имени любого объекта, если только начинается с буквы. Фрагмент программы, содержащий подобную переменную, будет синтаксически безупречен.

И все же имеет смысл воспользоваться дополнительной возможностью облегчить восприятие и понимание последовательностей операторов. Для этого достаточно закодировать с помощью имен содержательную информацию.

Желательно создавать составные осмысленные имена. В этом случае в одно слово можно «втиснуть» предложение, которое в доступной форме представит информацию о типе объекта, его назначении и особенностях использования.

Ключевые слова и имена

Часть идентификаторов C# входит в фиксированный словарь ключевых слов. Эти идентификаторы образуют подмножество ключевых слов (они так и называются ключевыми словами). Прочие идентификаторы после специального объявления становятся именами. Имена служат для обозначения переменных, типов данных, функций.

Ниже приводится таблица со списком ключевых слов. Нельзя использовать эти имена для образования классов, функций, переменных и других языковых структур.

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	u short
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

Комментарии

Часто бывает полезно вставлять в программу текст, который является комментарием только для читающего программу человека и игнорируется компилятором. В C# это можно сделать одним из двух способов.

Символы /* начинают комментарий, заканчивающийся символами */.

Такая последовательность символов эквивалентна символу пропуска (например, символу пробела). Это особенно полезно для многострочных комментариев и изъятия частей программы при редактировании, однако следует помнить, что комментарии /* */ не могут быть вложенными.

Символы // начинают комментарий, заканчивающийся в конце строки, на которой они появились. И здесь вся последовательность символов эквивалентна пропуску. Этот способ наиболее полезен для коротких комментариев. Символы // можно использовать для того, чтобы закомментировать символы /* или */, а символами /* можно закомментировать //.

Литералы

В C# существует четыре типа литералов:

- целочисленный;
- вещественный;
- символьный;
- строковый.

Литералы – это особая категория слов языка. Для каждого подмножества литералов используются собственные правила словообразования. Не будем приводить их здесь, ограничимся лишь общим описанием структуры и назначения каждого подмножества литералов. После этого правила станут более-менее понятны.

Целочисленный литерал служит для записи целочисленных значений и является соответствующей последовательностью цифр (возможно, со знаком '-'). Целочисленный литерал, начинающийся со знака 0, воспринимается как восьмеричное целое. В этом случае цифры 8 и 9 не должны встречаться среди составляющих литерал символов. Целочисленный литерал, начинающийся с Ox или OX, воспринимается как шестнадцатеричное целое. В этом случае такой литерал может включать символы от A или a, до F или f, которые в шестнадцатеричной системе эквивалентны десятичным значениям от 10 до 15. Непосредственно за литералом могут располагаться в произвольном сочетании один или два специальных суффикса: U (или и) и L (или l).

Вещественный литерал служит для отображения вещественных значений. Он фиксирует запись соответствующего значения в обычной десятичной или научной нотациях. В научной нотации мантисса

отделяется от порядка литерой E (или e). Непосредственно за литералом может располагаться один из двух специальных суффиксов: F (или f) и L (или l).

Значением символьного литерала является соответствующее значение ASCII кода (это, разумеется, не только буквы, буквы-цифры или специальные символы алфавита C#). Символьный литерал представляет собой последовательность одной или нескольких литер, заключенных в одинарные кавычки. Символьный литерал служит для представления литер в одном из форматов представления. Например, литера Z может быть представлена литералом «Z», а также литералами «\132» и «\x5A». Любая литера может быть представлена в нескольких форматах представления: обычном, восьмеричном и шестнадцатеричном.

Строковые литералы являются последовательностью (возможно, пустой) литер в одном из возможных форматов представления, заключенных в двойные кавычки. Строковые литералы, расположенные последовательно, соединяются в один литерал, причем литеры соединенных строк остаются различными. Так, последовательность строковых литералов «\xF» «F» после объединения будет содержать две литеры, первая из которых является символьным литералом в шестнадцатеричном формате «\xF», вторая – символьным литералом «F». Строковый литерал и объединенная последовательность строковых литералов заканчиваются пустой литерой, которая используется как индикатор конца литерала.

Типы данных C#

C# является жестко типизированным языком. При его использовании необходимо объявлять тип каждого объекта, который создается (например, целые числа, числа с плавающей точкой, строки, окна, кнопки, и т. д.), и компилятор поможет избежать ошибок, связанных с присвоением переменным значений только того типа, который им соответствует. Тип объекта указывает компилятору размер объекта (например, объект типа int занимает в памяти 4 байта) и его свойства (например, форма может быть видима и невидима, и т.д.).

Подобно языкам C++ и Java, C# подразделяет типы на два вида: встроенные типы, которые определены в языке, и определяемые пользователем типы, которые выбирает программист.

C# также подразделяет типы на две другие категории: размерные и ссылочные. Основное различие между ними – это способ, которым

их значения сохраняются в памяти. Размерные типы сохраняют свое фактическое значение в стеке. Ссылочные типы хранят в стеке лишь адрес объекта, а сам объект сохраняется в куче. Куча – основная память программ, доступ к которой осуществляется намного медленнее, чем к стеку. Если работать с очень большими объектами, то сохранение их в куче имеет много преимуществ.

C# также поддерживает и указатели на типы, но они редко употребляются. Применение указателей связано с использованием неуправляемого кода.

3.4. Особенности использования стека и кучи

Стек – это структура данных, которая сохраняет элементы по принципу: первым пришел, последним ушел (полная противоположность очереди). Стек относится к области памяти, поддерживаемой процессором, в которой сохраняются локальные переменные. Доступ к стеку во много раз быстрее, чем к общей области памяти, поэтому использование стека для хранения данных ускоряет работу программы. В C# размерные типы (например, целые числа) располагаются в стеке: для их значений зарезервирована область в стеке, и доступ к ней осуществляется по названию переменной.

Ссылочные типы (например, объекты) располагаются в куче. Куча – это оперативная память компьютера. Доступ к ней осуществляется медленнее, чем к стеку. Когда объект располагается в куче, то переменная хранит лишь адрес объекта. Этот адрес хранится в стеке. По адресу программа имеет доступ к самому объекту, все данные которого сохраняются в общем куске памяти (куче).

«Сборщик мусора» уничтожает объекты, располагающиеся в стеке, каждый раз, когда соответствующая переменная выходит за область видимости. Таким образом, если объявить локальную переменную в пределах функции, то объект будет помечен как объект для «сборки мусора» и будет удален из памяти после завершения работы функции.

Объекты в куче тоже очищаются сборщиком мусора после того, как конечная ссылка на них будет разрушена.

Встроенные типы

Язык C# предоставляет программисту широкий спектр встроенных типов, которые соответствуют CLS (Common Language Specification) и отображаются на основные типы платформы .NET.

Это гарантирует, что объекты, созданные на C#, могут успешно использоваться наряду с объектами, созданными на любом другом языке программирования, поддерживающем .NET CLS (например, VB.NET).

Каждый тип имеет строго заданный для него размер, который не может изменяться. В отличие от языка C++, в C# тип int всегда занимает 4 байта, потому что отображается к Int32 в .NET CLS. Представленная ниже табл. 3.1 содержит список всех встроенных типов, предлагаемых C#.

Таблица 3.1

Список встроенных типов, предлагаемых C#

Тип	Область значений	Размер
sbyte	-128 до 127	Знаковое 8-бит целое
byte	0 до 255	Беззнаковое 8-бит целое
char	U+0000 до U+ffff	16-битовый символ Unicode
bool	true или false	1 байт
short	-32768 до 32767	Знаковое 16-бит целое
ushort	0 до 65535	Беззнаковое 16-бит целое
int	-2147483648 до 2147483647	Знаковое 32-бит целое
uint	0 до 4294967295	Беззнаковое 32-бит целое
long	-9223372036854775808 до 9223372036854775807	Знаковое 32-бит целое
ulong	0 до 18446744073709551615	Беззнаковое 32-бит целое
float	±1,5*10-45 до ±3,4*1033	4 байта, точность – 7 разрядов
double	±5*10-324 до ±1,7*10306	8 байт, точность –16 разрядов
decimal		12 байт, точность – 28 разрядов

В дополнение к этим примитивным типам C# может иметь объекты типа enum и struct.

Преобразование встроенных типов

Объекты одного типа могут быть преобразованы в объекты другого типа неявно или явно. Неявные преобразования происходят автоматически, компилятор делает это вместо программиста. Явные преобразования осуществляются, когда «приводят» значение к дру-

гому типу. Неявные преобразования гарантируют также, что данные не будут потеряны. Например, можно неявно приводить от short (2 байта) к int (4 байта).

Независимо от того, какое значение находится в short, оно не потеряется при преобразовании к int:

```
short x = 1;
int y = x;
//неявное преобразование
```

Если делать обратное преобразование, то, конечно же, можно потерять информацию. Если значение в int больше, чем 32,767, оно будет усечено при преобразовании. Компилятор не станет выполнять неявное преобразование от int к short:

```
short x;
int y - 5;
x = y;
//не скомпилируется
```

Нужно выполнить явное преобразование, используя оператор приведения:

```
short x;
int y - 5;
x = (short) y;
//OK
```

Переменные

Переменная – это расположение в памяти объекта определенного типа. В приведенных выше примерах x и y – переменные. Переменные могут иметь значения, которыми они проинициализированы, или эти значения могут быть изменены программно.

Назначение значений переменным. Чтобы создать переменную, нужно задать тип переменной и затем дать этому типу имя. Можно проинициализировать переменную во время ее объявления или присвоить ей новое значение во время выполнения программы. Вот пример программы, которая в первом случае использует инициализацию

для присвоения значения переменной, во втором – присвоение значения переменной с помощью оператора «=»:

```
class Variables
   {
      static void Main()
      {
        int myInt = 10;
        System.Console.WriteLine("Инициализированная переменная myInt: {0} ", myInt)
;
        myInt = 5;
        System.Console.WriteLine("Переменная myInt после присвоения значения: {0}",
myInt);
        Console.ReadLine();
      }

   }
```

Результат работы этой программы будет следующий:

Инициализированная переменная myInt: 10

myInt после присвоения значения: 5

Здесь строка:

int myInt = 10;

означает объявление и инициализацию переменной myInt. Строка:

myInt= 5 ;

означает присвоение переменной myInt значения 5.

Определение значений переменных

C# требует определения значений, то есть переменные перед использованием должны быть инициализированы. Чтобы проверить это правило, рассмотрим следующий пример:

```
class Variables
  {
     static void Main()
     {
        int myInt;
        System.Console.WriteLine("Неинициализированная переменная myInt: {0}", myInt);
        myInt = 5;
        System.Console.WriteLine("После присвоения переменной myInt: {0}", myInt);

     }
  }
```

Если попробовать скомпилировать этот пример, компилятор отобразит следующее сообщение об ошибке:

error CS3165: Use of unassigned local variable 'myInt'

Нельзя использовать неинициализированную переменную в C#. Может сложиться впечатление, что нужно инициализировать каждую переменную в программе. Это не так. Необходимо лишь назначить переменной значение, прежде чем ее использовать. Ниже представлен пример правильной программы без использования инициализации переменной

```
class Variables
   {
      static void Main()
      {
         int myInt;
         myInt = 10;
         System.Console.WriteLine("Инициализированная переменная myInt: {0}", myInt);

         myInt = 5;
         System.Console.WriteLine("После присвоения значения, myInt: {0}", myInt);

      }
   }
```

В данном примере вместо инициализации выбирается присвоение значения переменной myInt до ее использования:

```
myInt = 10;
```

Константы

Константа – это переменная, значение которой не может быть изменено. Переменные – это более гибкий способ хранения данных. Однако иногда нужно гарантировать сохранение значения определенной переменной. Например, число π. Как известно, значение этого числа никогда не изменяется. Следовательно, нужно гарантировать, что переменная, хранящая это число, не изменит своего значения на протяжении всей работы программы. Программа будет лучше читаемой, если вместо записи:

```
y = x*3.14159265358979323846264338327 95
```

использовать переменную pi, которая хранит значение π. В таком случае используемой переменной должна быть константа:

```
const double pi = 3.14159265358979323846264338327 95;
y = x * pi;
```

Существует три разновидности константы: литералы, символические константы и перечисления. Рассмотрим следующий случай:

```
x = 100;
```

Значение 100 – это литеральная константа. Нельзя установить новое значение на 100 и сделать так, чтобы 100 представляло значение 99. Символические константы устанавливают имя для некоторого постоянного значения. Объявляем символическую константу, используя ключевое слово const, и применяем следующий синтаксис для создания константы:

```
const тип идентификатор = значение;
```

Константа обязательно должна быть проинициализирована, и ее значение не может быть изменено во время выполнения программы. Например:

```
const double pi = 3.14159265358979323846264338327 95;
pi = 5.0; //недопустимая операция
```

В этом примере число 3.14... – литеральная константа, а pi – символическая константа типа double. Ниже представлен пример использования символических констант.

```
class Constants
  {
     static void Main ()
     {
        const double pi = 3.14159265358979323846264338327 95;
        const double g = 9.81; //гравитационная постоянная
        System.Console.WriteLine("Число пи: {0}" ,pi);
        System.Console.WriteLine("Гравитационная постоянная: {0}",g);
     }
  }
```

Результат работы программы будет следующий:

```
Число пи: 3.14159265358979323846264338327 95
Гравитационная постоянная: 9.81
```

В последнем примере создаются две символические целочисленные константы: pi и g. Эти константы преследуют ту же цель, что и использование их литеральных значений. Но данные константы имеют имена, которые несут в себе гораздо больше информации об используемом значении, чем просто набор цифр.

Для того чтобы убедиться, что константные значения не могут быть изменены во время выполнения программы, добавим в код следующую строку: g = 10;

Во время компиляции получится следующее сообщение об ошибке:

error CS0131: The left-hand side of an assignment must be a variable, property or indexer.

Перечисления

Перечисления являются мощной альтернативой константам. Это особый тип значений, который состоит из набора именованных констант.

Допустим, есть список констант, содержащих годы рождения знакомых. Для того чтобы запрограммировать это при помощи констант, придется написать:

```
const int maryBirthday = 1955;
const int ivanBirthday = 1980;
const int pavelBirthday = 1976;
```

Получились три совершенно не связанные константы. Для того чтобы установить логическую связь между ними, в C# предусмотрен механизм перечислений. Вот как выглядит тот же код, записанный при помощи перечислений:

```
enum FriendsBirthday
{
const int maryBirthday = 1955;
const int ivanBirthday = 1980;
const int pavelBirthday = 1976;  }
```

Теперь три символические константы являются элементами одного перечисления типа FriendsBirthday. Каждое перечисление имеет свой базовый тип, которым может быть любой встроенный целочисленный тип C# (int, long, short и т. д.), за исключением char.

Перечисление задается следующим образом:

[атрибуты] [модификаторы] enum идентификатор[: базовый тип] (список перечислений);

Атрибуты и модификаторы будут рассматриваться далее. Пока остановимся на второй части этого объявления. Перечисление начи-

нается с ключевого слова enum, которое сопровождается идентификатором типа:

```
enum MyEnurnerators
```

Базовый – это основной тип для перечисления. Если не учитываеть этот описатель при создании перечисления, то будут использоваться значения по умолчанию int. Но можно применить любой из целочисленных типов (например, ushort, long), за исключением char.

Например, следующий фрагмент кода объявляет перечисление целых чисел без знака (uint):

```
enum Sizes: uint
{
Small = 1,
Middle = 2,
Large = 3
}
```

Внутри каждого перечисления записывается список возможных значений перечисления, разделенных запятой. Каждое значение может представлять собой либо просто набор символических констант, либо набор символических констант в сочетании с литеральным целочисленным значением. Если не указать для элементов перечисления целочисленных значений, то компилятор пронумерует их сам, начиная с 0. Например, следующие фрагменты кода аналогичны:

```
enum Sizes: uint
{
Small,
Middle,
Large
}
```

Если объявить свое перечисление следующим образом:

```
enum Sizes: uint
{
Small,
Middle = 20,
Large
}
```

то элементы перечисления будут иметь такие числовые значения:

```
Small = 0;
Middle = 20;
Large = 21.
```

Рассмотрим пример, который наглядно показывает, каким образом перечисления помогают упростить код приложения:

```
class ScreenResolutions
{
//перечисление размеров мониторов в дюймах
   enum Screens
     {
       Small = 14,
       Medium = 17,
       Large = 19,
       SuperLarge = 21
     }
  static void Main()
    {
     System.Console.WriteLine("Самые маленькие мониторы имеют размер: {0}", (int) Screens.Small );
     System.Console.WriteLine("Самые большие мониторы имеют размер: {0}", (int) Screens.SuperLarge);
      }
  }
```

Как видно, значение перечисления (SuperLarge) должно быть специфицировано именем перечисления (Screens). По умолчанию, значение перечисления представляется его символическим именем (типа Small или Medium). Если хочется отобразить значение константы перечисления, то нужно привести константу к ее основному типу (в данном случае, int).

Целочисленное значение передается в функцию WriteLine и отображается на экране.

Строковые константы

Для объявления в программе константной строки необходимо заключить содержимое строки в двойные кавычки («My string»). Это можно делать практически в любом месте программы: в передаче параметров функции, в инициализации переменных. Уже неоднократно применялись строковые константы при выводе данных на экран.

System.Console.WriteLine(«Самые большие мониторы имеют размер: {C}», (int) Screens.SuperLarge);

Здесь в качестве одного из параметров функции используется строка «Самые большие мониторы имеют размер: {0}».

string strMessage = «Здравствуй Мир!»;

В данном случае константная строка «Здравствуй Мир!» инициализирует переменную strMessage.

Массивы

Рассмотрим массивы с точки зрения типов данных. Массивы в C# несколько отличаются от других C-подобных языков.

Начнем сразу с примеров. Пример первый:

```
int [ ] к;  //к - массив
k=new int [3];  //определяем массив из трех целых
к[0]=-5;  к[1]-4;  к[2]=55;  //Задаем элементы массива
//Выводим третий элемент массива
Console.WriteLine(к[2] .ToString ());
```

Смысл приведенного фрагмента ясен из комментариев. Стоит обратить внимание на некоторые особенности. Во-первых, массив определяется именно как

```
int[] к;
```

а не как один из следующих вариантов:

```
int k[];
 //Неверно!
int k [3];
 //Неверно !
int [3] к;
 //Неверно!
```

Во-вторых, так как массив представляет собой ссылочный объект, то для создания массива необходима строка

```
k=new int [3];
```

Именно в ней и определяется размер массива. Кроме того, возможны конструкции вида

```
int [] k = new int [3];
```

Элементы массива можно задавать сразу при объявлении. Например:

```
int [] к = {-5, 4, 55} ;
```

Разумеется, приведенные конструкции применимы не только к типу int, и не только к массиву размера 3.

В C#, как и в C/C++, нумерация элементов массива идет с нуля. Таким образом, в данном примере начальный элемент массива – это k [0], а последний – к [2]. Элемента к [3], естественно, нет.

Теперь переходим к многомерным массивам. Вот так задается двумерный массив:

```
int [] k = new int [2, 3];
```

Необходимо обратить внимание на то, что пара квадратных скобок только одна. В примере у массива 6 (=2*3) элементов (к [0,0] – первый, к [1,2] – последний).

Аналогично можно задавать многомерные массивы. Вот пример трехмерного массива:

```
int [] к = new int [10,10,10];
```

А вот так можно сразу инициализировать многомерные массивы:

```
int[,] k = {{ 2, -2 }, { 3, -22 }, { 0, 4 }};
```

Приведенные выше примеры многомерных массивов называются прямоугольными. Если их представить в виде таблицы (в двумерном случае), то массив будет представлять собой прямоугольник.

Наряду с прямоугольными массивами существуют так называемые ступенчатые. Вот пример:

```
//Объявляем 2-мерный ступенчатый массив
int[][] k = new int [2][];
//Объявляем 0-й элемент нашего ступенчатого массива
//Это опять массив и в нем 3 элемента
k[0]=new int[3];
//Объявляем 1-й элемент нашего ступенчатого массива
//Это опять массив и в нем 8 элементов
k[1]=new int[8];
k[1][7]=100; //записываем 100 в последний элемент массива
```

Стоит обратить внимание на то, что у ступенчатых массивов задается несколько пар квадратных скобок (по размерности массива). И точно так же что-нибудь делается с элементами массива – запись, чтение и т. п.

Самая важная и интересная возможность ступенчатых массивов – это их «непрямоугольность». Так, в приведенном выше примере в первой «строке» массива k три целых числа, а во второй – восемь. Часто это оказывается очень кстати.

Выражения, инструкции и разделители

Выражения (Expressions)

Выражение – это строка кода, которая определяет значение. Пример простого выражения:

```
myValue = 100;
```

Данная инструкция выполняет присвоение значения 100 переменной myValue. Оператор присвоения (=) не сравнивает стоящее справа от него значение (100) и значение переменной, которая находится слева от оператора (myValue). Оператор «=» устанавливает значение переменной myValue равным 100.

Поскольку myValue = 100, то как выражение, которое определяет значение 100, myValue может использоваться другим оператором присвоения.

Например:

```
mySecondValue = myValue = 100;
```

В данном выражении литеральное значение 100 присваивается переменной myValue, а затем оператором присвоения устанавливается вторая переменная mySecondValue с тем же значением 100. Таким образом, значение 100 будет присвоено обеим переменным одновременно. Инструкцией такого вида можно инициализировать любое число переменных с одним и тем же значением, например, 20:

```
a = b = c = d = e = 20;
```

Инструкции (Statements)

Инструкция – это законченное выражение в коде программы. Программа на языке C# состоит из последовательностей инструкций. Каждая инструкция обязательно должна заканчиваться точкой с запятой (;). Например:

```
int x;  // инструкция
```

```
x = 100;  //другая инструкция
int y = x;  //тоже инструкция
```

Кроме того, в C# существуют составные инструкции. Они, в свою очередь, состоят из набора простых инструкций, помещенных в фигурные скобки { }.

```
{
int x;  // инструкция
x = 100;  //другая инструкция
int y = x;  //тоже инструкция
```

В этом примере все три инструкции являются элементами одной инструкции.

C# инструкции рассматриваются в соответствии с порядком их записи в тексте программы. Компилятор начинает рассматривать код программы с первой строки и заканчивает концом файла.

Разделители (Delemiters)

В языке C# пробелы, знаки табуляции и переход на новую строку рассматриваются как разделители. В инструкциях языка C# лишние разделители игнорируются. Таким образом, можно написать:

```
myValue = 100;
```

или:

```
myValue  =  100;
```

Компилятор обработает эти две инструкции как абсолютно идентичные. Исключение состоит в том, что пробелы в пределах строки не игнорируются. Если написать:

```
Console.WriteLine(«Я изучаю C# !»);
```

каждый пробел между словами «Я», «изучаю», «C#» и знаком «!» будет обрабатываться как отдельный символ строки.

В большинстве случаев использование пробелов происходит чисто интуитивно. Обычно они применяются для того, чтобы сделать программу более читаемой для программиста; для компилятора разделители абсолютно безразличны.

Надо заметить, что есть случаи, в которых использование пробелов является весьма существенным. Например, выражение:

```
int myVaiue = 25;
```

то же самое, что и выражение:

```
int myValue=25;
```

но следующее выражение не будет соответствовать двум предыдущим:

```
intmyValue =25;
```

Компилятор знает, что пробел с обеих сторон оператора присвоения игнорируется (сколько бы много их не было), но пробел между объявлением типа int и именем переменной myVaiue должен быть обязательно.

Это не удивительно, пробелы в тексте программы позволяют компилятору находить и анализировать ключевые слова языка. В данном случае это int, а некоторый термин intmyValue для компилятора неизвестен. Можно свободно добавлять столько пробелов, сколько нравится, но между int и myVaiue должен быть, по крайней мере, один символ пробела или табуляции.

Ветвление программ

Для того чтобы программы на C# были более гибкими, используются операторы перехода (операторы ветвления). В C# есть два типа ветвления программы: безусловный и условный переходы.

Кроме ветвлений, в C# также предусмотрены возможности циклической обработки данных, которые определяются ключевыми словами: for, while, do, in и foreach. Пока рассмотрим несколько способов условного и безусловного переходов.

Безусловные переходы

Безусловный переход осуществляется двумя способами.

Первый способ – это вызов функций. Когда компилятор находит в основном тексте программы имя функции, то происходит приостановка выполнения текущего кода программы и осуществляется пере-

ход к найденной функции. Когда функция выполнится и завершит свою работу, то произойдет возврат в основной код программы, на ту инструкцию, которая следует за именем функции. Следующий пример иллюстрирует безусловный переход с использованием функции:

```
using System;
class Functions
{
   static void Main()
   {
      Console.WriteLine(«Метод Main. Вызываем метод Jump...»);
      Jump();
      Console.WriteLine(«Возврат в метод Main.»);
   }
   static void Jump()
   {
      Console.WriteLine(«Работает метод Jump!»);
   }
}
```

Если откомпилировать и запустить программу, то результатом ее работы будет:

```
Метод Main. Вызываем метод Jump...
Работает метод Jump!
      Возврат в метод Main.
```

Программа начинает выполняться с метода Main() и осуществляется последовательно, пока компилятор не вызовет функцию Jump(). Таким образом происходит ответвление от основного потока выполняемых инструкций. Когда функция Jump() заканчивает свою работу, то продолжается выполнение программы со следующей строки после вызова функции.

Второй способ реализации безусловного перехода можно осуществить при помощи ключевых слов: goto, break, continue, return или throw. Дополнительная информация об инструкциях goto, break, continue и return будет дана чуть ниже.

Условные переходы

Условный переход можно реализовать в программе с помощью ключевых слов языка: if, else или switch. Такой переход возможен только при условии его истинности.

if...else оператор

if...else – это оператор ветвления, работа которого определяется условием. Условие оператора анализируется инструкцией if. Если условие верно (true), то выполняется блок инструкций программы, описанных после условия.

```
if ( expression ) statement 1
 [else statement2 ]
```

Такой вид оператора можно найти в документации по C#. Он показывает, что работа условного оператора определяется булевым выражением (выражение, которое имеет значение true или false) в круглых скобках. Если значение этого выражения истинно, то выполняется блок инструкций statement1. Если же выражение ложно, произойдет выполнение блока инструкций statement2. Необходимо заметить, что вторая часть оператора (else statement2) может не указываться. Если инструкций в блоках statement1 или statement2 больше одной, то блок обязательно нужно брать в фигурные скобки.

```
using System;

class Conditional
{
  static void Main()
  {
     int valueOne = 50;
     //устанавливаем второе значение больше первого
     int valueTwo = 5;
     if ( valueOne > valueTwo )
        Console.WriteLine ("valueOne: {0} больше чем valueTwo: {1}", valueOne, valueTwo);
     else
        Console.WriteLine("valueTwo: {0} больше или равно valueOne: {1}", valueTwo, valueOne);
     //устанавливаем первое значение больше второго
     valueTwo = 50;
     if ( valueOne > valueTwo )
     {
        Console.WriteLine("valueOne: {0} больше чем valueTwo: {1}", valueOne, valueTwo);
     }
     //делаем значения одинаковыми
      valueOne = valueTwo;
      if (valueOne == valueTwo)
      {
         Console.WriteLine ( "valueOne и valueTwo равны: {0}=={1}", valueOne, valueTwo);
      }
   }
}
```

Операторы сравнения: больше чем (>), меньше чем (<) или равно (==) достаточно понятны и просты в использовании.

В этом примере первый оператор if проверяет значение переменной valueOne относительно значения переменной valueTwo. Если значение valueOne больше значения переменной valueTwo (valueOne будет равным 50-ти, а valueTwo будет равным пяти), то условие оператора (valueOne > valueTwo) является истинным и на экране появится строка:

valueOne: 50 больше, чем valueTwo: 5

Во втором операторе if условие осталось прежним, но значение valueTwo уже изменилось на 50. Поэтому условие оператора (valueOne > valueTwo) является ложным и результат не будет отображен на экране.

Последний оператор условия проверяет равенство двух значений (valueOne == valueTwo). А поскольку перед этим мы приравняли две переменные, то результат проверки будет истинным. И на экране появится информация:

valueOne и valueTwo равны: 50==50

Результатом работы программы будет следующая информация:

valueTwo: 50 больше или равно valueOne: 5

valueOne и valueTwo равны: 50==50

Вложенные операторы условия

Для обработки сложных условий возможно вложение условных операторов в блоки инструкций других условных операторов. Например, необходимо оценить рабочую температуру воздуха в офисе и выдать сообщение пользователю.

Если значение температуры больше 21° и меньше 26°, то температура находится в пределах нормы. Если же температура равна 24°, то выдается сообщение о том, что рабочий климат оптимален.

Есть много способов написать такую программу. Приведем пример, иллюстрирующий один из способов, в котором используется вложенность оператора if...else:

```
using System;

class Values
{
   static void Main( )
```

```
    {
        int temp = 25;
        if (temp > 21)
        {
            if (temp < 26)
            {
                Console.WriteLine ("Температура {0} находится в пределах нормы", temp);
                if (temp == 24)
                {
                    Console.WriteLine("Рабочий климат оптимален");
                }
                else
                {
                    Console .WriteLine ("Рабочий климат не оптимален\n" + "Оптимальная температура 24");
                }
            }
        }
        Console.ReadLine();
    }
}
```

В теле программы встречается два условных оператора if...else. В первом операторе происходит проверка на попадание значения температуры в нижний предел (21°). Значение temp больше, чем 21, значит, условие (temp > 21) истинное и выполнится следующая проверка. Во втором операторе происходит проверка на попадание значения температуры в верхний предел (26°). Значение temp равно 24, значит, условие (temp < 26) истинное и будет выполняться блок инструкций в фигурных скобках.

Таким образом, уже выяснилось, что температура в пределах нормы, и остается узнать, является ли она оптимальной – это и реализует последний оператор if. Значение temp равно 25°, значит, условие (temp == 24) последнего оператора ложно и на экране появится сообщение:

Температура 25 находится в пределах нормы.
Рабочий климат не оптимален. Оптимальная температура 24.

В условном операторе в круглых скобках стоит два знака равно: if(tempValue == 24) Если бы тут стоял один знак, то такую ошибку трудно заметить. И результатом такого выражения стало бы присвоение переменной temp значения 24. В C и C ++ любое значение, отличное от нуля, определяется как булева истина (true), следовательно, условный оператор вернул бы истину, и на экране вывелась

бы строка «Рабочий климат не оптимален». Таким образом, действие выполнилось бы неправильно, а побочным эффектом стало бы нежелательное изменение значения temp.

C# требует, чтобы условные операторы принимали в качестве условий только булевы значения. Поэтому если бы такая ошибка возникла, то компилятор не смог бы преобразовать выражение temp = 24 к булеву типу и выдал бы сообщение об ошибке в процессе компиляции программы. Это является достоинством по сравнению с C++, так как разрешается проблема неявных преобразований типов данных, в частности, целых чисел и булева типа.

Использование составных инструкций сравнения

Оператор if в инструкции сравнения может применять несколько инструкций, объединенных арифметическими операторами. В качестве последних используются операторы (&& – И), (|| –ИЛИ) и (! – НЕ).

Рассмотрим пример использования составных инструкций в блоке if:

```
using System;

class Conditions
{
   static void Main(string[] args)
   {
      int n1 = 5;
      int n2 = 0;
      if ((n1 == 5) && (n2 == 5))
         Console.WriteLine("Инструкция 'И' верна");
      else
         Console.WriteLine("Инструкция 'И' неверна");
      if ((n1 == 5) || (n2 == 5))
         Console.WriteLine("Инструкция 'ИЛИ' верна");
      else
         Console.WriteLine("Инструкция 'ИЛИ' неверна");
      Console.Read();
   }
}
```

В данном примере каждая инструкция if проверяет сразу два условия. Первое if условие использует оператор (&&) для проверки условия. Необходимо, чтобы сразу два условия в блоке if были истинны. Только тогда выражение будет считаться верным:

if ((n1==5) &&(n2==5))

При этом не всегда выполняются все выражения в блоке if. Если первое выражение однозначно определяет результат операции, то второе уже не проверяется. Так, если бы условие n1 == 5 было ложным, то условие n2==5 уже не проверялось бы, поскольку его результат перестал бы играть роль. Второе выражение использует оператор (||) для проверки сложного условия: if ((n1==5)||(n2==5)).

В данном случае для верности всего выражения достаточно истинности лишь одного из условий. И тогда действует правило проверки условий до выяснения однозначности результата. То есть если условие n1 == 5 верно, то n2 == 5 уже проверяться не будет. Это свойство может сыграть очень важную роль при разработке кода программ. Рассмотрим пример:

```
using System;

class Conditions
{
   static void Main(string[] args)
   {
      const int MAX_VALUE = 3;
      int n1 = 1;
      int n2 = 1;
      if((++n1 < MAX_VALUE) && (++n2 < MAX_VALUE))
         Console.WriteLine("Операция && nl:={0} n2:={1}", n1, n2);
      if((++n1 < MAX_VALUE + 1) || (++n2 < MAX_VALUE + 1))
         Console.WriteLine("Операция || nl:={0} n2:={1}; ", n1, n2);
   }
}
```

Результат работы программы будет следующий:

```
Операция && n1:=2 n2:=2
Операция || n1:=3 n2:=2
```

Заметим, что первый оператор if нарастил значения обеих переменных, потому как осуществлялась проверка обоих условий. Второй оператор if выполнил инкремент только переменной n1.

Оператор switch как альтернатива оператору условия

Достаточно часто встречаются ситуации, когда вложенные условные операторы выполняют множество проверок на совпадение значения переменной, но среди этих условных операторов только один является истинным.

```
        if (myValue == 10) Console.WriteLine("myValue равно 10");
    else
        if (myValue == 20) Console.WriteLine("myValue равно 20 " );
    else
        if (myValue == 30} Console.WriteLine("myValue равно 30 " );
else ....
```

Когда имеется такой сложный набор условий, лучше всего воспользоваться оператором switch, который является более удобной альтернативой оператору if.

Логика оператора switch следующая: «найти значение, соответствующее переменной для сравнения, и выполнить соответствующее действие». Иными словами, он работает как оператор выбора нужного действия.

```
switch (выражение)
{
case константное выражение: инструкция
выражение перехода
[default:: инструкция]
```

Видно, что, подобно оператору условия if...else, выражение условия помещено в круглые скобки в начале оператора switch.

Внутри оператора switch есть секция выбора – case и секция действия по умолчанию – default. Секция выбора нужна для определения действия, которое будет выполняться при совпадении соответствующего константного выражения и выражения в switch. В этой секции обязательно нужно указать одно или несколько действий. Секция default может в операторе switch не указываться. Она выполняется в том случае, если не совпала ни одна константная инструкция из секции выбора.

Оператор case требует обязательного указания значения для сравнения (constant-expression) – константного выражения (литеральная, или символическая константа, или перечисление), а также блока инструкций (statement) и оператора прерывания действия (jump-statement).

Если результат условия совпадет с константным значением оператора case, то будет выполняться соответствующий ему блок инструкций. Как правило, в качестве оператора перехода используют опе-

ратор break, который прерывает выполнение оператора switch. Альтернативой ему может быть и другой оператор – goto, который обычно применяют для перехода в другое место программы.

Чтобы можно было увидеть, как оператор switch заменяет сложный набор условий, приведем пример той же программы, но с использованием оператора switch:

```
switch ( myValue )
{
   case 100:
      Console.WriteLine(«Переменная myValue равна 100»);
   break;
   case 200:
      Console.WriteLine(«Переменная myValue равна 200»);
   break;
      case 300: Console.WriteLine(«Переменная myValue равна 300»);
   break;
}
```

В этом примере проверяется значение переменной myValue на равенство ее одному из следующих значений: 100, 200, 300. Если, например, myValue будет равно 100, то на экран выведется строка:

myValue равно 100

В языке C и C++ можно автоматически перейти к секции следующего case, если в конце предыдущего не стоит инструкция перехода break или goto. Таким образом, на C++ можно написать:

```
case 1: statment1;
case 2: statment2;
break;
```

В этом примере на C++ после выполнения statement1 будет автоматически выполняться секция statment2.

В C# это работать не будет. Автоматический переход от case 1 к следующей секции case 2 будет выполняться только в том случае, если секция case 1 окажется пустой (не будет содержать ни одной инструкции). В противном же случае перехода к выполнению case 2 не произойдет, так как в C# каждая непустая секция инструкций оператора case должна содержать в себе оператор break.

```
case 1: Console.WriteLine(«Выражение секции 1»);
case 2:
```

Здесь case 1 содержит инструкцию, поэтому нельзя автоматически перейти к выполнению case 2. Такой код вообще не будет компилироваться. Если после выполнения case 1 нужно перейти к выполнению case 2, то необходимо явно указать это при помощи оператора goto:

```
case 1: Console.WriteLine(«Выражение секции 1»);
goto case 2;
case 2:
```

Однако другая форма использования оператора switch позволит обойтись без инструкции goto:

```
case 1:
case 2:
Console.WriteLine("Выражение секций 1 и 2»);
```

Такой принцип работы оператора switch используется в случае, когда необходимо выполнить одно и то же действие (или часть действия) для разных значений условного оператора.

```
        using System;
namespace SwitchStatement
{
   class MyClass
   {
      static void Main(string[] args)
      {
         int user = 1;
         user = Convert.ToInt32(Console.ReadLine());
         switch (user)
                {
            case 1:
               Console.WriteLine(«Здравствуйте, Userl»);
            break;
            case 2:
               Console.WriteLine(«Здравствуйте, User2»);
            break;
```

```
            case 3:
               Console.WriteLine(«Здравствуйте, User3»);
            break;
            default:
               Console.WriteLine(«Здравствуйте, новый пользователь»);
             break;
      }
    }  }
}
```

Этот пример выводит на экран сообщение с приветствием пользователя в зависимости от введенного на экране значения. Так, если ввести число 1, то на экране появится сообщение «Здравствуйте, User2».

В данном примере для выбора выражения, выводящего сообщение на экран, использовалось числовое значение {0, 1 или 2}. Если же ввести иное значение, нежели представленное в массиве, то на экране появится сообщение «Здравствуйте, новый пользователь».

Объявление переменных внутри case инструкций

Рассмотрим случай, когда необходимо создать в программе сложную case инструкцию. Для придания программе большей читабельности создание переменных, необходимых для использования, лучше всего объявлять непосредственно перед их применением. Так, если каждый блок case использует свой набор переменных, то и объявление переменных следует делать внутри блоков case. Посмотрим на следующий код:

```
        using System;
class Part
{
   public static void Main()
   {
      Console.WriteLine("Выбрать:");
      Console.WriteLine("1: для ввода наименования товара\n2: для ввода количества товара");

      int Choice = Convert.ToInt32(Console.ReadLine());
      switch (Choice)
      {
         case 1:
            string Name;
            Console.Write("Введите наименование товара ");
            Name = Console.ReadLine();
```

```
            break;
        case 2:
            int Count;
            Console.Write("Введите количество товара ");
            Name = Console.ReadLine();
            Count = Convert.ToInt32(Console.ReadLine());
            break;
        default:
        break;
    }
  }
}
```

На C++ такая switch инструкция компилироваться не будет. Причина этому – объявление переменных внутри блоков case. Однако разработчики C# постарались и предусмотрели возможность создания переменных внутри case блоков. Поэтому на C# такой код является рабочим.

Switch и работа со строками

В приведенном ранее примере переменная user была целочисленного типа. Если необходимо использовать в качестве условия оператора switch переменную строкового типа, то можно сделать это следующим образом: case «Валентина»: Если строк для сравнения много, то по аналогии с целочисленной переменной user используется несколько инструкций case.

Ниже дан пример использования строковой переменной в качестве условия оператора switch:

```
using System;
class MyClass
{
    static void Main(string[] args)
    {
        string user;
        Console.WriteLine(«Возможные варианты:»);
        Console.WriteLine(«User1»);
        Console.WriteLine(«User2»);
        Console.WriteLine(«User3»);
        user = Console.ReadLine();
        switch (user)
        {
            case "User1":
```

```
            Console.WriteLine(«Здравствуйте, пользователь один»);
            break;
        case "User2":
            Console.WriteLine(«Здравствуйте, пользователь два»);
            break;
        case "User3":
            Console.WriteLine(«Здравствуйте, пользователь три»);
            break;
        default:
            Console.WriteLine(«Здравствуйте, новый пользователь»);
             break;
    }   }
}
```

В данном случае для идентификации пользователя необходимо применить не числовое значение, а строку. Если ввести строку «User1», то на экране появится сообщение «Здравствуйте, пользователь один».

Циклические операторы

C# включает достаточно большой набор циклических операторов, таких как for, while, do...while, а также цикл с перебором каждого элемента foreach. Кроме того, C# поддерживает операторы перехода и возврата, например, goto, break, continue и return.

Оператор goto был основой для реализации других операторов цикла. Но он был и базой многократных переходов, вследствие чего возникла запутанность кода программы. Поэтому опытные программисты стараются его не использовать, но для того чтобы узнать все возможности языка, рассмотрим и этот оператор.

Он используется следующим образом:

1. Создается метка в коде программы Label 1.
2. Организуется переход на эту метку goto Labe1.

Имя метки Labe1 обязательно должно заканчиваться двоеточием. Оно указывает на точку в программе, с которой будет выполняться программа после использования инструкции goto. Обычно инструкция goto привязывается к условию, как показано в следующем примере:

```
Using System;
public class Labels
{
   public static int Main()
```

```
    {
        int i = 0;
    Label:
        Console.WriteLine("i: {0 } ", i);
         i = i + 1 ;
        if (i < 10)
            goto Label;
        return 0;
    }
}
```

Здесь на экран выводится строка со значением i десять раз (от 0 до 9). Инструкция goto помогает повторить выполнение одних и тех же инструкций определенное число раз. В этой программе число повторов определяется инструкцией if(i < 10). Следовательно, до тех пор, пока переменная I будет иметь значение меньшее, чем 10, goto будет переходить на метку label:, а, значит, вывод строки на экран будет повторяться. То есть с использованием goto можно организовать циклический повтор операций в программе.

Именно это явление привело к созданию альтернативного метода организации циклов, такого как while, do..while, for или foreach. Большинство программистов понимают, что использование goto в программе лучше заменять чем-нибудь другим, что приведет к созданию более структурированного и понятного программного кода.

Цикл while работает по принципу: «Пока выполняется условие – происходит работа». Ее синтаксис выглядит следующим образом:

```
while (выражение)
{
   инструкция;
}
```

Как и в других инструкциях, выражение – это условие, которое оценивается как булево значение. Если результатом проверки условия является истина, то выполняется блок инструкций, в противном случае в результате выполнения программы while игнорируется. Рассмотрим пример, приведенный выше, только с использованием while:

```
    using System;
public class WhileCycle
{
```

```
    public static int Main()
    {
        int i = 0;
        while (i < 10)
        {
            Console.WriteLine("i: {0}", i);
            ++i;
        }
        return 0;
    }
}
```

По своей функциональности и та, и другая реализация программы работают абсолютно одинаково, но логика работы несколько изменилась. Заметим, что цикл while проверяет значение i перед выполнением блока statement. Это гарантирует, что цикл не будет выполняться, если проверяемое условие ложно. Таким образом, если первоначально i примет значение 10 и более, цикл не выполнится ни разу. Инструкция while является вполне самостоятельной, а в данном примере ее можно прочитать подобно предложению: «пока i меньше 10, выводим сообщение на экран и увеличиваем i».

Цикл do... while. Бывают случаи, когда цикл while не совсем удовлетворяет требованиям. Например, нужно проверить условие не в начале, а в конце цикла. В таком случае лучше использовать цикл do...while:

```
do
{
Инструкция
}
while (выражение);
```

Подобно while, выражение – это условие, которое оценивается как булево значение.

Это выражение можно прочитать, как: «выполнить действие; если выполняется условие – повторить выполнение еще раз». Заметим разницу между этой формулировкой и формулировкой работы цикла

while, которая состоит в том, что цикл do...while выполняется всегда минимум один раз, до того как произойдет проверка условия выражения:

```
using System;
public class DoWhile
{
   public static int Main()
   {
      int i = 0;
      do
      {
         Console.WriteLine("i : {0}", i);

         ++i;
      }
      while (i < 10);
      return 0;
   }
}
```

На этом примере видно, что если первоначально i примет значение 10 и более, цикл выполнится. Затем произойдет проверка условия while (i < 10), результатом которой станет ложь (false), и повтора выполнения цикла не произойдет. То есть он выполнится один раз. Как уже было сказано, при таких же начальных условиях while не выполнился ни разу.

Цикл for

Если еще раз внимательно посмотреть на примеры (while, do...while, goto), можно заметить постоянно повторяющиеся операции: первоначальная инициализация переменной i, ее наращивание на 1 внутри цикла, проверка переменной i на выполнение условия (i < 10). Цикл for позволяет объединить все операции в одной инструкции.

```
    for ( [инициализация ];  [ выражение];  [ наращивание] )
{  инструкция
      }
```

Выполним тот же пример, но уже с использованием цикла for:

```
using System;

public class ForCycle
{
    public static int Main()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine("i: {0}", i);
        }
        return 0;
    }
}
```

Результатом выполнения такого цикла будет вывод на экран информации вида:

```
0
1
2
...
9
```

Принцип работы такой инструкции очень прост:

1. Происходит инициализация переменной i.

2. Выполняется проверка соответствия условию. Если условие истинно, то происходит выполнение блока вложенных инструкций; если условие оказалось ложным, то цикл прекращается и выполняется программа за фигурными скобками.

3. Переменная i увеличивается на 1. Наращивание переменной внутри цикла происходит на такое число единиц, на которое зададается. Операция i++ означает «увеличить значение переменной на 1». Если нужно использовать другой шаг изменения i, то смело можно написать так: i += 2. В этом случае значение переменной i будет изменяться на 2 единицы, и на экране можно увидеть:

```
0
2
4
```

6
8

Цикл foreach. Эта инструкция не знакома программистам на языке C, она используется для перебора массивов и объединений (collection) по элементам.

break и continue. Бывают ситуации, когда необходимо прекратить выполнение цикла досрочно (до того, как перестанет выполняться условие) или при каком-то условии не выполнять описанные в теле цикла инструкции, не прерывая при этом цикла. Для таких случаев очень удобно использовать инструкции break и continue. Если нужно на каком-то шаге цикла его прекратить, не обязательно выполняя до конца описанные в нем действия, то лучше всего использовать break. Следующий пример хорошо иллюстрирует такую ситуацию:

```
using System;

class Values
{
   static void Main()
   {
      //объявляем флаг для обозначения простых чисел
      bool IsPrimeNumber;
      for(int i=100;  i > 1;  i--)
      {
         //устанавливаем флаг
         IsPrimeNumber = true;
         for (int j = i-1; j > 1; j--)
         {
            //если существует делитель с нулевым остатком
            if(i%j == 0)
            {
               //сбрасываем флаг IsPrimeNumber = false;
               // если не нашлось ни одного делителя
               //с нулевым остатком, то число простое
               if(IsPrimeNumber == true)
               Console.WriteLine("{0}– простое число", i);
            }
         }
      }
   }
```

Программа выполняет поиск всех простых чисел от 2 до 100. В программе используется два цикла for. Первый цикл перебирает все числа от 100 до 2. Именно от 100 до 2, а не наоборот. Переменная i инициализируется значением 100 и затем уменьшается на 1 с каждой итерацией. Второй цикл перебирает все числа от i до 2. Таким образом, второй цикл будет повторяться 99 + 98 + 97 + ... + 3 + 2 раз. То есть первый раз он выполнится 99 раз, второй – 98 и т. д. В теле второго цикла проверяется выполнение условия: делится ли число i на число j без остатка (i%j == 0). Если это условие верно, то число i нельзя отнести к разряду простых. Следовательно, флажок, определяющий число как простое, устанавливается в false. По окончании работы вложенного цикла проверяется условие – не установился ли флажок, определяющий число как простое, в false. Если нет, то число является простым, и на экран выводится соответствующее сообщение.

В данной программе происходит выполнение всех описанных действий внутри цикла. А что если программа уже отнесла число к разряду непростых чисел? Зачем в этом случае продолжать проверку на существование нулевого делителя? В этом нет необходимости. Это лишь дополнительная загрузка ресурсов программы. Для того чтобы прервать выполнение вложенного цикла, можно воспользоваться инструкцией break. Для этого необходимо изменить код второго цикла так, как показано ниже:

```
for(int j = i - 1; j > 1; j--)
{
   //если существует делитель с нулевым остатком
   if (i%j == 0)
   {
      //сбрасываем флаг IsPrimeNunber = false;
      // дальнейшая проверка бессмысленна
      break;
   }
}
```

Как только сбросится флаг IsPrimeNumber, вложенный цикл сразу же прервется и выйдет в основной цикл. Таким образом количество итераций сократится многократно, что благоприятно скажется на производительности работы программы.

Оператор continue, в отличие от break, не прерывает хода выполнения цикла. Он лишь приостанавливает текущую итерацию и сразу переходит к проверке условия выполнения цикла:

```
    for (int j = 0; j < 100; j++ )
{
   if (j%2 == 0)
   continue;
   Console.WriteLine("{0}",   j);
}
```

Такой цикл позволит вывести на экран все нечетные числа. Работает он следующим образом: перебирает все числа от 0 до 100. Если очередное число – четное, то все дальнейшие операции в цикле прекращаются, наращивается число j, и цикл начинается сначала.

Вечные циклы

При написании приложений с использованием циклов следует остерегаться зацикливания программы. Зацикливание – это ситуация, при которой условие выполнения цикла всегда истинно и выход из цикла невозможен. Рассмотрим простой пример:

```
using System;

class Cycles
{
   public static void Main()
   {
      int nl, r2;
      r1 = 0;
      r2 = n1 + 1 ;
      while(nl < n2)
      {
         Console.WriteLine("nl = {0} , n2 = {1} " , n1 , n2 );
      }
   }
}
```

Здесь условие (n1 < n2) всегда истинно. Поэтому выход из цикла невозможен. Следовательно, программа войдет в режим вечного цикла. Такие ошибки являются критическими, поэтому следует очень

внимательно проверять условия выхода из цикла. Однако иногда бывает полезно задать в цикле заведомо истинное условие. Типичным примером вечного цикла является следующая запись:

```
while(true)
{...}
```

Возможно, что такая конструкция приведет к зависанию системы, если не задать в теле цикла инструкцию его прерывания. Рассмотрим пример программы:

```
using System;
class Cycles
{
    public static void Main()
    {
        String Name;
        while (true)
        {
            Console.Write(«Введите Ваше имя»);
            Name = Console.ReadLine();
            Console.WriteLine(«Здравствуйте {0}», Name);
        }
    }
}
```

Такая программа не имеет выхода. Что бы не ввел пользователь, программа выдаст строку приветствия и запросит ввод имени заново. Однако все изменится, если в программу добавить условие, при выполнении которого цикл прерывается.

```
using System;
class Cycles
{
    public static void Main()
    {
        string Name;
        while (true)
        {
            Console.WriteLine(«Введите Ваше имя»);
            Name = Console.ReadLine();
            if (Name == "")
                break;
            Console.WriteLine(«Здравствуйте {0 )», Name);
        }
    }
}
```

На этот раз, как только пользователь нажмет клавишу «Enter» без ввода строки данных, сработает инструкция break, и программа выйдет из цикла.

Создание вечных циклов оправдывает себя, если существует несколько условий прерывания цикла и их сложно объединить в одно выражение, записываемое в блоке условия.

Вечный цикл можно создать не только при помощи оператора while.

Любой оператор цикла может быть использован для создания вечных циклов. Вот как выглядит та же программа, но с использованием цикла for:

```
using System;
class Cycles
{
    public static void Main()
    {
        string Name;
        for (; ; )
        {
            Console.Write(«Введите Ваше имя»);
            Name = Console.ReadLine();
            if (Name == "")
                break;
            Console.WriteLine(«Здравствуйте {0}», Name);
        }
    }
}
```

Цикл for может не содержать ни инструкции инициализации, ни инструкции проверки, ни инструкции итерации. Два оператора (;) внутри цикла for означают вечный цикл.

3.5. Элементы объектно-ориентированного подхода

Классы – это сердце каждого объектно-ориентированного языка. Класс представляет собой инкапсуляцию данных и методов для их обработки. Это справедливо для любого объектно-ориентированного языка. Языки различаются в этом плане лишь типами данных, хранимых в виде членов, а также возможностями классов. В том, что касается классов и многих функций языка, C# кое-что заимствует из C++ и Java и привносит новое, помогающее найти элегантные решения.

Определение классов

Синтаксис определения классов на C# прост, особенно если программирование происходит на C++ или Java. Поместив перед именем вашего класса ключевое слово class, нужно вставить члены класса, заключенные в фигурные скобки, например:

```
class MySimpleClass
{
  private long myClassID;

}
```

Этот простейший класс с именем MySimpleClass содержит единственный член – myClassID.

Назначение классов

Представим себе, что есть некоторый объект, который характеризуется рядом свойств. Например, работник на предприятии. У него есть такие свойства, как фамилия, возраст, стаж и т. п. Так вот, в этом случае удобно каждого работника описывать не рядом независимых переменных (строкового типа для фамилии, целого типа для возраста и стажа), а одной переменной типа Worker, внутри которой и содержатся переменные для фамилии, возраста и стажа. Здесь самое важное то, что в переменной типа Worker содержатся другие переменные. Конечно, типа Worker среди встроенных типов данных нет, но это не беда – можно ввести его.

Еще одна важная вещь: в классах, помимо переменных разных типов, содержатся функции (или, что то же самое, методы) для работы с этими переменными. Скажем, в примере с классом Worker логично ввести специальные функции для записи возраста и стажа. Функции будут, в частности, проверять правильность вводимой информации. Например, ясно, что возраст у работника не может быть отрицательным или большим 150. Так вот, функция и будет проверять правильность введенного пользователем возраста.

Рассмотрим первый пример класса. Создается новое консольное приложение для C# и вводится следующий текст:

```
using System;
namespace test
{
```

```
    //Начало класса
    class Worker
    {
       public int age = 0;
       public string name;
    }
    //Конец класса
    class Test
    {
       [STAThread]
       static void Main(string[] args)
       {
          Worker wrkl = new Worker();
          wrkl.age = 30;
          wrkl.name = «Петров Иван Сергеевич»;
          Console.WriteLine(wrkl.name + " - " + wrkl.age + " года");
       }
    }
}
```

Запускается программа, которая, как и следовало ожидать, выведет на экран: «Петров Иван Сергеевич – 30».

Разберем этот код:

```
class Worker
{
  public int age = 0;
  public string name;
}
```

Определяем класс Worker. Внутри этого класса существуют две переменные: целая age для возраста и name строкового типа для имени.

В отличие от C/C++, можно задавать некоторое начальное значение непосредственно сразу после объявления переменной:

public int age=0;

Начальное значение задавать вовсе не обязательно – это видно по переменной name.

Перед переменными пишется ключевое слово public. Значение у него такое же, как и в C++, что означает, что переменная (или функция) будет видна вне класса. Если не написать перед переменной ни-

какого модификатора доступа или указать private, то переменная не будет видна снаружи класса, и ее смогут использовать только функции этого же класса (т. е. она будет для «внутреннего использования»).

Далее в строчке Worker wrkl = new Worker(); заводится экземпляр класса в куче (heap) и возвращается ссылка на него.

Затем в строчках

```
wrkl.age = 30;
wrkl.name = "Петров Иван Сергеевич";
Console.WriteLine(wrkl.name + " - " + wrkl.age + " года");
```

используется класс, присваивая некоторые значения для возраста и имени и выводя их потом на экран.

Состав классов

Поле. Так называется член-переменная, содержащий некоторое значение. В ООП поля иногда называют данными объекта. К полю можно применять несколько модификаторов – в зависимости от того, как его собираются использовать. В число модификаторов входят static, readonly и const. Ниже познакомимся с их назначением и способами применения.

Метод. Это реальный код, воздействующий на данные объекта (или поля). Здесь нужно сосредоточиться на определении данных класса. Подробнее о методах расскажем далее.

Свойства. Иногда их называют «разумными» полями (smart fields), поскольку на самом деле они являются методами, которые клиенты класса воспринимают как поля. Это обеспечивает клиентам большую степень абстрагирования за счет того, что им не нужно знать, обращаются ли они к полю напрямую или через вызов метода-аксессора.

Константы. Как можно предположить, исходя из имени, константа – это поле, значение которого изменить нельзя.

Индексаторы. Если свойства – это «разумные» поля, то индексаторы – это «разумные» массивы, которые позволяют индексировать объекты методами-аксессорами get и set. С помощью индексатора легко проиндексировать объект для установки или получения значений.

События. Событие вызывает исполнение некоторого фрагмента кода. События – неотъемлемая часть программирования для Microsoft Windows. Например, события возникают при движении мыши, щелчке или изменении размеров окна.

Модификаторы доступа

Теперь, зная, что типы могут быть определены как члены класса C#, познакомимся с модификаторами, используемыми для задания степени доступа или доступности данного члена для кода, лежащего за пределами его собственного класса. Они называются модификаторами доступа (access modifiers) и приведены в табл. 3.2.

Таблица 3.2

Модификаторы доступа в C#

Модификатор доступа	Описание
public	Член доступен вне определения класса и иерархии производных классов
protected	Член невидим за пределами класса, к нему могут обращаться только производные классы
private	Член недоступен за пределами области видимости определяющего его класса. Поэтому доступа к этим членам нет даже у производных классов
internal	Член видим только в пределах текущей единицы компиляции. Модификатор доступа internal в плане ограничения доступа является гибридом public и protected, зависимым от местоположения кода

Если нужно оставить модификатор доступа для данного члена по умолчанию (private), то стоит явно задать для него модификатор доступа. Этим C# отличается от C++, где член, для которого явно не указан модификатор доступа, принимает на себя характеристики видимости, определяемые модификатором доступа, заданным для предыдущего члена. Например, в приведенном ниже коде на C++ видимость членов a, Ь и c определена модификатором public, а членов d и e – как protected:

```
class AccessCPlusPlus
{
  public:
  int a;
  int b;
  int c ;
  protected:
  int d;
  int e;
}
```

А в результате выполнения этого кода на C# член b объявляется как private. Для объявления членов на C# как public необходимо использовать следующую инструкцию:

```
class AccessCSharp
{
 public int a;
 public int b;
 public int c;
 protected int d;
 protected int e;
}
```

В результате выполнения следующего кода на C# член b объявляется как private:

```
public DifAccessInCSharp
{
  public int a;
  int b;
}
```

Метод Main

У каждого приложения на C# должен быть метод Main, определенный в одном из его классов. Кроме того, этот метод должен быть определен как public и static (ниже будет объяснено, что значит static). Для компилятора C# не важно, в каком из классов определен метод Main, а класс, выбранный для этого, не влияет на порядок компиляции. Здесь есть отличие от C++, где зависимости должны тщательно отслеживаться при сборке приложения. Компилятор C# достаточно «умен», чтобы самостоятельно просмотреть файлы исходного кода и отыскать метод Main. Между тем, этот очень важный метод является точкой входа во все приложения на C#.

Можно поместить метод Main в любой класс, но для его размещения рекомендуется создавать специальный класс. Это можно сделать, используя простой класс MyClass:

```
using System;
class MyClass
{
```

```
    private int MyClassID;
    class AppClass
      {
        static public void Main()
        {
          MyClass MyObj = new MyClass();
        }
      }
}
```

В данном примере два класса. Этот общий подход используется при программировании на C# даже простейших приложений. MyClass представляет собой класс предметной области, а AppClass содержит точку входа в приложение (Main). В этом случае метод Main создает экземпляр объекта MyClass, и, будь это настоящее приложение, оно использовало бы члены объекта MyClass.

Аргументы командной строки

Можно обращаться к аргументам командной строки приложения, объявив метод Main как принимающий аргументы типа массива строк. Затем аргументы могут обрабатываться так же, как любой массив. Ниже приводится простой код, который по очереди выводит все аргументы командной строки на стандартное устройство вывода.

```
using System;
class CommandLineApp
{
  public static void Main(string[] args)
  {
    foreach (string arg in args)
    {
      Console.WriteLine("Аргумент: {0}", arg);
    }
  }
}
```

А вот пример вызова этого приложения с парой случайно выбранных чисел:

```
C:\projects\CommandLine>CommandLineApp 100 200
Аргумент: 100
Аргумент: 200
```

Аргументы командной строки передаются в виде массива строк. Если это флаги или переключатели, их обработку можно запрограммировать самостоятельно.

Возвращаемые значения

Чаще всего метод Main определяется так:

```
using System;
class SomeClass
{
   public static void Main()
   {
   }
}
```

Метод Main можно определить так, чтобы он возвращал значения типа int. Хотя это не является общепринятым в приложениях с графическим интерфейсом, такой подход может быть полезным в консольных приложениях, предназначенных для пакетного исполнения. Оператор return завершает исполнение метода, а возвращаемое при этом значение применяется вызывающим приложением как код ошибки для вывода определенного пользователем сообщения об успехе или неудаче.

Для этого служит следующий прототип:

```
public static int Main()
{
   // вернуть некоторое значение типа int,
   // представляющее код завершения.
   return 0;
}
```

Несколько методов Main

В C# разработчиками включен механизм, позволяющий определять более одного класса с методом Main. Это нужно для того, чтобы поместить в классы тестовый код. Затем, используя переключатель /main; <имя_Класса>, компилятору C# можно задавать класс, метод Main которого должен быть задействован. Ниже приведен пример, в котором создано два класса, содержащих методы Main.

Скомпилировать это приложение так, чтобы в качестве точки входа в приложение применялся метод Main1.Main можно, используя переключатель:

```
csc MultipleMain.es /main:Main1
```

При изменении переключателя на /main:Main2 будет использован метод Main2.Main.

Следует соблюдать осторожность и задавать в указанном переключателе имени класса верный регистр символов, так как C# чувствителен к регистру. Кроме того, попытка компиляции приложения, состоящего из нескольких классов с определенными методами Main без указания переключателя /main, вызывает ошибку компилятора:

```
using System;
class Main1
{
   public static void Main()
   {
      Console.WriteLine("Main1");
   }
   class Main2
   {
      public static void Main()
      {
         Console.WriteLine("Main2");
      }
   }
}
```

Инициализация классов и конструкторы

Одно из величайших преимуществ языков ООП, таких как C#, состоит в том, что можно определять специальные методы, вызываемые всякий раз при создании экземпляра класса. Эти методы называются конструкторами (constructors). C# вводит в употребление новый тип конструкторов – статические (static constructors),.

Гарантия инициализации объекта должным образом, прежде чем он будет использован, – ключевая выгода от конструктора. Когда пользователь создает экземпляр объекта, вызывается его конструктор, который должен вернуть управление до того, как пользователь сможет выполнить над объектом другое действие. Именно это помогает обеспечивать целостность объекта и делать написание приложений на объектно-ориентированных языках гораздо надежнее.

Но как назвать конструктор, чтобы компилятор знал, что его надо вызывать при создании экземпляра объекта? Разработчики C# последовали в этом вопросе за разработчиками C++ и провозгласили, что

у конструкторов в C# должно быть то же имя, что и у самого класса. Вот простой класс с таким же простым конструктором:

```
using System;
class ConstructorApp
{
   ConstructorApp()
   {
      Console.WriteLine("Конструктор");
   }
   public static void Main()
   {
      ConstructorApp App = new ConstructorApp();
   }
}
```

Значений конструкторы не возвращают. Если использовать с конструктором в качестве префикса имя типа, то компилятор сообщит об ошибке, пояснив, что нельзя определять члены с теми же именами, что у включающего их типа.

Следует обратить внимание и на способ создания экземпляров объектов в C#. Это делается при помощи ключевого слова new:

<класс> <объект> = new <класс>(аргументы конструктора)

В C++ можно было создавать экземпляр объекта двумя способами: объявлять его в стеке, скажем, так:

```
//Код на Си- Создает экземпляр CMyCleas в стеке
CMyClass myClass;
```

или создавать копию объекта в свободной памяти (или в куче), используя ключевое слово C++ new:

```
//Код на C+-. Создает экземпляр CmyCiass в куче.
CMyClass myClass = new CmyClass();
```

Экземпляры объектов на C# формируются иначе, что и сбивает с толку новичков при разработке на C#. Причина путаницы в том, что для создания объектов оба языка используют одни и те же ключевые слова. Хотя с помощью ключевого слова new в C++ можно указать,

где именно будет создаваться объект, место его построения на C# зависит от типа объекта, экземпляр которого создается. Как уже известно, ссылочные типы создаются в куче, а размерные – в стеке. Поэтому ключевое слово new позволяет делать новые экземпляры класса, но не определяет место создания объекта.

Хотя можно сказать, что приведенный ниже код на C# не содержит ошибок, он делает совсем не то, что может подумать разработчик на C++:

MyClass myClass;

На C++ он создаст в стеке экземпляр MyClass. Как сказано выше, на C# можно создавать объекты, только используя ключевое слово new.

Поэтому на C# эта строка лишь объявляет переменную myClass как переменную типа MyClass, но не создает экземпляр объекта.

Примером служит следующая программа, при компиляции которой компилятор C# предупредит, что объявленная в приложении переменная ни разу не используется:

```
using System;
class ConstrctorApp
{
   ConstrctorApp()
   {
      Console.WriteLine("конструктор");
   }
   public static void Main()
   {
      ConstrctorApp App = new ConstrctorApp();
   }
}
```

Поэтому, объявляя объект, нужно создать где-нибудь в программе его экземпляр с помощью ключевого слова new:

```
Constructor App =new ConstructorApp() ;
```

Объекты объявляются перед использованием или созданием их экземпляров с помощью new, если объявляется один класс внутри другого. Такая вложенность классов называется включением (containment), или агрегированием (aggregation).

Статические члены класса

Можно определить член класса как статический (static member) или член экземпляра (instance member). По умолчанию, каждый член определен как член экземпляра. Это значит, что для каждого экземпляра класса делается своя копия этого члена. Когда член объявлен как статический, имеется лишь одна его копия. Статический член создается при загрузке содержащего класс приложения и существует в течение жизни приложения. Поэтому можно обращаться к члену, даже если экземпляр класса еще не создан.

Один из примеров – метод Main. CLR (Common Language Runtime) нужна универсальная точка входа в приложение. Поскольку CLR не должна создавать экземпляры объектов, существуют правила, требующие определить в одном из классов статический метод Main.

Можно захотеть использовать статические члены при наличии метода, который формально принадлежит классу, но не требует реального объекта. Скажем, если нужно отслеживать число экземпляров данного объекта, которое создается во время жизни приложения. Поскольку статические члены «живут» на протяжении существования всех экземпляров объекта, должен работать такой код:

```
using System;

class InstCount
{
   public InstCount()
   {
      instanceCount++;
   }
   static public int instanceCount;
   //instanceCount = 0;
   class AppClass
   {
      public static void Main()
      {
         Console.WriteLine(InstCount.instanceCount);
         InstCount ic1 = new InstCount();
         Console.WriteLine(InstCount.instanceCount);
         InstCount ic2 = new InstCount();
         Console.WriteLine(InstCount.instanceCount);
      }
```

```
    }
  }
```

В этом примере выходная информация будет следующая:

```
0
1
2
```

И последнее замечание по статическим членам: у них должно быть некоторое допустимое значение. Его можно задать при определении члена:

```
static public int instanceCount = 10;
```

Если переменная не инициализируется, это сделает CLR после запуска приложения, установив значение по умолчанию, равное 0. Поэтому следующие строки эквивалентны:

```
static public int instanceCount;
static public int instanceCount=0;
```

Константы и неизменяемые поля

Можно с уверенностью сказать, что возникнут ситуации, когда изменение некоторых полей при выполнении приложения будет нежелательно, например, это могут быть файлы данных, от которых зависит приложение, значение pi (π) для математического класса или любое другое используемое в приложении значение, о котором известно, что оно никогда не изменится. В этих ситуациях C# позволяет определять члены тесно связанных типов: константы и неизменяемые поля.

Константы

Константы (constants), представленные ключевым словом const, – это поля, остающиеся постоянными в течение всего времени жизни приложения. Определяя что-либо как const, достаточно помнить два правила. Во-первых, константа – это член, значение которого устанавливается в период компиляции программистом или компилятором (в последнем случае это значение по умолчанию). Во-вторых, значение члена-константы должно быть записано в виде литерала.

Чтобы определить поле как константу, нужно указать перед определяемым членом ключевое слово const:

```
using System;
class MagicNumbers
{
   public const double pi = 3.1415;
   public const int g = 10;
   class ConstApp
   {
      public static void Main()
      {
         Con-
sole.WriteLine("pi = {0}, g = {1}", MagicNumbers.pi, MagicNumbers.g);
      }
   }
}
```

Стоит обратить внимание на один важный момент, связанный с этим кодом. Клиенту нет нужды создавать экземпляр класса MagicNumbers, поскольку по умолчанию члены const являются статическими.

Неизменяемые поля

Поле, определенное как const, ясно указывает, что программист намерен поместить в него постоянное значение. Это плюс. Но оно работает, только если известно значение подобного поля в период компиляции. А что же делать программисту, когда возникает потребность в поле, чье значение не известно до периода выполнения, но после инициализации не должно меняться? Эта проблема (которая обычно остается нерешенной в большинстве других языков) разрешена разработчиками языка C# с помощью неизменяемого поля (readonly field).

Определяя поле с помощью ключевого слова readonly, можно установить значение поля лишь в одном месте – в конструкторе. После этого поле не могут изменить ни сам класс, ни его клиенты. Допустим, для графического приложения нужно отслеживать разрешение экрана. Справиться с этой проблемой с помощью const нельзя, так как до периода выполнения приложение не может определить разрешение экрана у пользователя. Поэтому лучше всего использовать такой код:

```
using System;
class GraphicsPackage
{
   public readonly int ScreenWidth;
   public readonly int ScreenHeighz;
   public GraphicsPackage()
   {
      this.ScreenWidth = 1024;
      this.ScreenHeight = 768;
   }
   class ReadOnlyApp
   {
      public static void Main()
      {
         GraphicsPackage graphics = new GraphicsPackage();
         Console.WriteLine("Ширина = {0}, Высота = {1}", graphics.ScreenWidth, graphics.ScreenHeight);
      }
   }
}
```

На первый взгляд, кажется: это то, что нужно. Но есть одна маленькая проблема: определенные неизменяемые поля являются полями экземпляра, а значит, чтобы задействовать их, пользователю придется создавать экземпляры класса. Может, это и не проблема, и код даже пригодится, когда значение неизменяемого поля определяется способом создания экземпляра класса.

Что нужно сделать если нужна константа, по определению статическая, но инициализируемая в период выполнения? Тогда нужно определить поле с обоими модификаторами – static и readonly, а затем создать особый, статический тип конструктора. Статические конструкторы (static constructor) используются для инициализации статических, неизменяемых и других полей. Здесь предыдущий пример был изменен так, чтобы сделать поля, определяющие разрешение экрана, статическими и неизменяемыми, а также добавлен статический конструктор. Стоит обратить внимание на ключевое слово static, добавленное к определению конструктора:

```
using System;
class ScreenResolution
{
   public static readonly int ScreenWidth;
   public static readonly int ScreenHeight;
   static ScreenResolution()
   {
      // code would be here to
```

```
        // calculate resolution
        ScreenWidth = 1024;
        ScreenHeight = 768;
    }
    class  ReadOnlyApp
    {
        public static void Main()
        {
            Con-
sole.WriteLine("Ширина = {0}, Высота = {1}", ScreenResolution.S
creenMidth, ScreenBesolution.ScreenHeight);
        }
    }
}
```

Вложенные классы

Иногда некоторый класс играет чисто вспомогательную роль для другого класса и используется только внутри него. В этом случае логично описать его внутри существующего класса. Вот пример такого описания:

```
using System;
namespace test
{
    class ClassA
    {
        //Вложенный класс _Класс _Класс
        private class ClassB
        {
            public int z;
        }
        //Переменная типа вложенного класса _Класса
        private ClassB w;
        //Конструктор
        public ClassA()
        {
            w = new ClassB();
            w.z = 35;
        }
        //Некоторый метод
        public int SomeMethod()
        {
            return w.z;
        }
        class Test
```

```
        {
            static void Main(string[] args)
            {
                ClassA v = new ClassA();
                int k = v.SomeMethod();
                Console.WriteLine(k);
            }
        }
    }
}
```

После запуска программа выведет результат:

35

Здесь класс ClassB объявлен внутри класса ClassA, причем со словом private, так что его экземпляры можно создавать только внутри класса ClassA (что и делается в конструкторе класса ClassA). Методы класса ClassA имеют доступ к экземпляру класса ClassB (как, например, метод SomeMethod).

Вложенный класс имеет смысл использовать тогда, когда его экземпляр применяется только в определенном классе. Кроме того, с вложением классов улучшается читаемость кода – если не интересует устройство основного класса, то разбирать работу вложенного класса нет необходимости.

Стоит обратить также внимание на то, как вложенные классы показываются на вкладке ClassView (рис. 3.6).

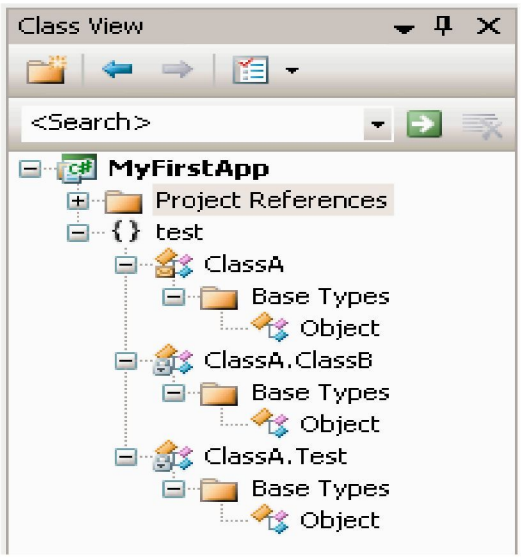


Рис. 3.6. Вкладка ClassView

Контрольные тесты

Контрольный тест 1

1. Какое значение, по умолчанию, возвращает программа операционной системе в случае успешного завершения?

а) 0;
б) программа не возвращает значение;
в) -1;
г) 1.

2. Язык программирования C++ разработал:

а) Никлаус Вирт;
б) Дональд Кнут;
в) Бьерн Страуструп;
г) Кен Томпсон.

3. Какой из ниже перечисленных операторов не является циклом в C++?

а) repeat until;
б) while;
в) do while;
г) for.

4. Чему будет равна переменная a после выполнения следующего кода: int a; for(a = 0; a < 10; a++) {}?

а) 10;
б) 1;
в) 9.

5. Каков результат работы следующего фрагмента кода?

```
1  int x = 0;
2
3       switch(x)
4       {
5
6        case 1: cout << "Один";
7
8        case 0: cout << "Нуль";
9
```

```
10       case 2: cout << "Привет мир";
11
12    }
```

а) Привет мир;
б) НульПривет мир;
в) Один;
г) Нуль.

6. Программа, переводящая входную программу на исходном языке в эквивалентную ей выходную программу на результирующем языке, называется:

а) сканером;
б) компилятором;
в) транслятором;
г) интерпретатором.

7. Чтобы подключить заголовочный файл в программу на C++, например, iostream, необходимо написать:

а) include (iostreamh);
б) include #iostream,h;
в) #include <>; с iostream.h внутри скобок;
г) #include <> с iostream внутри скобок.

8. Какой из перечисленных типов данных не является типом данных в C++?

а) float;
б) double;
в) real;
г) int.

9. До каких пор будут выполняться операторы в теле цикла while (x < 100)?

а) пока x меньше или равен 100;
б) x строго меньше 100;
в) x больше 100;
г) x равен 100.

10. В приведённом коде измените или добавьте один символ, чтобы код напечатал 20 звёздочек *:

```
1  int i, N = 20;
```

```
 2   for(i = 0;  i < N;  i--)
 3       printf("*");
```
а)
```
1   int i, N = 40;
2   for(i = 0;  i < N;  i--)
3       printf("*");
```
б)
```
1   int i, N = 20;
2   for(i = 0;  i < N;  N--)
3       printf("*");
```
в)
```
1   int i, N = 20;
2   for(i = 19;  i < N;  i--)
3       printf("*");
```
г)
```
1   int i, N = 20;
2   for(i = 20;  i < N;  i--)
3       printf("*");
```

11. Какой служебный знак ставится после оператора case ?

а) -

б) .

в) :

г) ;

12. Структура объявления переменных в C++:

а) [==]; < идент. 2>,...;

б) [:=], < идент. 2>,...;

в) [=]; < идент. 2>,...;

г) [=], < идент. 2>,...;

13. Общий формат оператора множественного выбора – switch:

а)
```
1   switch (switch_expression)
2   {
3       case constant1: statement1;  [break; ]
4       case constant2: statement2;  [break; ]
5       case constantN: statementN;  [break; ]
6       [else: statement N+I; ]
7   }
```
б)
```
1   switch (switch_expression)
2   {
```

```
3      case constant1: statement1;  [break; ]
4      case constant2: statement2;  [break; ]
5      case constantN: statementN;  [break; ]
6      [default: statement N+l; ]
7   }
```

в)

```
1   switch  (switch_expression)
2   {
3     case constant1, case constant2: statement1;  [break; ]
4     case constantN: statementN;  [break; ]
5     [default: statement N+l; ]
6   }
```

14. Простые типы данных в C++:

а) целые – bool, вещественные – float или double, символьные – string;

б) целые – int, вещественные – float или double, символьные – string;

в) целые – int, вещественные – float или real, символьные – char;

г) целые – int, вещественные – float или double, символьные – char.

15. Какому зарезервированному слову программа передаёт управление в случае, если значение переменной или выражения оператора switch не совпадает ни с одним константным выражением?

а) contingency;

б) all;

в) other;

г) default.

16. Какие среды программирования (IDE) предназначены для разработки программных средств?

а) MVS, Code::Blocks, QT Creator, AutoCAD, Eclipse;

б) MVS, Code::Blocks, QT Creator, RAD Studio, MathCAD;

в) MVS, NetBeans, QT Creator, RAD Studio, Dev-C++.

17. Укажите правильное определение функции main в соответствии со спецификацией стандарта ANSI:

а)

```
1   void main(void);
```

б)

```
1  int main(void);
```

в)

```
1  int main();
```

г)

```
1  void main().
```

18. Циклом с предусловием является:

а) while;

б) for;

в) do while.

19. Какие служебные символы используются для обозначения начала и конца блока кода?

а) begin end

б) < >

в) ( )

г) { }

20. Какая из следующих записей является правильным комментарием в C++?

а) /* комментарий */

б) {комментарий}

в) ** Комментарий **

г) */ Комментарии */

21. Циклом с постусловием является:

а) do while;

б) for;

в) while.

22. Какой из следующих операторов является оператором сравнения двух переменных?

а) =

б) :=

в) equal

г) ==

23. Что будет напечатано?

```
1   int main()
2   {
3     for (int i = 0; i < 4; ++i)
4     {
5       switch (i)
6       {
7         case 0  : std::cout << "0";
8         case 1 : std::cout << " 1 " ; continue;
9         case 2  : std::cout << "2";  break;
10        default : std::cout << "D";  break;
11      }
12      std::cout << ".";
13    }
14    return 0;
15  }
```

а) 0112.D.
б) 0.1.2.
в) 011.2.D
г) Ошибка компиляции в строке 10
д) 01.2.D.

24. Название C++ предложил:
а) Бьерн Страуструп;
б) Рик Масситти;
в) Кэн Томпсон;
г) Дональд Кнут.

25. Выберите правильный вариант объявления константной переменной в C++, где type – тип данных в C++ , variable – имя переменной, value – константное значение:
а) const type variable := value;
б) const type variable = value;
в) const variable = value;

26. Укажите правильную форму записи цикла do while:
а)

```
1  // форма записи оператора цикла do while:
2  do // начало цикла do while
3  {
4  /*блок операторов*/;
5  }
6  while (/*условие выполнения цикла*/);  // конец цикла do while
```

б)

```
1  // форма записи оператора цикла do while:
2  do // начало цикла do while
3  {
4  /*блок операторов*/;
5  }
6  while {/*условие выполнения цикла*/} // конец цикла do while
```

в)

```
1  // форма записи оператора цикла do while:
2  do // начало цикла do while
3  {
4  /*блок операторов*/;
5  }
6  while (/*условие выполнения цикла*/) // конец цикла do while
```

27. Укажите объектно-ориентированный язык программирования:

а) C++;

б) Eiffel;

в) Java;

г) все варианты ответов верны.

28. Какими знаками заканчивается большинство строк кода в Си++?

а) , (запятая);

б) ; (точка с запятой);

в) : (двоеточие);

г) . (точка).

29. Тело любого цикла выполняется до тех пор, пока его условие:

а) ложно;

б) у цикла нет условия;

в) истинно.

30. Какую функцию должны содержать все программы на C++?

а)

```
1  start()
```

б)

```
1  system()
```

в)

```
1  program()
```

г)

```
1  main()
```

31. Какой оператор не допускает перехода от одного константного выражения к другому?

а) Stop;
б) break;
в) end;
г) точка с запятой.

Контрольный тест 2

1. Что будет напечатано, после выполнения следующего кода: cout << (5 << 3); ?

а) 40;
б) 35;
в) 53.

2. Укажите операцию, приоритет выполнения которой ниже остальных:

а) >>
б) ?:
в) <<
г) &&
д) |
е) ||
ж) ^
з) &

3. Укажите правильное приведение типа данных:

а) char:a;
б) a(char);
в) to(char, a);
г) (char)a;

4. Какой из ниже перечисленных вариантов ответа показывает правильно записанный оператор выбора if ?

а)
1 условное выражение if;
б)
1 if (условное выражение);

в)

```
1 if { условное выражение};
```

г)

```
1 if условное выражение.
```

5. Какое значение будет содержать переменная x?

```
1   #include
2
3   int x;
4
5   int main()
6   {
7    int y;
8     std::cout << x << std::endl;
9     std::cout << y << std::endl;
10   return 0;
11  }
```

а) 0;

б) неопределённое.

6. Тело оператора выбора if будет выполняться, если его условие:

а) истинно (true);

б) ложно (false).

7. Может ли переменная x быть доступна в другом блоке программы?

```
1   int main(int argc, char** argv)
2   {
3       if ( argc > 2 )
4       {
5           int x = 5;
6       }
7       else
8       {
9
10      }
11
12      return 0;
13  }
```

а) да;

б) нет.

8. Ввод данных в C++

а)

```
1  cin » <выражение1> » <выражение2>...;
```

б)

```
1  cin » <выражение1> » <выражение2> » endl »...;
```

в)

```
1  cin » <выражение1>,<выражение2>,...;
```

9. Если условие оператора выбора ложное, то:

а) выполняется тело оператора выбора;

б) выполняется следующий оператор сразу после оператора if;

в) программа завершает работу.

10. Преобразование целочисленной переменной value в ASCII эквивалентно:

а) char (value);

б) atoi(value);

в) cout << value;

г) (char) value.

11. Какое из следующих значений эквивалентно зарезервированному слову true?

а) 0.1;

б) 1;

в) -1;

г) все варианты ответов;

д) 66.

12. Укажите операцию, приоритет выполнения которой выше остальных:

а) /

б) *

в) ++

г) ()

д) +

13. Какое значение будет напечатано?

```
1  #include
2
```

```
3   int main(int argc, char** argv)
4   {
5     int x = 0;
6     int y = 0;
7
8     if (x++ && y++)
9     {
10      y += 2;
11    }
12
13    std::cout << x + y << std::endl;
14
15  return 0;
16 }
```

а) 1;

б) 4;

в) 3;

г) 2.

14. Вывод данных в C++:

а)

```
1  cout « <переменная >,< “< строка выводится на экран>”
   ,<выражение > ,endl;
```

б)

```
1  cout « <переменная >,< “< строка выводится на экран>”
   ,<выражение > ,endl;
```

в)

```
1  cout « <переменная > « “< строка выводится на экран>” «
   <выражение > « endl;
```

15. Результат выполнения следующего фрагмента кода: cout << 22 / 5 * 3;

а) 1;

б) другое;

в) 1.47;

г) 13.2;

д) 12.

16. Результатом выполнения следующего фрагмента кода: !((1 || 0) && 0) является:

а) 0;

б) 1;

в) результат не может быть заранее определен.

17. Что появится на экране после выполнения этого фрагмента кода?

```
1  int a = 1, b = 2;
2  if (a == b);
3  cout << a << " = " << b << endl;
```

а) синтаксическая ошибка;
б) 1 = 2;
в) a = b;
г) вывод на экран не выполнится.

18. Какое значение будет напечатано в результате выполнения программы?

```
1   #include <iostream>
2
3   int main()
4   {
5     int x = 3;
6
7     switch(x)
8     {
9       case 0:
10        int x = 1;
11        std::cout << x << std::endl;
12      break;
13      case 3:
14        std::cout << x << std::endl;
15      break;
16      default:
17        x = 2;
18        std::cout << x << std::endl;
19  }
20
21  return 0;
22 }</iostream>
```

а) 3;
б) 2;
в) 1;
г) ничего не напечатается, программа вообще не будет работать;
д) 0.

19. В каком случае лучше всего использовать приведение типов данных?

а) во всех вышеуказанных случаях;

б) при делении двух целых чисел, для того чтобы вернуть результат с плавающей точкой;

в) чтобы разрешить программе использовать только целые числа;

г) чтобы изменить тип возвращаемого значения функции.

20. Чему равен результат выполнения следующего выражения: 1000 / 100 % 7 * 2 ?

а) 10;

б) 1000;

в) 6;

г) 250.

21. Оператор вывода cout может печатать несколько значений или переменных в одной команде, используя следующий синтаксис:

а)

```
1  cout << "Привет" << name << "n";
```

б)

```
1  cout << "Привет", name, "n";
```

в)

```
1  cout << "Привет" + name + "n";
```

г)

```
1  cout << ("Привет" & name & "n");
```

22. Какое ключевое слово указывает, что целая переменная не может принимать отрицательные значения?

а) другое;

б) positive;

в) long;

г) нет такого зарезервированного слова;

д) unsigned.

23. Какой заголовочный файл следует подключить, чтобы можно было пользоваться приведением типов данных?

а) cmath;

б) cctype;

в) никакого.

24. Почему приведение типов данных может быть небезопасно?

а) можно временно потерять часть данных – таких, как отсечение десятичной части чисел с плавающей точкой;

б) можно навсегда изменить значение переменной;

в) некоторые преобразования не определены компилятором, такие как преобразование символа в целое;

г) нет никаких опасностей.

25. Дано значение 5.9875e17, которое может быть сохранено в переменной типа:

а) long;

б) float;

в) short;

г) int;

д) boo.l

26. Укажите неправильно записанную операцию отношения:

а) <=

б) все операторы записаны правильно

в) =!

г) >=

27. В каком случае можно не использовать фигурные скобки в операторе выбора if?

а) если в теле оператора if всего один оператор;

б) если в теле оператора if нет ни одного оператора;

в) если в теле оператора if два и более операторов;

г) нет правильного ответа.

28. Какой из следующих логических операторов –- логический оператор И?

а) &

б) |&

в) |

г) &&

29. Каков будет результат выражения !(1 && !(0 || 1))?

а) false;

б) true;

в) неоднозначность.

30. Оператор if else позволяет определить действие:

а) только для истинного условия;

б) только для ложного условия;

в) для истинного и ложного условий.

31. Укажите блок кода, в котором переменная y доступна:

```
1    int main(int argc, char** argv)
2    {
3
4     if ( argc > 10 )
5     {
6      }
7     else if (int y = argc - 1 )
8     {
9
10   }
11   else
12   {
13
14   }
15
16   return 0;
17   }
```

а) строки 4 – 15;

б) строки 8 – 15;

в) строки 8 – 17;

г) строки 4 – 17;

д) строки 8 – 11.

32. Какие преобразования типов данных невозможны без потери данных?

а) char to float;

б) float to int;

в) int to float;

г) все перечисленные преобразования невозможны.

33. Результат выполнения следующего фрагмента кода: 54 << 3:

а) 623;

б) 432;

в) 556;

г) 440;

д) нет правильного ответа.

34. Логическая операция с большим приоритетом выполнения:
а) ||
б) !
в) &&

ЛИТЕРАТУРА

1. Дрейер, М. C# для школьников: учебное пособие: пер. с англ. / М. Дрейер; под ред. В. Биллига. – М.: Интернет-Университет Информационных Технологий; БИНОМ: Лаборатория знаний, 2010. – 128 с.

2. Декстер, М. Joomla. Программирование/ М. Декстер, Л. Лэндри. – М.: Вильямс, 2013. – 592 с.

3. Попов, В.Б. Delphi для школьников/ В.Б. Попов. – М.: Бином, 2010. – 320 с.

4. Душистов, Д.В. Решение 50 типовых задач по программированию на языке Pascal / Д.В. Душистов. – Майкоп: Адыгейский государственный университет, 2014. – 68 с.

5. Ушаков, Д.М. Паскаль для школьников / Д.М. Ушаков, Т.А. Юркова. – СПб.: Питер, 2010. – 256 с.

6. Князева, М.Д. Практикум по дисциплине «Информатика и программирование. Программирование на Delphi 7» / М.Д. Князева. – М.: ГОУ ВПО «РЭА имени Г.В. Плеханова», 2010. – 100 с.

7. Вабищевич, П.Н. Численные методы. Вычислительный практикум / П.Н. Вабищевич. – М.: Либроком, 2010. – 320 с.

8. Колдаев, В.Д. Численные методы и программирование / В.Д. Колдаев. – М.: ИД «ФОРУМ», 2009. – 336 с.

9. Дейтел, Х.М. Как программировать на C++ / Х.М. Дейтел, П.Дж. Дейтел. – М.: Бином-Пресс, 2008. – 1454 с.

10. Голицына, О.Л. Языки программирования / О.Л. Голицына, Т.Л. Партыка, И.И. Попов. – М.: ФОРУМ, ИНФРА-М., 2008. – 400 с.

11. Культин, Н.Б. C# в задачах и примерах / Н.Б. Культин. – СПб: БХВ, 2007. – 241 с.

12. Страуструп, Б. Дизайн и эволюция C++ / Б. Страуструп. – М.: ДМК Пресс, 2011. – 445 с.

13. Мозговой, М.В. Классика программирования – алгоритмы, языки, автоматы, компиляторы / М.В. Мозговой. – СПб.: Наука и техника , 2006. – 320 с.

14. Ставровский, А.Б. Первые шаги в программировании. Самоучитель / А.Б. Ставровский, Т.А. Карнаух. – М.: ООО «И.Д. Вильямс», 2006. – 400 с.

15. Уилсон, М. Практический подход к решению проблем программирования C++ / М. Уилсон. – М.: КУДИЦ-ОБРАЗ, 2006. – 736 с.

16. Подкур, М.Л. Программирование в среде Borland C++ Builder с математическими библиотеками Matlab C/C++ / М.Л. Подкур, П.Н. Подкур, Н.К. Смоленцев. – М.: ДМК Пресс, 2006. – 496 с.

17. Керниган, Б. Язык программирования Си / Б. Керниган, Д. Ритчи. – Санкт-Петербург: Невский диалект, 2001. – 352 с.

18. Керниган, Б. Язык программирования С / Б. Керниган, Д. Ритчи. – М.: Вильямс, 2006. – 304 с.

Учебное издание

Лобанова Валентина Андреевна
Воронина Оксана Александровна

АЛГОРИТМИЧЕСКИЕ ОСНОВЫ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Учебное пособие

Редактор Г.В. Карпушина
Технический редактор Н.А. Соловьева

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Орловский государственный университет имени И.С. Тургенева»

Подписано к печати 05.12.2016 г. Формат 60×90 1/16.
Усл. печ. л. 14,2. Тираж 100 экз.
Заказ № _____

Отпечатано с готового оригинал-макета
на полиграфической базе ОГУ имени И.С. Тургенева
302030, г. Орел, ул. Московская, 65.