ТОМСКИЙ ГОСУДАРСИВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

В.В. Кручинин

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Учебное пособие

ТОМСК – 2006

Федеральное агентство по образованию

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра промышленной электроники

В.В. Кручинин

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Учебное пособие

2006

Кручинин В.В.
Технологии программирования: Учебное пособие. — Томск: Томский государственный университет систем управления и радиоэлектроники. — 271 с.

СОДЕРЖАНИЕ

Глава 1. ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

1.1 Основные термины и определения

Программирование это процесс создания программ для ЭВМ [1]. На заре компьютерной эры программирование воспринималось как искусство [2]. Однако, по мере развития систем программирования этот процесс становится все более изученным. Появляются термины *методология программирования* [3], *технология программирования* [4–6], *наука программирования* [7–9] *инженерия программного обеспечения* [10,11]. Однако, в споре о том, что программирование это искусство или наука нет окончательного решения.

Основным, определяющим элементом современных систем программирования является методология. Методология, как философская категория, это система принципов и основ организации и построения творческой и практической деятельности [12]. Таким образом, методология программирования есть система принципов, основ организации и построения программного обеспечения. В настоящее время уже разработано достаточно большое количество методологий. Ниже перечислены наиболее известные:

1) модульное программирование [13,14];
2) структурное программирование [15–17];
3) логическое программирование [18–20];
4) событийно-ориентированное программирование;
5) объектно-ориентированное программирование [21–24];
6) визуальное программирование [25–27].
7) функциональное программирование [28–30];
8) концептуальное программирование [31];
9) доказательное программирование [32];
10) мобильное программирование [33, 34];
11) экстремальное программирование [35–37].

Исторически, многие методологии развивались параллельно и воплощались в различных технологиях программирования и соответственно в системах программирования. Кроме того, методологии модульного, структурного, объектно-ориентированного и визуального программирования — развитие одной ветви, которая имеет универсальный характер применения. Другие, такие как концептуальное, функциональное, логическое программирование также носят универсальный характер, но применение нашли только в области искусственного интеллекта.

Важно разграничить методологию и технологии программирования. Методология программирования больше отвечает за идейную сторону и философский аспект создания программ, в то время как технология базируется на той или иной методологии и несет больше практическую нагруз-

ку по созданию программного обеспечения. В рамках одной методологии может быть большое количество технологий. Таким примером является методология объектно-ориентированного программирования. Технологий, поддерживающих объектно-ориентированное программирование, огромное количество. Например, OLE, COM, RUP и др. [38–43].

Кроме того, методологии программирования можно разделить на два класса [44]: тяжелые и гибкие (облегченные). В первом классе разработчик воспринимается как элемент методологии, который может быть заменен. Поэтому каждый шаг в таких методологиях должен быть подробно описан. Во втором классе, человек воспринимается как существенно-нелинейный элемент методологии [45], а программный код является исходной документацией. В настоящее время наблюдается переход от тяжелых методологий к гибким.

1.2 Жизненный цикл программы

Компьютерные учебные программы можно отнести к сложным программным системам. Это объясняется тем, что, во-первых, учебный процесс слабо формализуем, во-вторых, сама предметная область может быть достаточно сложной для обучения и соответственно для реализации ее модели в компьютерной учебной программе, в-третьих, сложность может представлять направленность КУП для определенной группы обучаемых.

Известно [46], что для сложных программных систем жизненный цикл можно представить в виде шести этапов:

1. Выявление и анализ требований, предъявляемых к компьютерным учебным программам.
2. Определение спецификаций.
3. Проектирование.
4. Кодирование.
5. Тестирование и отладка.
6. Эксплуатация и сопровождение.

Рассмотрим каждый этап этого цикла.

Выявление и анализ требований, как правило, производится с помощью системного анализа. Возьмем за основу методику системного анализа, представленную в работе [47, 48].

Эта методика первоначально предполагает выявление всех «заинтересованных сторон» — участников проблемной ситуации. На рис. 1 представлены основные целеполагающие системы для КУП.

Это прежде всего «обучаемый» — конечный пользователь КУП. Требования со стороны обучаемого можно разделить на три группы:

– психолого-педагогические;
– инженерно-психологические;
– медицинские.

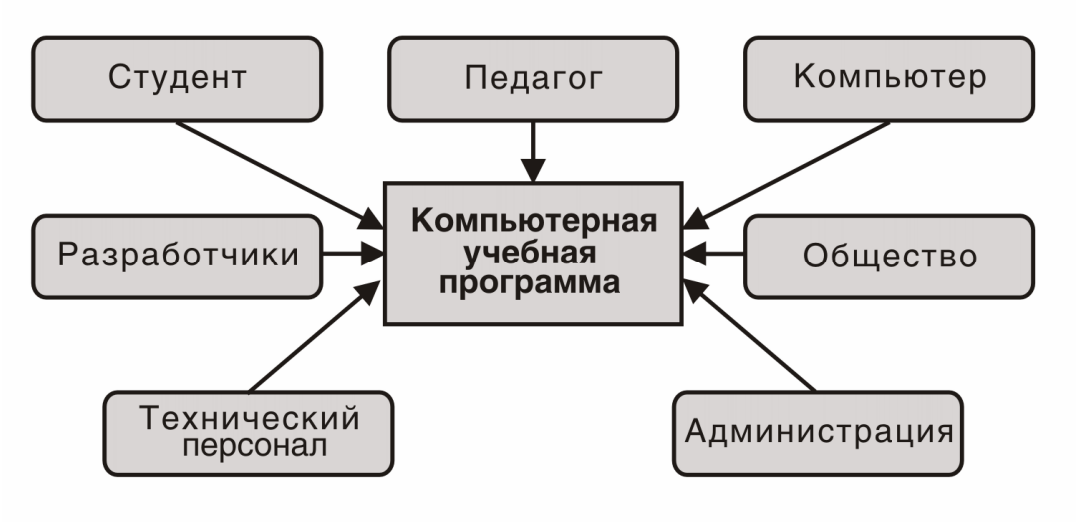


Рис. 1 — Основные целеполагающие системы

Психолого-педагогические требования в целом определяют эффективность учебного процесса. Важнейшим требованием здесь является требование «компьютерная учебная программа должна научить». При этом:

1. КУП должна адаптироваться к физиологическим и психологическим особенностям обучаемого (память, темперамент, реакция, физическое и умственное развитие, возраст, зрение, слух).
2. КУП должна быть основана на деятельностном подходе в формировании психики, эрудиции и нравственных качеств.
3. КУП должна обеспечить постоянную и положительную мотивацию деятельности обучаемого.
4. КУП должна использовать комбинированные приемы обучения, которые развивают и используют как абстрактно-логическое, так и образно-эмоциональное мышление, интуицию обучаемого.
5. КУП должна впитывать в себя последние достижения в области педагогических наук.

Инженерно-психологические требования определяют интерфейс между обучаемым и КУП. Здесь требования будут следующие:

1. Простота работы с КУП.
2. Дружелюбность интерфейса.
3. Приспособление к требованиям конкретного обучаемого, (например, настройка цвета и шрифта текста, возможность увеличения шрифта).
4. Организация комфортного интерфейса.

Медицинские требования определяют факторы КУП, которые влияют на здоровье обучаемого. Эти требования не только определяют влияние компьютера на обучаемого, но и влияние самой КУП. Прежде всего, это касается зрения, психики и нервной системы. Некоторые требования, например время обучения с помощью некоторой КУП для разных групп учащихся, определяются федеральными санитарными правилами, нормами и гигиеническими нормативами.

Педагог непосредственно организует обучение предмета с помощью готовой КУП. Запускает при необходимости программу, наблюдает за ходом работы студента, приходит на помощь при возникновении трудностей. Регистрирует текущие успехи учащегося. Основные требования со стороны учителя следующие:

1. Обеспечение различных форм организации работы с аудиторией — от коллективной до полностью индивидуальной с каждым учащимся.

2. Обеспечение различных видов связи педагога с обучаемыми: электронная почта, доски объявлений, переадресация учащегося к учителю для личного контакта; вмешательство педагога в ход обучения на любой стадии, связь со всеми обучаемыми или с каждым в отдельности; возможность негласного контроля.

3. Различные формы накопления опыта: протоколирование процесса обучения; статистический анализ; регистрация востребуемости тех или иных разделов КУП.

4. Возможность внесения изменений в КУП (по крайней мере, адаптацию КУП для конкретного вида обучения).

Важнейшим требованием администрации является повышение эффективности процесса обучения с использованием компьютерных учебных программ. Здесь эффективность толкуется в самом общем смысле. Т.е. это может быть сокращение прямых и косвенных затрат на образование, или повышение качества обучения, или создание комфортной творческой атмосферы педагогического коллектива. Кроме указанных были выявлены следующие требования: информационное обеспечение административных функций (сбор данных и статистический анализ, составление отчетов); соблюдение требований стандартизации и унификации.

Основным требованием со стороны технического персонала является снижение затрат на эксплуатацию разрабатываемой КУП. Это требование предусматривает:

1) простоту запуска и настройку разрабатываемой КУП;
2) минимизацию объемов требуемой памяти;
3) минимизацию времени выполнения;
4) использование стандартных технических устройств.

В процессе разработки компьютерной учебной программы выделяются три основные составляющие: психолого-педагогическая, организационно-экономическая и техническая. Психолого-педагогическая составляющая обеспечивает педагогические цели и методы достижения их в разрабатываемой КУП.

Организационно-экономическая составляющая обеспечивает реализацию и тиражирование данной КУП при заданных финансовых, трудовых и временных ограничениях. Техническая составляющая собственно реализует КУП в виде программного, информационного и иных обеспечениях.

«Разработчики» — довольно большой коллектив специалистов, состоящий из программистов, психологов, методистов, художников, музыкантов, звуковых режиссеров, мультипликаторов, сценаристов, экономистов, менеджеров по маркетингу и рекламе, юристов, медиков, специалистов по тестированию, организаторов. Условно всех разработчиков можно разделить на 11 групп (см. рис. 2).

Рис. 2 — Основные группы разработчиков компьютерных учебных программ

1. Группа методистов определяет цели, содержание и этапы обучения с помощью разрабатываемой КУП, основываясь на последних достижениях психолого-педагогической науки и практики. Определяет педагогическую технологию обучения. Эта группа формирует учебный материал для представления ее в КУП, разрабатывает контрольные вопросы и задания, определяет алгоритм оценивания знаний. В состав этой группы входят высококвалифицированные эксперты в области преподавания данного предмета (раздела знаний), обладающие большим педагогическим опытом.

2. Группа психологов обеспечивает разработку разнообразных тестов состояния здоровья, тестов способностей и достижений. Также эта группа решает вопросы разработки эффективного интерфейса между студентом и КУП.

3. Группа программистов непосредственно реализует КУП. Программистов можно разделить на системных и прикладных. Системные программисты обеспечивают интерфейс КУП с вычислительной системой

конкретного класса компьютера. Прикладные программисты обеспечивают разработку отдельных модулей КУП.

4. Художественная группа обеспечивает компьютерное представление учебного материала. При этом решаются следующие задачи: выбор или разработка шрифтов для представления текстовой учебной информации. Ввод, редактирование и форматирование текста. Разработка, ввод и редактирование иллюстраций представленных в графической форме. Создание компьютерной анимации и визуальных эффектов. Запись и редактирование видеоклипов или видеофильмов.

5. Музыкальная группа обеспечивает музыкальное сопровождение, разработку и ввод различных звуковых эффектов в КУП. Сопровождение подачи учебного материала голосом диктора.

6. Медицинская группа обеспечивает определение параметров КУП, влияющих на здоровье учащихся. Это прежде всего, предполагаемое время работы с данной КУП.

7. Экономическая группа обеспечивает экономическую обоснованность и целесообразность разработки КУП, производит поддержку финансового состояния предприятия при реализации данного проекта.

8. Группа маркетинга и рекламы обеспечивает разработку стратегии рекламной компании, разработку рекламных материалов (роликов, демонстрационных версий, рекламных листков и писем и др.), определение рынков сбыта и цен на разрабатываемую КУП, работу по заключению дилерских соглашений на тиражирование проектируемой КУП.

9. Юридическая группа обеспечивает правовую поддержку, которая включает разработку вопросов правовой защиты всех заинтересованных сторон: обучаемого, учителя, администрации и разработчиков.

10. Тестирующая группа — довольно большая группа разнообразных специалистов, производящая тестирование разрабатываемой КУП. Прежде всего, это преподаватели, имеющие опыт работы с подобными программами.

11. Группа управления проектом обеспечивает управление, планирование и координацию отдельных этапов разработки компьютерной учебной программы. В эту группу входят руководитель проекта, его помощники по отдельным направлениям, технические работники.

Развитие компьютерной техники привело к ситуации, когда на рынке компьютеров для образования осталось всего два типа. Это IBM PC–подобные и Макинтоши фирмы Apple. Основным требованием для компьютера, предназначенного для обучения, является оснащение его средствами мультимедиа. Другим важным требованием является возможность подключения компьютера в компьютерную сеть Интернет.

Еще одним важным аспектом с точки зрения реализации КУП является выбор программной платформы. В настоящее время имеется достаточно много разнообразных операционных систем: MS DOS, Windows

3.1/95/ NT/2000/2003, Unix, OS/2 и т.д. В настоящее время одной из самых распространенных операционных систем, используемых в образовании, является ОС Windows.

Следует также отметить, что развитие компьютерной техники для образовательных целей не стоит на месте и в недалеком будущем появятся принципиально новые классы компьютеров. Прообразом таких компьютеров являются компьютеры-блокноты. Главные черты новых компьютеров:

1. Основным устройством ввода будет ручка типа шариковой. Здесь компьютер будет распознавать почерк студента.
2. Экран компьютера будет расположен не вертикально, а горизонтально. Таким образом, компьютер будет лежать на столе как обычная тетрадь, в которой можно писать и рисовать.
3. В компьютере будет реализован речевой ввод и вывод информации.
4. Компьютеры будут оснащены беспроводной связью.

В целом общество также выдвигает ряд требований к разрабатываемой КУП. Это требования соблюдения безопасности использования знаний. Т.е. знания, полученные с помощью данной КУП, не должны использоваться во вред людям и природе. Другим важным требованием является повышение культурного уровня обучаемого. Т.е. обучаемый должен получать не только чистые знания по данной конкретной теме, но и выдающиеся примеры приобретения или применения этих знаний.

После выявления требований необходимо провести их анализ и записать спецификацию на разрабатываемую компьютерную учебную программу. Здесь под спецификацией будем понимать описание компьютерной учебной программы, по которому производится ее проектирование. Это описание может быть составлено на естественном языке, или может быть использован некоторый формализованный язык.

В спецификации должно быть записано: платформа и тип компьютера, определен подход к проектированию компьютерной учебной программы, определены виды представления учебного материала (например, использует ли данная КУП аудио- или видео- учебную информацию). Если в данной КУП предусматриваются тесты, то определяются тип вопроса, способы ввода ответа и его анализа, определяются способы оценивания ответов и их фиксации и т.д.

Этап проектирования предполагает разработку алгоритмов и информационной базы, для компьютерного представления учебного материала. На этом этапе готовится учебный материал, определяется его структура, пишется текст, готовятся иллюстрации, подготавливается видеоматериал, подбирается аудиоматериал. Для тестовых программ готовятся вопросы, разрабатываются алгоритмы анализа и оценивания ответов. Разрабатываются алгоритмы представления и предъявления учебной информации.

Ниже будут подробно рассмотрены конкретные методы проектирования для определенного класса компьютерных учебных программ.

На этапе кодирования непосредственно создается компьютерная учебная программа. Кодирование включает создание информационной базы, состоящей из текста, иллюстраций, анимации, аудио- и видео - информации. Причем вся эта учебная информация связана в виде некоторой информационной сети. Например, отдельные фрагменты текста, иллюстраций и т.д. представлены в виде отдельных компьютерных страниц (кадров), которые в свою очередь могут быть связаны с другими кадрами.

Алгоритмы, разработанные на этапе проектирования, реализуются программно с использованием некоторой инструментальной среды.

На данном этапе производится комплексная сборка всех модулей КУП и отладка программы.

Тестирование является необходимым и важным этапом создания КУП. На данном этапе производится проверка функционирования всех составных частей КУП, определяются реальные характеристики. Первоначально тестируются отдельные модули программы. Далее производится тестирование информационного обеспечения. Производится проверка представления и предъявления информации с точки зрения обучения данному предмету, производится определение корректности вопросов и способов их оценивания. Проверяется связанность учебной информации (например, все ссылки должны быть на реальные кадры). Далее производится проверка режима ожидания ввода, т.е. всех ситуаций ввода информации со стороны обучаемого.

1.3 Подходы к проектированию программ

Можно выделить два подхода к проектированию КУП, а именно:

1) кибернетический;
2) информационный.

Сущность кибернетического подхода заключается в следующем:

1. Работа компьютерной учебной программы должна удовлетворять общей теории управления.

2. При реализации этих требований необходимо опираться на психологическую теорию учения.

Отсюда, основываясь на общей теории управления, предлагается:

1) указать цели управления (чему учить);

2) установить исходное состояние управляемого процесса (определить уровень знаний и способность к восприятию соответствующей учебной информации);

3) определить программу воздействий, предусматривающую основные переходные состояния процесса от исходного состояния к целевому (определить эффективную программу обучения);

4) обеспечить систематическую обратную связь, осуществляя сбор информации о состоянии управляемого процесса (обеспечить контроль усвояемости учебной информации);

5) обеспечить переработку информации, полученной по каналам обратной связи, с целью определения качества переходного процесса и выработки корректирующих воздействий.

Информационный подход к обучению с помощью компьютера заключается в следующем: компьютер выдает порцию информации обучаемому; обучаемый вводит ответную порцию информации; тем самым производится обмен информацией между компьютером и обучаемым. Если обмен производится оперативно, то говорят о диалоговом взаимодействии между обучаемым и компьютером. Информационный подход к проектированию КУП приобрел в настоящее время большую популярность, поскольку большинство известных КУП основано на этом подходе. Рассмотрим подробнее этот подход к проектированию КУП.

Работу компьютерных учебных программ можно представить как последовательность элементарных актов, каждый из которых состоит из следующих обобщенных действий:

1) синтез порции учебной информации;
2) отображение порции информации;
3) ожидание ввода ответа;
4) фиксация ответа;
5) анализ ответа.

Как правило, учебная информация в КУП представлена в некоторой информационной базе (это может быть база данных или база знаний). Синтез порции учебной информации предполагает, что элементы этой порции извлекаются из информационной базы и производится их объединение. Далее производится отображение этой порции информации на экран терминала. После вывода порции информации КУП переходит в режим ожидания, при котором обучаемый должен ввести некоторую информацию. Это может быть указание, что делать дальше КУП, например, показать следующую порцию, вернуться к предыдущей, или это может быть ответ на вопрос, поставленный КУП. Режим, при котором КУП ждет некоторого отклика от обучаемого, назовем режимом ввода ответа. В этом режиме может быть несколько стандартных ситуаций: вызов помощи; вызов некоторой вспомогательной программы, например калькулятора; вопрос может быть задан на время и время это истекло и т.д. Фиксация ответа предполагает, что данный ответ записывается в некоторую базу ответов. Ответ может быть записан с некоторыми параметрами, например: время ответа, его порядковый номер, тип вопроса и т.д. При анализе ответа КУП должна определить степень правильности ответа и возможную типичную ошибку, при этом анализ может проводиться с учетом всех предыдущих ответов. Как правило, в процессе проведения анализа определяется оценка ответа.

Самая простая ситуация, когда ответ оценивается без учета предыстории, а сама оценка имеет всего два значения: *правильно* и *неправильно*.

После анализа введенной порции информации КУП синтезирует следующую порцию. Процесс обучения продолжается до тех пор, пока не будет произведен выход из данной КУП.

Таким образом, для разработки КУП при использовании информационного подхода необходимо рассмотреть следующие вопросы:

1) компьютерное представление учебной информации;
2) синтез и компьютерное предъявление учебной информации;
3) организацию ввода информации учащимся;
4) анализ информации, введенной учащимся;
5) оценивание ответов.

Поскольку работать с КУП должен учащийся (неподготовленный пользователь), необходимо также рассмотреть вопросы организации интерфейсов.

1.4 Модели компьютерных учебных программ

1.4.1 Кадровые компьютерные учебные программы

Учебная информация в компьютерной учебной программе может быть предъявлена обучаемому только фрагментарно. Это связано с тем, что терминал компьютера имеет конкретные размеры и параметры отображения информации. Поэтому один из подходов к построению КУП состоит в том, что вся учебная информация разбивается на порции, каждая из которых может уместиться на экране компьютера. Эту порцию стали называть кадром. Первоначально кадр был статичен, т.е. заранее готовилось компьютерное представление учебной информации и, далее при необходимости, эта информация просто выводилась на экран терминала, при этом размеры кадра заранее были фиксированы. В настоящее время понятие кадра стало более сложное:

1) учебная информация, предназначенная для отображения, предварительно синтезируется (например, собирается из некоторого текста и рисунка или фотографии, причем компьютерное представление для текста, рисунка и фотографии может быть различно);

2) отображение кадра производится в окно, которое может иметь различное местоположение и размеры;

3) способ появления кадра в окне может быть различным;

4) для мультимедийных и гипермедийных КУП возможно звуковое оформление (например, кадр появляется с некоторым звуком, после появления кадра может быть слышна речь диктора, дающего пояснения или читающего текст кадра).

Теперь о кадре можно говорить как о некоторой программе или акте некоторого общего сценария по сбору, формированию и предъявлению учебной информации. Обычно кадры, содержащие только учебную информацию, назывались информационными кадрами, а кадры, содержащие вопросы, — контрольными кадрами. Если в кадре кроме текстовой учебной информации используется звуковое сопровождение и/или видеоинформация, то говорят, что данная КУП использует средства мультимедиа.

При работе обучающей программы данного класса кадры последовательно предъявляются обучаемому. Процесс смены кадров назовем листанием. Листание может быть организовано:

1) под управлением КУП;
2) по указанию обучаемого;
3) используется смешанная стратегия.

В первом случае КУП предъявляет обучаемому текущий кадр, и далее КУП заранее знает какой кадр предъявить следующий, а от обучаемого требуется сообщить КУП, что он прочитал данный кадр (или ответил на вопрос в данном контрольном кадре) и ждет предъявления следующего. Такая организация характерна для разнообразных тестовых программ, например для экзаменаторов .

Во втором случае обучаемому самому предлагается осуществлять процесс листания. При этом КУП предоставляет возможность выбора следующего кадра. Например, в простейшем случае это может быть листание в прямом и обратном направлении. Такая организация характерна для разнообразных компьютерных справочников, энциклопедий, информационной части компьютерных учебников, т.е. в КУП, состоящих из информационных кадров.

В смешанной стратегии при просмотре информационной части КУП обучаемому дается возможность самому листать, а в контрольной части листание осуществляет КУП.

После того, как вся учебная информация представлена в виде кадров, необходимо произвести связывание всей совокупности кадров в некоторое целое. Рассмотрим структуры и механизмы связывания кадров.

Механизм связывания может быть внутренним и внешним. При внутреннем связывании в самом кадре содержатся ссылки на другие кадры. При внешнем связывании в КУП задаются некоторые структуры, в которых имеются ссылки на соответствующие кадры, а сами кадры никаких ссылок не имеют.

В общем случае внешние структуры можно разделить на следующие классы: линейные; иерархические; сетевые; смешанные. Для линейных структур характерна жесткая последовательность смены кадров. Листание, как правило, осуществляется только в прямом или обратном порядках. Такую КУП можно представить в виде линейного двухсвязнного списка.

Иерархические структуры позволяют организовать листание более гибко. Весь учебный материал разбивается на темы, темы на подтемы, подтемы на разделы и т.д. Каждый кадр, соответствующий теме или подтеме, имеет специальный механизм выбора или перехода от данной темы к заданной подтеме, а также механизм возврата из нижестоящего в иерархии кадра к соответствующему вышестоящему кадру. Иерархические структуры позволяют довольно быстро находить необходимый кадр.

Сетевые структуры позволяют организовать листание еще гибче, чем иерархические. В данном случае механизмы перехода и возврата организованы на смысловой связанности текста. Если в тексте данного кадра имеются термины или понятия, которые объясняются в других кадрах, то можно осуществить переход от кадра, в котором встретился некоторый термин, к кадру, в котором дано его определение.

Такая организация учебного материала получила название гипертекста [157,158]. Гипертекстовая организация учебного материала в настоящее время получила всеобщее признание. Отчасти это связано с тем, что в основе глобальной компьютерной сети Интернет также лежит гипертекстовая технология организации информации. Рассмотрим подробнее организацию гипертекстовой КУП.

Вся учебная информация разбивается на страницы (кадры). В каждой странице выделяются слова и словосочетания, пояснения которых даются на других страницах. Кроме того, можно явно задать ссылки на другие страницы. Например, «пример решения данной задачи показан на странице 45». Таким образом, формируется некоторое множество ссылок от данной страницы к другим. Обычно листание в гипертекстовых КУП организовано на основе меню. Все ссылки должны быть явно выделены в тексте данной страницы, и обучаемый для перехода на необходимую страницу щелкает мышкой на необходимой ссылке. При этом на экране появляется окно с указанной по ссылке страницей. Другое использование сетевой структуры может быть при задании последовательности изучения учебного материала. Т.е. первоначально задается некоторое подмножество кадров, которые необходимо просмотреть (или изучить) в первую очередь, далее, в зависимости от результатов обучения (это можно проверить с помощью тестов), дается возможность изучать следующее подмножество кадров и т.д.

Если гипертекстовая КУП использует средства мультимедиа, то говорят о гипермедиа КУП. Если все страницы гипермедийной КУП представить узлами, а среду ссылок представить в виде дуг со стрелками, то можно записать ориентированный граф. В общем случае этот ориентированный граф может иметь петли. К недостаткам КУП с гипертекстовой организацией относится то, что ссылки, как правило, организуются внутренние и поэтому нельзя изменять среду ссылок. Т.е. если по какой-либо причине необходимо сократить учебный материал, то необходимо убрать довольно большое количество ссылок.

Применение смешанных структур позволяет использовать достоинства каждой из описанных выше. Например, внутреннюю среду ссылок можно организовать как гипертекст (т.е. переход от кадра к кадру по ссылкам); внешнюю структуру задать в виде дерева, которое хорошо описывает структуру учебного материала, а в виде линейного списка можно задать последовательность кадров, которые обучаемый просмотрел в данный момент. Причем механизм перехода между кадрами включает:

1) переход по гипертекстовым ссылкам;

2) переходы по иерархии тем;

3) переходы по отношению «соседний кадр» (вариант продолжения или возврата);

4) возврат по списку просмотренных кадров (по истории прохождения учебного материала).

Таким образом, модель компьютерной учебной программы, основанную на кадрах можно представить пятеркой {I,Q,V,C,P}.

I — Множество информационных кадров;

Q — Множество контрольных кадров;

V — Множество связей;

C — Вид управления;

P — Протокол.

1.4.2 Модель интеллектуальной компьютерной учебной программы обучения решения задач

Альтернативой кадровым КУП являются интеллектуальные обучающие системы (ИОС) — класс КУП, в основе которого лежит применение идей искусственного интеллекта. Это направление связано с моделированием интеллектуальной деятельности человека в сфере обучения. В западных странах это направление получило название Intelligent Tutorial System (ITS).

Для работы ИОС необходимо иметь базу знаний, в которой должны быть представлены следующие знания:

1) знания из предметной области;
2) знания технологий обучения;
3) знания о конкретном студенте.

Одним из характерных признаков ИОС является наличие модели обучаемого. Различают три основных класса моделей: скалярная; оверлейная, пертурбационная. Скалярная модель простейшая отражающая уровень знания студента в виде некоторого числа. Оверлейная модель представлена в виде некоторого массива или структуры, каждый элемент которого отвечает за конкретный фрагмент знаний. Пертурбационная модель учитывает не только истинные знания студента, но и искажение знаний, в простейшем случае, это типичные ошибки студентов.

Важнейшим свойством ИОС является способность решать задачи и моделировать деятельность эксперта. Это свойство роднит ИОС с экспертными системами и интеллектуальными решателями. Часто интеллектуальный решатель или экспертная система является частью ИОС, которая отвечает за реализацию механизмов семантического вывода на основе базы знаний предметной области.

С развитием сети Интернет ИОС переводятся на веб-технологии. Однако методологически структура ИОС не изменилась. С развитием мультиагентного подхода к построению систем ИИ появляются мультиагентные ИОС.

В настоящее время тематика ИОС становится одной из важнейшей в области проектирования компьютерных учебных программ. Об этом свидетельствует постоянный рост числа публикаций. Например, на поисковом сервере Google более 72000 ссылок на тему «Intelligent Tutoring Systems», на сервере Yahoo свыше 140000.

Рассмотрим обобщенную модель компьютерной учебной программы для обучения решению задач, основанной на идеях ИИ. База знаний такой программы должна содержать:

1) приемы, правила, законы, используемые в данной предметной области;
2) методы представления задач;
3) общие правила поиска решения задач.

Основными функциями КУП обучения решению задач являются следующие:

1) генерация конкретной задачи;
2) решение задачи;
3) анализ хода решения задачи, выполняемого обучаемым;
4) выдача рекомендаций и советов при решении задачи;
5) итоговый анализ решения;
6) планирование хода обучения;
7) ведение протоколов и статистики.

Тогда процесс обучения с помощью подобной обучающей программы может выглядеть так [6]: первоначально на экране появляется некоторая задача, обучаемый может ее решать самостоятельно или с помощью программы, при этом он должен указывать (или вводить каждый шаг). Параллельно программа решает эту задачу самостоятельно. Если обучаемый указывает неверный шаг, то программа может указать на ошибку и дать некоторую рекомендацию. Если обучаемый не может указать следующего шага, то он может запросить помощь. Программа при этом может дать некоторую рекомендацию или подсказать конкретный шаг. После ввода окончательного решения программа производит анализ всего решения, полученного обучаемым, путем сравнения его с собственным. При этом де-

лается вывод о правильности решения и выдаются некоторые комментарии по поводу найденного решения.

Процесс обучения может быть итеративным, т.е. обучаемому может быть предложена некоторая последовательность задач, либо с заранее заданными параметрами, либо программа сама определяет параметры следующей задачи в зависимости от решения предыдущих задач. Кроме того, программа может вести протоколы и различную статистику, например время решения задач и т.д.

Рассмотрим обобщенную структуру такой КУП, показанную на рис. 3:

1. База знаний — предназначена для хранения в формализованном виде знаний о данной предметной области, прежде всего это: способы представления задач; приемы, правила и законы, используемые в данной предметной области; методы поиска решения задач.

2. Планировщик — модуль КУП, который производит процесс планирования обучения, основываясь на использовании базы знаний, возможно на основании предварительного тестирования (например, на основании коэффициента интеллекта и т.д.) и конкретной информации, накопленной в процессе обучения (это ошибки и успехи обучаемого, возможно его намерения и представления). При этом планировщик инициирует параметры для генератора задач.

3. Генератор задачи — модуль КУП, предназначенный для получения конкретной задачи. Генератор задачи — это текстовый конструктор, который, основываясь на параметрах, указанных планировщиком (например, коэффициент сложности), формулирует некоторую задачу. При этом он использует базу знаний.

4. Анализатор шага — модуль КУП, который производит анализ очередного шага обучаемого, записывает результат анализа в базу знаний и выдает соответствующие комментарии. Например, «Молодец, правильно» или «Неверно, используй то-то и то-то» и т.п.

5. Модуль интерфейса — предназначен для организации взаимодействия между обучаемым и КУП. Он организует ввод очередного шага решения, производит вывод разнообразных сообщений, которые формируют другие модули. Одним из возможных режимов работы модуля является диалоговое построение задачи (см. описание работы текстовых конструкторов).

6. Решатель — модуль КУП, предназначенный для решения конкретной задачи, полученной генератором.

7. Советчик — модуль КУП, предназначенный для помощи в процессе поиска решения задачи. По запросу обучаемого выдается некоторая рекомендация, например: «Попробуй использовать правило ...» или это может быть явное указание, например: «Используй такую то формулу». В своей работе данный модуль также использует базу знаний и последовательность шагов, полученную от обучаемого.

Рис. 3 — Обобщенная структура системы
для обучения решению задач

8. Анализатор решения — модуль КУП, который производит итоговый анализ решения обучаемого, сравнивая его с решением, полученным *решателем.* Заранее предполагается, что решатель всегда верно решает задачу. На основании сравнения выставляется некоторая итоговая оценка, и эта итоговая оценка передается планировщику, который определяет тип и параметры следующей задачи.

9. Модуль статистики — обеспечивает ведение протоколов и накопление разнообразны статистических данных.

Таким образом, модель интеллектуальной компьютерной учебной программы для обучения решению задач можно представить шестеркой

$$\{BK, ML, S, G, P, A\},$$

где *BK* — база знаний;
ML — модель обучаемого;
S — решатель;
G — генератор;
P — планировщик;
A — анализатор.

1.4.3 Модель интеллектуальной системы контроля знаний

Компьютерный прием экзаменов становится важным элементом современных образовательных технологий. Практика использования таких программ показала:

1. Существующие типы КУП обычно содержат ограниченное множество вопросов (не более 100–200 вопросов). Основным недостатком компьютерного приема экзамена является то, что спустя некоторое время после использования теста студенты готовят шпаргалки на все вопросы теста. Это приводит к тому, что после непродолжительного использования преподаватель вынужден изменять тест (менять последовательность вопросов, изменять их форму и т.д.), что требует дополнительных затрат времени и усилий преподавателя.

2. Создание программы довольно трудоемкий процесс, требующий существенных затрат времени. Естественно, что получение вопросов требует от преподавателя-методиста немалых затрат времени, к тому же программирование теста требует значительных временных затрат, внимания и определенных навыков от программиста.

3. Практически в любом тесте есть вопросы, сформулированные некорректно (в среднем 2% от общего числа вопросов). Эта проблема возникает в основном по двум причинам. Во-первых, многим преподавателям довольно сложно сформулировать вопросы по теме, которые легко представимы в компьютерном виде. И второй, немаловажной причиной является то, что обучаемые не всегда корректно вводят ответ (используют недопустимые символы).

Для того чтобы использование компьютерных экзаменов было эффективным необходимо, чтобы тестирующие программы отвечали определенным требованиям. Во-первых, для того чтобы тест объективно оценивал знания необходимо, чтобы он содержал достаточное количество вопросов по всему курсу или предмету. Причем, желательно, чтобы вопросы были различных уровней сложности. Во-вторых, для повышения эффективности тестирования, необходимо формировать последовательность вопросов для каждого тестируемого индивидуально, в зависимости от ответов, полученных на предыдущие вопросы теста. При этом, естественно, нужно учитывать уровень сложности вопросов. Возможен вариант, когда

перед основным тестированием проводится мини-тест, направленный на выявление психологических особенностей восприятия тестируемого.

Для достижения наибольшей эффективности проведения компьютерного экзамена предлагается автоматизировать процесс создания вопросов теста, то есть реализовать систему проведения экзаменов, построенную на основе автоматического генератора вопросов. После детального анализа процесса создания компьютерных тестов, рассмотрения литературы и проблем, предложена обобщенная модель системы проведения компьютерного экзамена, представленная на рис. 4.

Рис. 4 — Обобщенная структура системы проведения экзамена

Предложенный подход не только повышает эффективность проведения компьютерного экзамена, но и оптимизирует процесс создания экзамена. В данной модели системы процесс создания теста сводится к созданию модели базы знаний, в основе которой лежит информация по определенному курсу лекций, предмету, теме.

Процесс проведения компьютерного экзамена можно описать следующим образом: первоначально работает планировщик, используя информацию в настройках, возможно, используется психологический тест для определения особенностей восприятия информации, формируется общий план проведения экзамена или, по крайней мере, определяются параметры первого вопроса. Далее, используя эти параметры, работает генератор вопросов. Этот модуль генерирует сам вопрос, причем, формы вопросов могут быть самые разнообразные и зависеть от настроек. Кроме того, генератор строит множество правильных ответов на данный вопрос. Затем сам вопрос подается на вход модуля интерфейса, который обеспечивает визуализацию вопроса и ввод ответа обучаемого. После ввода ответа проверяется его правильность путем сравнения введенного и правильного ответа. Когда задано достаточное для определения уровня знаний обучаемого количество вопросов, выставляется оценка. Для получения оценки определяется количество правильных ответов на вопросы каждого уровня сложности, то есть производится сравнение модели знаний обучаемого и эталонной модели знаний.

Таким образом, модель компьютерной учебной программы для проведения экзамена будет пятерка

$$\{BK, ML, G, P, A\},$$

где *BK* — база знаний;

ML — модель обучаемого;

G — генератор;

P — планировщик;

A — анализатор.

Оценивая описанные выше модели, можно утверждать, что:

1) модель кадровой компьютерной учебной программы является наиболее простой, однако она имеет ряд существенных недостатков (отсутствие гибкости, статичный характер модели, отсутствие модели обучаемого);

2) следующие две модели, достаточно похожи друг на друга (наличие базы знаний, генератора, модели обучаемого, анализатора), однако внутреннее содержание элементов моделей существенно отличаются;

3) существенным элементом, влияющим на характер модели, является генератор, который присутствует во второй и третьей модели.

Глава 2. ПРОЦЕДУРНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Одной из основных технологий создания программ является процедурно-ориентированная технология, которая основана на применении модульного и структурного программирования. Эта технология базируется на применении процедурно-ориентированных языков, таких как Паскаль или Си.

2.1 Для чего нужен язык программирования

За сравнительно небольшое время поменялось несколько поколений компьютеров. Вместе с ними менялось и программное обеспечение. Ясно, что компьютер не может существовать без программ. Задача одна, как быстро и без ошибок создать программу. Однако практика программирования ставила все сложнее и сложнее задачи. История развития автоматизации программирования началась с построения специальных программ — ассемблеров. Вместо программирования в машинных кодах, предлагается простой язык, заменяющий адреса ячеек памяти на имена (поскольку имена легче запоминаются), кроме того, стали заменять числовые коды команд на символьные, например, Add — сложение, Stop — останов и т.д. Появилось понятие метки. Это символическое обозначение адреса некоторой команды в программе. Язык ассемблера стал большим шагом в автоматизации программирования. Он до сих пор используется при написании программ, особенно системного характера. Однако этот язык имеет также много недостатков. На смену ассемблеру появились такие языки программирования, как FORTRAN и COBOL. Первый из них был предназначен для создания программ вычислительного характера, например, научные расчеты.

COBOL, наоборот, предназначен обработки экономической информации, для которой характерны большие объемы входных данных и простые алгоритмы обработки. Успешное применение этих языков дало толчок развитию универсальных языков программирования. Это языки PL/1 и Алгол-60. Достоинством данных языков введение блочной структуры. Дальнейшее развитие программирования связано с идеями направления, названного структурным программированием. Все предыдущие языки программирования позволяли создавать сложную и неуклюжую управляющую структуру программы, которая базировалась на операторе перехода GOTO. Вследствие дискуссий развернутых в научном мире появились идеи создания новых языков программирования, наибольшее распространение из них получили языки программирования Паскаль и Си. Первоначально язык Паскаль создан был для преподавания информатики в университетах, далее он нашел широкое распространение в университетских кругах. Язык программирования Си, был создан как

язык системного программирования, первоначально его предполагалось использовать для разработки операционных систем. На нем была написана операционная система (ОС) UNIX. Эта ОС получила широкое распространение, а язык Си стал хорошим инструментом для создания сложных программных систем.

2.2 Способы описания языка

Для описания языка программирования необходимо задать лексику, синтаксис и семантику. Лексика языка это элементы, из которых строятся слова, выражения и предложения языка.

Синтаксис языка это правила построения правильных выражений и предложений языка.

Семантика задает и исследует смысл предложений.

Проблема описания языка состоит в том, что всевозможных выражений в данном языке может быть достаточно много. Поэтому используют специальные приемы описания языка. Это запись, основанная на грамматиках Бэкуса-Науэра, синтаксические диаграммы и т.д.

2.3 Описание лексических элементов

2.3.1 Понятие лексемы

Лексема — это минимальная смысловая единица для языка программирования. Обычно к лексемам относятся константы, ключевые слова, знаки операций, разделители и т.д.

Процессу проведения лексического анализа (построение лексем из текста программы) предшествует несколько операций компилятора и встроенного в него препроцессора.

Исходная программа записывается в виде текстового файла, созданного каким либо текстовым редактором. Обычно для компилятора основной единицей трансляции является файл, с расширением .C или .CPP.

Препроцессор первый просматривает исходный текст программы и определяет в нем свои специальные директивы (подробное описание директив смотри на стр.) Например, директива ***#include***.

Важно отметить, что препроцессор не знает синтаксиса С.

2.3.2 Пространство между лексемами

Пространство между лексемами определим как множество символов (пробелы, вертикальные и горизонтальные табуляции, символы перевода строки, комментарии), которые игнорируются при трансляции. Эти символы могут служить для определения начала или конца лексемы, но в

процессе трансляции они удаляются. Например, рассмотрим две последовательности:

int i; float f;
и
int i;
float f;

Эти две последовательности символов с точки зрения лексического анализа являются эквивалентными, т.к. имеют всего 6 лексем:

1. ***int***
2. ***i***
3. ;
4. ***float***
5. ***f***
6. ;

2.3.3 Запись длинных строк

Для записи длинных строк символов используется обратная косая черта (\). Она ставится в конце строки. Обратная косая черта и символ перевода строки игнорируются, две строки (и более) воспринимаются как одно целое. Например,

"Томский государственный
университет систем управления
и радиоэлектроники"

Здесь записана одна длинная строка символов.

2.3.4 Комментарии

Комментарии представляют собой фрагменты текста, предназначенные для записи пояснений. Комментарии предназначены для программистов, которые будут читать исходный текст. Комментарии в процессе трансляции программы игнорируются. Комментарии можно записать двумя способами. В первом способе комментарий открывается парой символов /*, а закрывается символами */. Например,

int /* объявить */ i /* счетчик */;

В процессе трансляции будет получено всего три лексемы: ***int i ;***

Второй способ записи комментария в C++ состоит в записи двух подряд символов «косой черты» (//). Комментарий начинается от этих символов (//) и заканчивается символом перевода строки. Например,

```
class X {// это комментарий
… };
```

Комментарии могут быть вложенными. Например,

```
/* пример
  /*написания */
 //вложенного
комментария */
```

Следует быть осторожным в использовании /* и // одновременно. Это может привести к нежелательным последствиям. Например,

```
int i = j//* разделить на k */k;
+m;
```

здесь вместо выражения

```
int i=j/k;
+m;
```

получим

```
int i=j+m;
```

2.4 Лексемы

В C++ различаю 6 классов лексем: ключевые слова, идентификаторы, константы, символьные строки, знаки операций и разделители. Рассмотрим каждый класс в отдельности.

2.4.1 Ключевые слова

Ключевые слова — это слова, зарезервированные для специальных целей. Они не могут использоваться в качестве идентификаторов. Ниже приведена таблица основных ключевых слов C++.

Asm	*auto*	*break*	*case*
cdecl	*char*	*class*	*const*
continue	*default*	*delete*	*do*
double	*else*	*enum*	*extern*
float	*for*	*friend*	*goto*
huge	*if*	*inline*	*int*
interrupt	*long*	*near*	*new*
operator	*pascal*	*private*	*protected*
public	*register*	*return*	*short*
signed	*sizeof*	*static*	*struct*
switch	*template*	*this*	*typedef*
union	*unsigned*	*virtual*	*void*
volatile	*while*		

2.4.2 Идентификаторы

Идентификатор — это последовательность букв или цифр, начинающийся с буквы. В C++ для записи идентификатора разрешено использовать следующие символы:

a b c d e f g h i j k l m n o p q r s t u v x y z _
A B C D E F G H I J K L M N O P Q R S T U V X Y Z
1 2 3 4 5 6 7 8 9 0

Идентификаторы предназначены для именования классов, объектов, функций, переменных, типов данных определенных пользователем и т.д. Длина идентификатора может быть любой, однако учитываются первые 32 символа.

При записи идентификатора в C++ строчные и прописные буквы различаются. Например, ***Sum, sum, suM*** являются различными идентификаторами.

2.4.3 Константы

Константы предназначены для записи фиксированных числовых и строковых значений. C++ поддерживает четыре класса констант: числа с плавающей запятой, целые, символьные строки и перечисления.

Целые константы могут быть десятичными, восьмеричными и шестнадцатеричными. При отсутствии каких либо суффиксов тип целой константы определяется по его значению (см. табл.)

2.4.3.1 Десятичные константы

Разрешены десятичные константы в интервале 0 4294967295. Константы превышающие данный интервал урезаются. Десятичные константы, в отличии от восьмеричных, не должны иметь начального нуля. Например,

int i=10; //десятичное 10
int i=010; //восьмеричное 10 (8 десятичное)
int i=0; // десятичный 0 = восьмеричный 0

2.4.3.2 Восьмеричные константы

Все целые константы, начинающиеся с нуля являются, являются восьмеричными. Восмеричная константа не должна содержать цифры 8 и 9, в противном случае будет сгенерирована ошибка.

2.4.3.3 Шестнадцатеричные константы

Все шестнадцатеричные константы должны начинаться с символов 0x (или 0X). Например,

```
int i=0x0a; //шестнадцатеричная 0a – десятичная 10
```

Таблица констант без символов U и L

десятичные константы

от 0 до 32767	***int***
от 32768 до 2147483647	***long***
от 2147483647 до 4294967295	***unsigned long***
>4294967295	***truncated***

восьмеричные константы

от 00 до 077777	***int***
от 0100000 до 0177777	***unsigned int***
от 02000000 до 017777777777	***long***
от 020000000000 до 037777777777	***unsigned long***
>037777777777	

Шестнадцатеричные константы

от 0x0000 до 0x7FFF	***int***
от 0x7FFF до 0xFFFF	***unsigned int***
от 0x100000 до 0x7FFFFFF	***long***
от 0x800000000 до 0xFFFFFFFF	***unsigned long***
>0xFFFFFFFF	

2.4.3.4 Суффиксы для без знаковых и длинных целых чисел

Суффикс L (или l) присоединяется к константе для того, чтобы указать ее тип long. Суффикс U (или u) для указания, что константа без знаковая. Возможно использование двух суффиксов одновременно (ul, lu, UL и т.д.)

2.4.3.5 Символьные константы

Символьная константа — это один или несколько символов заключенные в одиночные кавычки. Например, ‘A’, ‘:’, ‘\n’. В C++ одиночные символьные константы имеют тип char. Если в одиночные кавычки заключено несколько символов, то данная константа имеет тип int.

2.4.3.6 Использование обратной косой черты (\)

Обратная косая черта используется для графического представления символов, которые не имеют обычного графического начертания. Например, константа \n обозначает символ перевода строки.

Обратная косая черта также используется для записи символов ASCII-таблицы или управляющих символов в восьмеричном и шестнадцатеричном коде. Например, ‘\03’ — Ctrl/C или ‘\x3F’ для знака вопроса. Для представления символов можно после косой черты записать не более трех чисел в шестнадцатеричном или восьмеричном коде, при этом записанной число должно лежать в интервале [0,255]. Если записанное число будет больше чем 255, то компилятор выдаст ошибку. Если при записи числа будут использованы символы, не относящиеся к восьмеричным или шестнадцатеричным, то концом числа будет считаться предыдущий символ. Например, \339 — здесь цифра 9 не является восьмеричной, поэтому будет рассматриваться только \33.

Таблица

\a	0x07	BEL	звонок
\b	0x08	BS	забой
\f	0x0C	FF	
\n	0x0A	LF	перевод строки
\r	0x0D	CR	возврат каретки
\t	0x09	HT	табуляция(гориз.)
\v	0x0B	VT	табуляция(верт.)
\\	0x5c	\	обратная косая черта
\’	0x27	‘	апостроф
\"	0x22	"	двойные кавычки
\?	0x3F	?	знак вопроса
\O			O — три восьмеричных числа
\xH			H — строка шестнад-цатеричных цифр
\XH			H — строка шестнад-цатеричных цифр

2.4.3.7 Константы с плавающей запятой

Константа с плавающей запятой состоит из:
– целого числа;
– десятичной точки;
– дробной части;
– e или E со знаковой экспонентой;
– суффикса типа: f или F или l или L.

При записи константы с плавающей запятой целая или дробная части могут отсутствовать (но не одновременно), символ e (или E) или десятичная точка также могут отсутствовать (но не одновременно).

Например

константа	значение
23.45e6	23.45×10^6

2.4.3.8 Типы данных констант с плавающей запятой

По умолчанию константа с плавающей запятой имеет тип double. Чтобы изменить тип необходимо явно записать суффикс типа (f,F,l,L). Для указания константы типа float необходимо запись суффиксы f или F. Суффиксы L и l дают константе тип long double.

2.4.4 Перечисляемые константы (enum)

Перечисляемый тип предназначен для записи последовательности целых чисел в мнемоническом виде. Перечисляемый тип записывается с помощью ключевого слова enum, за которым стоит идентификатор перечисления, за ним в фигурных через запятую скобках, список идентификаторов. Например,
enum Color {Black, Blue, Green, Red};

Здесь Black, Blue, Green, Red перечисляемые константы типа Color, которые могут присутствовать в выражениях C++. При этом по умолчанию будут присвоены следующие значения.

Black=0, Blue=1, Green=2, Red=3

Возможны также следующие варианты записи:

enum Color {Black, Blue=4, Green, Red=Black+1};
Black=0, Blue=4, Green=2, Red=5

2.5 Синтаксические структуры C

2.5.1 Объявления

В этом разделе рассматриваются вопросы связанные с объявлениями. К ним относятся следующие понятия: объект, тип, класс памяти, время действия, видимость, время существования, связывание. Рассмотрим каждое из этих понятий.

2.5.2 Объект

Под объектом будем понимать поименованный блок оперативной памяти, в котором может содержится некоторое фиксированное или переменное число значений (или множеств значений). С каждым значением связывается имя и тип (тип данных). Имя используется для доступа к объекту. Это имя может быть простым идентификатором, или

может быть сложным выражением, которое указывает на конкретный объект. Тип объекта используется для следующих целей

1. Для определения размера памяти необходимого для объекта.
2. Для определения составляющих объекта, для сложного объекта.
3. Для выполнения операций проверки и преобразования типа.

В С имеется множество стандартных и определенных пользователем типов данных. Это знаковые и без знаковые целые числа различных размеров, числа с плавающей запятой с различными точностями представления, структуры, объединения, массивы и классы.

Объявления в С необходимо для организации объекта и связывания объекта с некоторым именем.

Каждое объявление ассоциируется с идентификатором и типом данных. Большинство объявлений являются определяющими объявлениями, которые обеспечивают автоматическое создание объекта и его инициализацию. Другие объявления являются ссылочными и предназначены для указания того, что такой объект уже существует, и необходимо указать его имя и тип. Пример такого объявления может быть в случае, когда программа собирается из отдельных модулей (файлов) и некоторые объекты объявлены в одном файле, а используются в другом. Ссылочных объявлений может быть несколько, определяющее объявление должно быть одно.

В С все идентификаторы могут использоваться только после объявления. Однако в некоторых случаях (особенно при описании) необходимо использовать имя до явного объявления. Для этих целей используется опережающая ссылка, когда имя метки, функции, структуры, объединения или класса записывается до его объявления.

2.5.3 Левое значение (Lvalue)

Левое значение — это значение выражения, которое стоит слева от операции присваивания, предназначено для определения адреса объекта. Например, в выражении *P=10;, где P есть некоторый указатель на существующий объект.

2.5.4 Правое значение (Rvalue)

Выражение a+b не быть левым значением, поскольку нельзя записать a+b=a и оно не определяет адрес объекта. Такие выражения часто называют правыми значениями (Rvalue).

2.5.5 Тип и класс памяти

При объявлении объекта необходимо указать его тип и класс памяти. Класс памяти определяет размещение объекта в оперативной памяти. Это может быть сегмент данных, регистр, куча и стек.

Сегмент данных предназначен для хранения объектов, которые всегда присутствуют при работе программы (это статические объекты).

Регистр это сверхоперативная память расположенная в процессоре. Значения переменных в этих регистрах могут храниться очень короткое время. Однако временная эффективность использования этих регистров очень высока.

Куча — специальная память, управляемая операционной системой, предназначенная для хранения объектов, которые создаются в процессе работы программы. Это так называемая динамическая память.

Стек это специальная память операционной системы, предназначенная для хранения локальных объектов.

Класс памяти объекта определяется с помощью синтаксиса языка или по местонахождению объявления в программе.

Тип, как было записано ранее, предназначен для определения размера объекта и структуры памяти.

2.5.6 Область действия

Область действия идентификатора определяется как часть программы, в которой с помощью данного идентификатора имеется доступ к его объекту. Имеется пять категорий: блок, функция, прототип функции, файл, класс. Область действия зависит от от того, как и где объявлен идентификатор.

2.5.7 Блочная область действия

Область действия идентификатора объявленного в блоке начинается от места объявления и завершается в конце данного блока. Блок определяется фигурными скобками {...}. Параметры объявленные в определении функции имеют также блочную область действия.

2.5.8 Область действия — вся функция

Только метки имеют область действия всю функцию. Они могут быть определены в любом месте функции. Переход на них может быть также в любом месте функции. Имя метки в теле функции должно быть уникальным.

2.5.9 Область действия прототип функции

Идентификаторы, записанные в списке объявлений параметров, при описании прототипа функции, имеет область действия — прототип функ-

ции. Область действия начинается от объявления идентификатора и заканчивается концом прототипа функции.

2.5.10 Область действия — файл

Файловая область действия идентификатора устанавливается тогда, когда идентификатор объявляется вне функций или блоков, вне классов и прототипов. Такие идентификаторы называются глобальными. Их область действия начинается с объявления и закачивается концом исходного файла.

2.5.11 Пространство имен

Под пространством имен будем понимать область действия идентификатора, в котором он должен быть уникальным. В C++ имеется четыре класса идентификаторов:

1. Имя метки для оператора ***goto.*** Оно должно быть уникальным в теле функции.
2. Имена структур, объединений и перечислений. Они должны быть уникальными в блоке. Если они объявлены вне блоков, то они должны быть уникальны для всей программы в целом.
3. Имена членов структур и объединений. Они должны быть уникальны в структуре или объединении, в котором они объявлены. Имена никоим образом не влияют друг на друга если они объявлены в разных структурах или объединениях.
4. Переменные, определения ***typedef***, функции, члены перечислений. Они должны быть уникальны в своих областях действий.

2.5.12 Видимость

Видимость идентификатора это свойство, определяющее возможность доступа к объекту в в исходном тексте программы. Видимость обычно совпадает с областью действия, однако имеются случаи, когда объект временно становится невидимым из-за появления дублирующего идентификатора. В этом случае объект еще существует, но его идентификатор не может быть использован, пока не завершится область действия дублирующего идентификатора. Например,

```
{
 int i; char ch; //локальные идентификаторы
 i=3;                    //i и ch видимы
...
 {
  double i;
```

```
  i=3.0e3;       //double i видим
                 //int i=3  скрыт
  ch='A';        //char ch видим
 }
                 //для double i  область действия
                 //завершилась
i++;                  //int i видим и i=4
...              // char ch видим
}
...              //области действия для char ch и int i завершены.
```

2.5.13 Время существования

Время существования это период, в течение которого объявленный идентификатор имеет реальный, физический объект распределенный в оперативной памяти. Время существования скрытно связано с классом памяти. Время существования может быть:

– статическим;
– локальным;
– динамическим;

2.5.14 Статические объекты

Под статические объекты память распределяется во время загрузки программы в оперативную память. И они существуют, пока программа не будет выгружена из памяти. Статические объекты находятся в сегменте данных. Необходимо отметить, что имена функций, где бы они не были объявлены, являются статическими объектами. Все переменные, имеющие область действия файл, также являются статическими. Другие переменные могут быть статическими, если они объявлены с ключевыми словами ***static*** или ***extern***.

Статические объекты обнуляются, если они не имеют инициализации или конструктора.

2.5.15 Локальные объекты

Локальные объекты или автоматические объекты создаются в стеке при входе в функцию или блок и уничтожаются при выходе из функции или блока. Локальные объекты должны содержать инциализаторы внутри блока или функции. Область действия локальных объектов является функция или блок.

Все идентификаторы объявленные внутри функции или блока по умолчанию являются локальными.

2.5.16 Динамические объекты

Динамические объекты создаются и удаляются в любом месте программы на этапе выполнения программы с помощью вызова специальных функций. Все динамические объекты размещаются в куче (см. описание классов памяти), либо с помощью библиотечных функций типа ***malloc***, либо с помощью оператора C++ — ***new***. Память, выделенная для динамического объекта, освобождается с помощью вызова библиотечной функции типа ***free*** или с помощью оператора ***delete.***

2.6 Объявления

2.6.1 Типы объявлений

Объявления возможны:
1) переменных;
2) функций;
3) типов;
4) структур;
5) битовых полей;
6) объединений;
7) массивов;

2.6.2 Объявление переменных.

Переменные можно объявлять внутри функции и вне функции.

Переменные объявленные внутри функции будут локальными, время их действия определено рамками функции. Если переменная объявлена вне функции, то она автоматически становится глобальной и доступ к ней имеют все функции, записанные в данном файле.

При объявлении переменной необходимо указать тип. Имеются следующие определенные типы данных:

char, int, signed, double, long, unsigned, float, short

Переменные могут быть объявлены следующим образом:

char c; *//объявить переменную с типа char;*
int i, j*;//объявить переменные i и j как целые;*
double f;*//объявить переменную f плавающего типа*
long L;// *объявить переменную L типа длинного целого*

2.6.3 Объявление массивов

Массив это объект в Си содержащий последовательность однотипных элементов. Важно отметить, что элемент массива также является объектом. Взять значение элемента массива можно по номеру, называемого индексом. Для объявления массива необходимо указать тип элементов массива и его размерность. Например,

int vek[10];

В этом объявлении описан целый массив с именем vec, размерность этого массива 10 элементов. Первый элемент массива имеет индекс 0, второй элемент 1 и т.д. Чтобы взять значение элемента массива необходимо записать имя массива и в квадратных скобках указать номер элемента, например,

j=vec[0]; *//j присвоить значение первого элемента массива;*
k=vec[9]; *//k присвоить значение последнего элемента массива;*

Вместо константы при записи индексов может стоять выражение. Например,

vec[i]=i; //берется значение i, определяется соответствующий элемент
// массива и присваивается значение переменной i
vec[j*2+1]=0; //вычисляется значение выражения в квадратных скобках, это будет индекс элемента массива и этому элементу массива присваивается значение 0.

Можно объявить многомерный массив, например матрица с именем Matr может быть объявлена так:

int Matr[5][10]; //здесь объявлена матрица размерностью 5×10 целых элементов, иными словами можно сказать, что объявлено 5 векторов размерностью 10. Можно объявить трехмерный массив, например
int Tenz[2][5][10] ; //объявлено две матрицы размерность 5×10 целых элементов

Доступ к элементам многомерных массивов аналогичен как для одномерного массива.

Присвоение начальных значений для массивов производится следующим образом: объявляется массив и в фигурных скобках записываются значения элементов массива, разделенных запятой. Например,

int vector[5]= { 1, 2, 20, -5, 8 };
тогда ***vector[0]*** равен 1, ***vector[1]*** равен 2, ***vector[2]*** равен 20, ***vector[3]*** равен -5, ***vector[4]*** равен 8.

Для матрицы можно записать начальные значения построчно, например

double ObrMatr[2][3]= { { 1.0, 2.0, 8.5 } //первая строчка
{ 2.9, 1.9, 3.4 }//вторая строчка
};
тогда ***ObrMatr[0][0]*** равен 1.0, ***ObrMatr[0][1]*** равен 2.0, ***ObrMatr[0][2]*** равен 8.5, ***ObrMatr[1][0]*** равен 2.9, ***ObrMatr[1][1]*** равен 1.9, ***ObrMatr[0][1]*** равен 3.4 .

2.6.4 Объявление строк символов

Строка символов специальный массив в Си, элементы которого содержат коды символов. Объявление строки символов производится следующим образом:

char str[20]; //объявить строку символов длинной 20.

Важно помнить, что строка символов это тот же массив, только на него накладывается ограничение, последним элементом строки символов должен быть ноль. Например,

char str[20]={ "привет" };

Это означает, что ***str[0]*** равен ‘п’, ***str[1]*** равен ‘р’, ***str[2***] равен ‘и’, ***str[3]*** равен ‘в’, ***str[4]*** равен ‘е’, ***str[5]*** равен ‘т’, ***str[6]*** равен 0. Остальные значение элементов неопределенны и могут иметь самые разные значения.

Важно отметить, что для строк символов существует достаточно большая библиотека стандартных функций, которые описаны в специальном файле ***string.h***

2.6.5 Объявление структур

Структура это конструируемый тип, который описывает объект, содержащий некоторое количество разнотипных объектов. Например, если необходимо записать обобщенную информацию об адресе человека, то необходимо указать:

фамилию, имя, отчество,

город, улицу, номер дома, номер квартиры,

почтовый индекс,
телефон домашний,
телефон рабочий,
адрес электронной почты и т.д.

Мы бы хотели, чтобы это вся эта информация была воспринята как один объект (один сплошной участок памяти), то в Си необходимо записать структуру:

```
struct Address{
  char Name1[20];        //имя
  char Name2[20];        //фамилия
  char Name3[20];        //отчество
  char Town[30];         //город
  char Street[30];       //улица
  int Home;              // номер дома
  int Flat;              // номер квартиры
  long HomeTelephone;  // номер домашнего телефона
  long WorkTelephone;   //номер рабочего телефона
  char Email[100];       //электронная почта
};
```

Записав описание структуры можно объявить соответствующие объекты, например,

```
struct Address p1, MyAddress, AddressBook[10];
```

тогда к элементам структурных объектов можно обращаться следующим образом: первоначально записывается имя объекта, затем точка, затем имя элемента. Например,

```
p1.Home=28;
MyAddress.Flat=53;
AddressBook[0].WorkTelephone=233456;
```

Объектам типа структуры можно присвоить начальные значения. Например, для структуры Address можно записать следующее:

```
struct Address MyAddress={ "Кручинин", "Владимир", "Викторович",
 "Томск", "Киевская", 28, 53,  249604, 413572, "kruvv@mail.ru" };
```

Для массивов структур правило записи аналогичны массивам:
Например,

struct Address AddresBook[10]= {{начальные значения для первого элемента}, { начальные значения для второго элемента }, ...{ начальные значения для последнего элемента }};

2.6.6 Объявления объединений

Объединение предназначено для создания объекта, который может иметь несколько типов. Например, можно создать объект, который рассматривается как целое число и как массив, состоящий из двух байтов.

Объединение записывается, как структура только вместо ключевого слова ***struct*** записывается ключевое слово ***union***. Например,

union xxx { int k; char s[2]; };

В этом примере объединение xxx можно воспринимать как целое или как массив из двух байт. Причем изменение в ***k*** ведет к изменению в массиве и наоборот. Например***, xxx.k=10;*** при этом присваивании ***xxx.s[0]*** равно 0, а ***xxx.s[1]*** будет равно 10. Если ***xxx.k=256*** то ***xxx.s[0]*** равно 1, а ***xxx.s[1]*** равно 0. (в этом примере предполагается, что целое занимает 16 бит).Можно объединять любое количество элементов. Размер памяти, занимаемой объединением, равен элементу с максимальным размером.

2.6.7 Объявление собственного типа (*typedef)*

В Си предусмотрена возможность объявления собственного типа. Это делается с помощью специального ключевого слова ***typedef.*** Например,

```
typedef
 struct TagPoint
 {
  int x;
  int y;
  int color;
} POINT;
```

В этом примере объявлена структура ***TagPoint*** как новый тип ***POINT.***

Тогда можно объявить объекты нового (пользовательского) типа. Например,

```
  POINT t, z, Izo[10][10];
```

Обращение к соответствующим элементам записывается по правилам обращения к элементам структуры. Например,

t.x или t.y или Izo[i][j].x

2.6.8 Битовые поля

Можно объявить целые знаковые и без знаковые числа как битовые поля. Например,

```
struct MyStruct {
 int i              :2;
 unsigned j         :5;
int                 :4;
int k               :1;
int m               :4;
}                   a, b, c;
```

В этой записи объявлены три битовых поля a, b и c. Распределение битов начинается с самого младшего (0-го бита). Длинна (количество бит выделенных для данной переменной) записывается через двоеточие (см. выше). Ниже показано битовое распределение в 16-разрядном целом числе битовых полей a, b, c.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
m				k	не используется				j					i	

Рис. 5

Переменные битовых полей могут быть знаковыми и без знаковые. Если размер переменной битового поля равен 2, и ее значение «11», то для знакового это будет (–1), а для без знакового будет 3. Если встретится, например, a.i=6; то в результате выполнения будет a.i равное (–2), поскольку 6 это двоичное 110, ширина i равна двум, после присвоения a.i будет содержать битовое 10, и i объявлена как int, тогда битовое 10, будет интерпретироваться как –2.

Битовые поля могут быть объявлены в структурах и объединениях. Правило доступа к их элементам точно такие как у структур.

Следует предупредить, что выражение &mystruct.x является ошибочным, если x есть идентификатор битового поля. Поскольку оно не гарантирует, что может вычислить адрес этого элемента.

2.7 Функции

Каждая программа в Си должна обязательно иметь специальную функцию по имени main. Эта функция является точкой входа в программу. Функции обязательно должны иметь прототипы. Прототип это описание функции на уровне входа и выхода. Например, если функция ***MyFunc*** имеет на входе два целых параметра и возвращает целое значение, то необходимо записать следующий прототип:

int MyFunc(int a, int b);

имена параметров в прототипе могут отсутствовать:

int MyFunc(int , int);

Обычно прототипы функций записывают в специальных файлах называемых заголовочными (header). Они обычно имею расширение h.

Для записи определения функции необходимо:

1) описание типа возвращаемого значения;
2) имя функции
3) список параметров в скобках;
4) тело функции, заключенное в фигурные скобки.

Например,

```
int       //описание типа возвращаемого значения
 MyFunc //имя функции
   (int a, int b) //описание списка параметров
{                  //тело функции
  return a+b;   //возврат значения
}                  //закрывающая скобка, конец описания функции
```

2.8 Основные операции в Си

2.8.1 Унарные операции

& вычисление адреса объекта;
* вычисление значения косвенно;
+ унарный плюс;
- унарный минус;
~ инвертирование побитное;
! логическое отрицание;
++ автоинкремент;
-- автодекремент.

2.8.2 Бинарные операции

+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Остаток от деления на цело
<<	Сдвиг в лево
>>	Сдвиг вправо
&	Побитовое «и»
^	Побитовое исключающее «или»
|	Побитовое «или»
&&	Логическое «и»
||	Логическое «или»
=	Присвоение
*=	Присвоение с умножением
/=	Присвоение с делением
%=	Присвоение с операцией взятия остатка
-=	Присвоение с вычитание
<<=	Присвоение с левым сдвигом
>>=	Присвоение с правым сдвигом
&=	Присвоение с операцией побитового «и»
^=	Присвоение с операцией побитового исключающего «или»
|=	Присвоение с операцией побитового «или»
<	Меньше чем
>	Больше чем
<=	Меньше равно
>=	Больше равно
==	Равно
!=	Не равно
A ? x; y	Если a то x иначе y

Рассмотрим некоторые операции в С

2.8.3 Побитовые операции

Выражения в побитовых операциях должны быть целого типа. Правила для выполнения побитовых операций записаны в следующей таблице.

Бит в E1	Бит в E2	E1&E2	E1^E2	E1|E2
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Рассмотрим примеры
int x=10, y=18;

x=	0000000000001010	
y=	0000000000010010	
x&y	0000000000000010	результат побитового «и»
x|y	0000000000011010	результат побитового «или»
x^y	0000000000011000	результат исключающего или

2.8.4 Операции сдвига

Операция сдвига влево <<.
Эта операция производит сдвиг влево на указанное число бит.

k=i<<2; *//сдвиг на 2 бита влево*

Операция сдвига вправо >>.
Эта операция производит сдвиг вправо на указанное число бит.

k=i>>3; *//сдвиг на 3 бита вправо*

2.9 Выражения

Выражения в Си строятся из идентификаторов, констант, вызовов функций, операций и разделителей.
Примеры выражений:

x=a+b; //найти суммы и присвоить ***x***;

x=(y=a+20)+6; //найти сумму ***a+20*** и присвоить y, затем найти сумму ***y+6*** и //присвоить x;

t=(x>y)?x:y; //t присвоить если x>y то x иначе y

2.10 Операторы

2.10.1 Составной оператор

Группирование операторов производится с помощью фигурных скобок и эта конструкция может быть рассмотрена как один оператор. С другой стороны в некоторых языках программирование вводится понятие блока. Блок это выделенная группа операторов, в которой можно объявлять локальные переменные. Например,

{ x=20; y=30; }

2.10.2 Условный оператор

Условный оператор в Си предназначен для организации ветвления. Существует две синтаксические конструкции условного оператора:

1. ***if (***<выражение>***) <оператор>*** (рис. 6)
2. ***if (<выражение>)<оператор 1> else <оператор 2>*** (рис. 7)

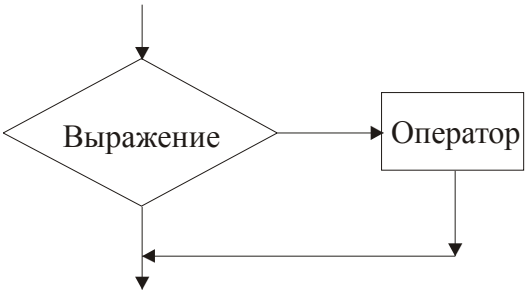


Рис. 6

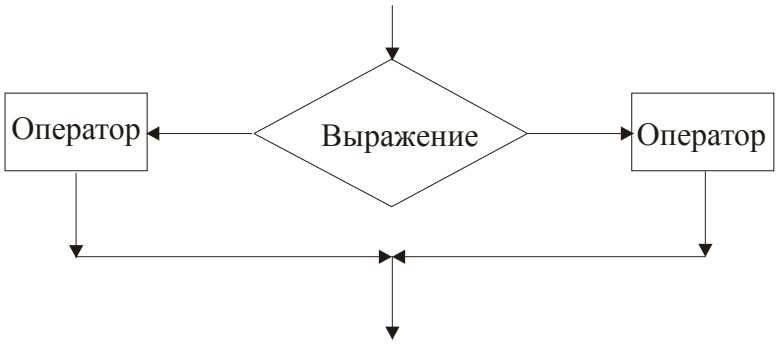


Рис. 7

При выполнении условного оператора номер 1, первым вычисляется выражение, стоящее в скобках после ключевого слова ***if.*** Если полученное значение выражения не равно нулю, то выполняется оператор (операторы), заключенный в фигурные скобки, стоящие непосредственно за закрывающейся круглой скобкой. Если выражение равно нулю, то оператор, следующий за закрывающей круглой скобкой не выполняется. Далее выполняется следующий за условием оператор.

Во втором случае, если выражение не равно нулю, то выполняется ***<оператор1>***; а оператор, следующий за ключевым словом «***else***», не выполняется. Если выражение равно нулю, то <оператор 1> не выполняется, а оператор, следующий за «else», выполняется.

Рассмотрим пример:

```
//пример 1
if (x==10) z=100;  //оператор z=100; будет выполняться если x равно 10

//пример 2
if (k) z=1;          //оператор z=1; если k будет не равно нулю

//пример 3
if (k&b) { m=10;        k=20; } //операторы будут выполнены если результат  k&b  будет не равен нулю
//пример 4
if ((c=getch()) == 'a')
  {
    puts ("введена буква a;");
    c='A';
  }
```

В этом примере первоначально производится ввод символа с клавиатуры (функция getch()). Присвоение кода введенного символа переменной ***c.*** И сравнение введенного кода символа с кодом символа ‘a’. Таким образом, если была нажата клавиша ‘a’, то будет выведено сообщение «***введена буква a;»*** и переменной ***c***

будет присвоен код символа ‘A’.

```
//пример 5
if (x == b)   z=d;   else   z=b;
```

Если значение ***x*** равно значению ***b***, то переменной ***z*** будет присвоено значение ***d***. В противном случае, переменной ***z*** будет присвоено значению ***b***.

//пример 6

```
if (x == a || x==b )  { m=a*b;   j++;}
                      else {  swap(a,b);    j--; }
```

В этом примере показано, что операторы можно группировать, используя фигурные скобки. Если значение переменной ***x*** равно значению переменных ***a*** или ***b***, то выполняются операторы в первом блоке. В противном случае, операторы в во втором блоке.

//пример 7

```
if (k)                                          if (k)  {
 if (m) x=1;              тоже самое            if (m) x=1;
else x=2;                                       else x=2;  }
```

В этом примере показано управление во вложенных операторах ***if.***

//пример 8

```
001          x=10;   m=2;   t=3;   k=3;
002          if (k == 2)
003            if (m == 2)    x=1;
004             else
005               if (t == 3)  x=4;
006             else;
007          else  x=5;
008          printf ("%d\n",x);
```

В этом примере будет напечатано число 5. Предложение ***else ;*** (строка 006) необходима для того, чтобы ***else*** в строке 007 относился к ***if*** строки 002.

//пример 9

```
    k=0;
if (k=10)  c=50;
  else c=40;
```

В этом примере записана часто встречающаяся ошибка у начинающих, когда вместо операции сравнения «==«, стоит операция присваивания «=». В этом случае условие проверяется в зависимости от результата присвоения, если результат присвоения не равен нулю, то выполняется оператор ***c=50***, иначе ***c=40.***

Можно дать общие рекомендации при написании вложенных условных операторов: если есть сомнения, ставьте фигурные скобки!

```
if (...) {
  if (...) ; else...
    }
else … }
```

2.10.3 Оператор *while*

Оператор ***while*** предназначен для организации цикла в программе. Под циклом в программировании понимают последовательность операторов, которая может выполняться некоторое количество раз. Блок-схема оператора while показана на рисунке: ***while*** (<выражение>) <оператор>

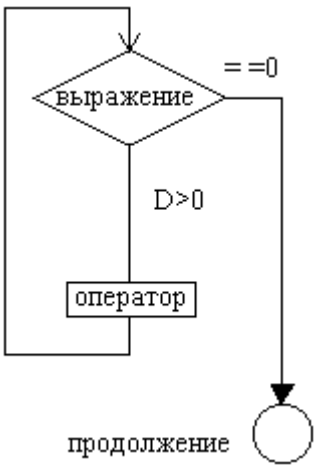


Рис. 8

Оператор ***while*** выполняется следующим образом: при входе в цикл вычисляется выражение, записанное в скобках; если оно не равно нулю, то выполняется оператор (или последовательность операторов, заключенные в фигурных скобках). Далее снова вычисляется выражение, стоящее в скобках. Оператор будет выполняться до тех пор, пока выражение не станет равным нулю.

Рассмотрим пример:

1) while (x>0) {s=s+x; x--;}

В этом пример производится накопление суммы до тех пор, пока x не станем меньше или равно нулю.

2) while (*s++=*p++);

В этом примере производится копирование последовательности байт(или слов) до тех пор, пока не встретится ноль. Это удобно при копировании строк символов.

3) while (1);

Бесконечный цикл.

4) i=0;
while (i<n) {s=s+v[i]; i++;}

В этом примере организуется цикл для вычисления суммы элементов массива.

5) while (mousestatus(&k,&y)= =1);

В этом примере в цикле будет вызываться функция ***mousestatus,*** пока она будет возвращать значение равное 1.

6) while (*ch= =' ') ch++;

В этом примере производится проверка строки символов, указатель наращивается, пока встретился пробел.

2.10.4 Оператор *for*

Оператор for предназначен для организации цикла. Правила записи этого оператора следующие:

for (<выражение 1>;<выражение 2>;<выражение 3>) <оператор А>

Выполнение оператора ***for*** начинается с вычисления <выражение 1>, затем вычисляется <выражение 2>. Если его значение равно нулю, то цикл завершается. При этом <оператор А> не выполняется. Если <выражение 2> не равно нулю, то выполняется <оператор А>, затем выполняется <выражение 3>, после этого снова вычисляет <выражение 2>. Если значение равно нулю, то цикл завершается; если нет, то цикл продолжается.

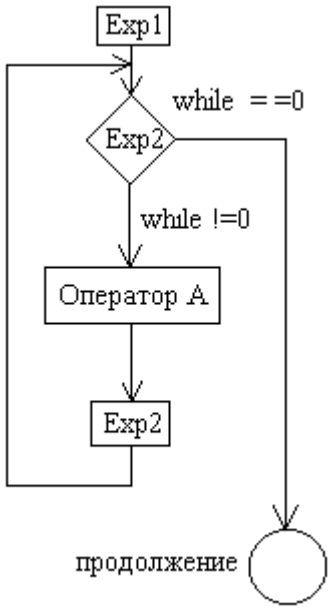


Рис. 9

Примеры:

1) for (;;);

В этом примере записан бесконечный цикл.

2) for (i=0; i<n; i++) s=s+v[i];

В этом примере записан цикл для вычисления суммы некоторого массива. Первоначально, i присваивается 0, далее производится проверка (i<n). Если она истинна, то цикл прекращается и управление переходит на первый оператор, записанный после цикла.

Если условие ложно, то выполняется оператор ***s=s+v[i];*** и далее i увеличивается на единицу.

3) for (i=n-1; i>=0; i--) s=s+v[i];

В этом примере находится та же сумма, только просмотр начинается с последнего элемента массива. Поскольку в Си элемент массива начинается с нуля, то последний имеет номер n-1.

4) for (i=0; s=0; i>0; s=s+v[i],i++);

Это одна из возможных записей для вычисления суммы массива.

5) ***for (t=head; t!=NULL; t=t->next) Show(t);***

Здесь записан пример организации цикла для просмотра некоторого списка.

2.10.5 Оператор *do while*

Оператор ***do while*** предназначен для организации цикла в программе. Правила записи оператора следующие:

do <оператор> ***while*** **<**выражение>);

При входе в цикл выполняется <оператор>. Далее производится вычисление выражения. Если значение выражения не равно нулю, то <оператор> выполняется снова; если выражение равно нулю, то цикл завершается. Выполнение цикла производится до тех пор, пока значение выражения не станет равным нулю.

В отличии от оператора ***while*** в круглых скобках сначала производится проверка, а потом выполнение оператора, в операторе ***do while*** сначала выполняется оператор, а потом производится проверка условия.

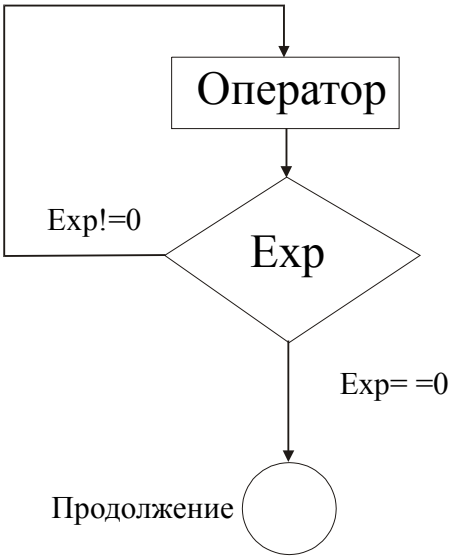


Рис. 10

2.10.6 Оператор *continue*

Оператор ***continue*** предназначен для завершения очередной итерации в операторе цикла. Например,

```
for(i=0; i<n; i++)
 {
  if(a[i]<0) continue;    //Завершить очередную итерацию
 s=s+a[i];  //этот оператор не будет выполнен если элемент
            //отрицателен
}
```

Этот фрагмент программы вычисляет суммы неотрицательных элементов массива.

2.10.7 Оператор *break*

Оператор ***break*** предназначен для выхода:

1) за пределы цикла;
2) для выхода из оператора switch.

При выполнении этого цикл прекращается и выполняется следующий оператор, записанный после цикла. Например,

```
for(i=0; i<n; i++)
 {
  if(a[i]<0) break;    //Завершить цикл если элемент отрицателен
 s=s+a[i];  //этот оператор не будет выполнен если элемент
```

```
        //отрицателен
}
```

Этот фрагмент программы вычисляет суммы неотрицательных элементов массива, до тех пор, пока не встретится отрицательный элемент.

2.10.8 Оператор switch

Оператор ***switch*** предназначен для организации множественного ветвления. Работу этого оператора наглядно представлена на рис. 11.

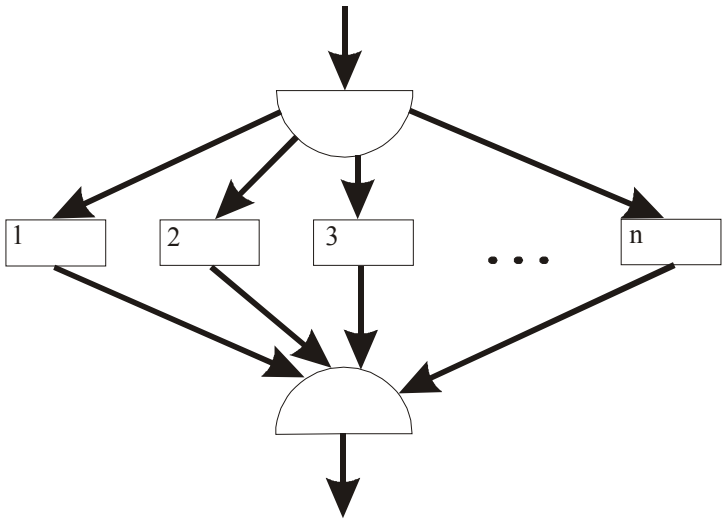


Рис. 11

Имеется входной блок, в котором проверяется условие ветвления и далее переходит на один из блоков записанных под номерами 1….n.

Формальная структура данного оператора следующая

```
switch(<условие>)
{
 case <константа 1>: <операторы>
    break;
 case <константа 2>: <операторы>
    break;
…
 defaut: <операторы>
}
```

Оператор break может отсутствовать, тогда будут выполняться операторы следующего блока case . Блок ***default*** необходим для случая, когда

нет перехода ни на один case-блок. Он также может отсутствовать. Рассмотрим примеры:

Пример 1.

```
switch(strcmp(str1,str2))
{
case 0:  puts("строки равны"); break;
case 1:  puts("str1>str2"); break;
case -1: puts("str1<str2"); break;
}
```

Пример 2

```
#define RETURN 13    //записать константы для спец клавиш
#define ESC         27
#define BAKCSPACE 9
int ch;
ch=getch();  //взять код клавиши
switch(ch)
{
 case RETURN:   puts("Перевод строки");
                break;
 case ESC:      puts("Нажата ESC");
                break;
case BACKSPACE:   puts("Нажата клавиша BS");
                break;

default:        puts("нажата буква или цифра или знак");
                break;
}
```

В этом примере показано, как можно организовать множественное ветвление с блоком ***default.*** В данном случае, коды специальных клавиш записаны как константы, код нажатой клавиши запоминается в переменной ***ch.*** Блок ***default*** нужен для выявления ситуации, когда нажата не специальная клавиша.

2.10.9 Оператор *return*

Оператор return предназначен для возврата управления и значения из функции. Правила записи следующие:

1) ***return;***

2) ***return <выражение>.***

В первом случае никакого значения не передается . Во втором случае передается значение выражения.

Например,

```
return 1;
return 3*b;
```

2.10.10 Метки и оператор *goto*

Любой оператор в функции может быть помечен меткой. Метка это некоторый идентификатор, стоящий перед оператором и разделенный точкой с запятой. Правила записи следующие:

<метка>: <оператор>

Например,

```
L1: x=10;
Metka: if(x>20) y=10; else y=20;
```

Метки определены только внутри одной функции.

Метки используют оператор ***goto***, который обеспечивает безусловный переход на указанную метку. Правила записи следующие:

goto <метка>;

Например,

```
for(i=0;i<n;i++)
for(j=0;j<m;j++)
{
  if(x[i][j]<0) goto Exit;
}
Exit: ShowX();
…
```

В этом примере показано организацию выхода сразу из нескольких циклов. Использование оператора ***break*** ограничено только одним циклом.

2.11 Указатели

2.11.1 Что такое указатель

Когда переменная объявляется в программе, то записывается ее имя и тип. При этом подразумевается, что переменная будет принимать значения из некоторого множества. На этапе выполнения программы

каждой переменной соответствует некоторый блок памяти. Этот блок имеет адрес (целое положительное число, обозначающее номер первого байта данного блока) и размер, который зависит от типа принимаемых значений. Например, для целой переменной выделяется 2 байта. Запишем следующее объявление переменной:

int k;

Рассматривая часть «int» оператора объявления компилятор будет знать, что для переменной «k» требуется два байта оперативной памяти для хранения целых значений. При трансляции само имя «k» будет записано в таблицу символов, а после трансляции данному имени будет сопоставлен адрес блока памяти, длина которого известна и соответствует заданному типу (для нашего примера это 2 байта). Позже, когда мы запишем

k=2;

мы ожидаем, что во время выполнения программы целое значение ***2*** будет записано в блок памяти, который выделен под переменную ***k***. Таким образом, ***k*** воспринимается как объект, с которым ассоциируется два значения: одно — это число, которое хранится в выделенном блоке памяти, и второе число это адрес данного блока памяти. В тех случаях, когда необходимо взять значение объекта ***k,*** то говорят ***«rvalue»*** (правое значение, это означает, что ***k*** стоит справа от знака присвоения «=»). В тех случаях, когда ***k*** хотим присвоить значение, то говорят о ***«lvalue»*** (левое значение, это означает, что k стоит слева от знака присвоения «=»).

Таким образом/Керниган и Ритчи /,

1) объект есть поименованная область памяти, имеющая адрес и размер;

2) ***rvalue*** объекта это значение, которое хранится в объекте;

3) ***lvalue*** объекта это выражение ссылающееся на объект (адрес объекта).

Приведем следующий текст фрагмента программы:

int k, j//1
k=2;//2
j=7;//3
k=j;//4

В строке 3 для переменной ***j*** берется ***lvalue***, и присваивается значение 7. В строке 4 для ***j*** берется ***rvalue*** (ее значение 7) и присваивается переменной k, для ***k*** берется ***lvalue*** (ее адрес, куда будет записано значение 7).

Теперь нам бы хотелось иметь переменную, значением которой был бы адрес некоторого объекта (иными словами ***lvalue*** некоторого объекта). Такие переменные стали переменная типа указатель или просто указатель. Указатель объявляется с помощью звездочки, например

int *ptr;

В этом примере ptr это имя переменной (такое как «k» в предыдущем примере); Символ «*» говорит о, что переменная ptr указатель , т.е. значение данной переменной есть адрес некоторого блока ОП. Ключевое слово int сообщает компилятору о том, что наша переменная содержит адрес блока памяти, в котором будет храниться целое число. При этом компилятор заранее знает, какой объем памяти будет занимать указатель (это зависит от процессора и системы программирования (см. модель памяти компилятора)). Таким образом, можно сказать, что указатель установлен на целое ***(int).*** Однако когда мы записываем ***int k;*** мы не присваивает значения переменной ***k***. Если это объявление сделано вне какой либо функции, то компилятор инициирует ее в ноль. В противном случае ***k*** может иметь любое значение.

В нашем примере переменная ***ptr*** не имеет значения, это означает, что указатель ***ptr*** не содержит адреса на блок памяти, где записано целое число. В случае, если указатель объявлен вне тела какой либо функции, ему присваивается специальное значение, которое называется ***null-указатель***. Это значение гарантирует, что указатель, имеющий данное значение не ссылается ни на какой либо объект или функцию.

В действительности реальное значение ***null*** не обязательно должно равняться нулю. Все зависит от специфики компьютера и операционной системы. Для того, чтобы сделать Си-программы совместимыми между разными типами компьютеров, ввели специальный макрос для представления ***null-указателя***. Этот макрос имеет имя ***NULL***. Таким образом, оператор присваивания ***ptr=NULL***; гарантирует, что указатель ***ptr*** примет ***null*** значение. Для проверки значения мы можем написать следующий условный оператор

if(ptr==NULL) {…}

Для присвоения значения переменной ***ptr***, т.е. присвоение адреса некоторого объекта, хранящего целое число, например адрес переменной ***k***, необходимо воспользоваться унарной операцией ***&***. Для нашего примера это выгляди следующим образом:

ptr=&k;

Операция ***&*** обеспечивает определение ***lvalue*** переменной ***k*** в случае, если ***k*** стоит справа от знака присвоения ***«=»***. После выполнения оператора присвоения говорят, что ***ptr*** указывает на объект ***k***.

Для того, чтобы присвоить значение объекту через указатель используют специальную операцию унарная звездочка ***«*»***. Рассмотрим следующий пример

****ptr=7;***

При выполнении этого оператора значение 7 будет записано в блок памяти адрес которого, записан в указателе. Таким образом, если ***ptr*** указывает на k (адрес переменной ***k*** хранится в ***ptr***), то переменной k будет присвоено значение 7. В этом примере операция «*» означает, что lvalue вычисляется с помощью указателя ***ptr***. Теперь, если мы запишем:

j=*ptr;

то переменной j будет присвоено значение 7. В данном случае *ptr означает rvalue, т.е. будет взято значение объекта, адрес которого хранится в указателе ptr. Ниже приведен простейший пример для пояснения идеи указателя.

```
#include <stdio.h>
int j,k;
int *ptr;
int main()
{
 j=1;
 k=2;
 ptr=&k;
 printf("\n ++++++ Пример №1 +++++++\n");
 printf("j имеет значение %d и хранится по адресу %p\n",j,(void *)&j);
 printf("k имеет значение %d и хранится по адресу %p\n",k,(void *)&k);
 printf("ptr имеет значение %p и хранится по адресу %p\n",ptr,(void *)&ptr);
 printf("значение целого на которое указывает ptr равно %d\n",*ptr);
 return 0;
}
```

2.11.2 Указатели на массивы

Указатели можно оргагнизовать не только на объекты, такие как ***char a*** или ***int c***, но и на массивы. Например это можно используя ключевое слово ***typedef***, которое предназначено для создание нового типа. Например,

typedef int Array[10];

При таком объявлении идентификатор Array становится новым типом данных.

Тогда можно записать:

Array my_arr;/* массив из 10 целых чисел */

или

Array arr2d[5];/* 5 массивов каждый из 10 целых чисел */

Можно также организовать указатель:

Array *p1d;

Тогда следующие присваивания будут корректны:

p1d=&my_arr;
p1d=&arr2d[0];
p1d=arr2d;

Важно также отметить, что ***sizeof(Array)*** будет равет ***10*sizeof(int).***

Поэтому если мы будем делать инкркмент (p1d+1), то адрес увеличится на ***10*sizeof(int)*** байт.

Можно объявить указатель на массив не используя ***typedef.*** Это достигается с помощью следующей синтаксической конструкции:

int (*aptr)[10]; /* объявление указателя на массив из 10 целых */

Не надо путать это объявление с похожим на него:

int *aptr[10];

В этом объявлении записан массив указателей типа ***int.***

2.11.3 Указатели и динамическое распределение памяти

Динамическое распределение памяти является эффективным средством программирования и используется тогда, когда размер объекта заранее вычислить нельзя или неэффективно. Например, размер матрицы, заранее не известен, а вычисляется в процессе выполнения программы.

Динамическое распределение памяти осуществляется с помощью специальных функций, на вход которых подается размер, требуемой памя-

ти, а на выходе адрес блока памяти. В некоторых системах программирования такие функции называются ***malloc()*** или ***calloc()***, прототипы которых описаны в файлах ***alloc.h*** или ***stdlib.h.*** Необходимо также отметить, что эти функции возвращают адрес типа void, поскольку они заранее не знают тип информации, который будет хранится в запрашиваемом блоке памяти.

Организация динамического распределения памяти в Си без указателей невозможна. Рассмотрим простой пример динамического распределения памяти под массив целых чисел:

```
int *iptr;
iptr=(int * )malloc(10*sizeof(int));
if(iptr==NULL) {   /* Обработка ошибки */ ; }
```

Если запрашиваемый блок памяти по какой либо причине не может быть выделен, то функция выделения памяти ***malloc()*** возвращает нулевое значение указателя ***NULL.*** Поэтому после вызова ***malloc()*** необходимо проверить значение указателя и если оно равно NULL. то записать код для обработки данного события (обычно, выводят сообщение о нехватки памяти и программа на этом завершается).

Если память выделена, то далее этот блок используют по назначению. В нашем примере можно предложить следующий код.

```
int k;
for(k=0; k<10; k++) iptr[k]=10-k;
```

Далее, если выделенный блок не нужен, то используя одну из специальных функций, с помощью которой возвращают ненужный блок памяти операционной системе. Одной из такой функцией является функция ***free()***. Аргументом этой функции является указатель, содержащий начальный адрес блока выделенной памяти.

Необходимо помнить, что при возврате всегда указывается начальный адрес блока выделенной памяти. Поэтому если необходимо использовать авто инкремент или что-нибудь подобное, то создают копию указателя, и далее используют эту копию. Например,

```
int *ptr_copy=iptr;
for(k=0; k<10; k++) *ptr_copy++=10-k;
```

Возврат выделенной пяти производится с помощью вызова функции ***free()***. Для нашего примера будет записано:

```
free(iptr);
```

```
#include <stdio.h>
#include <alloc.h>

//typedef int Ra[5];
//Ra *ptr;
int (*ptr)[5]; //указатель на массив из 5 элементов
main()
{
 int i,j,k=0;
 ptr=(int (*)[5])malloc(5*10*sizeof(int));
 for(i=0; i<10; i++)
 for(j=0; j<5; j++) ptr[i][j]=k++;

 for(i=0; i<10; i++)
 {
  printf("\n");
 for(j=0; j<5; j++)
  {
   printf("%d ",ptr[i][j]);
  }
 }
}
```

2.11.4 Указатели на структуры

Рассмотрим вопросы организации указателей на объекты — структуры. Предположим, что задана структура:

```
struct Student {
   char Name[20];
   char Group[20];
   char BirthDay[20];
   int YearsOld;
};
```

Предположим также что объявлен объект и указатель указанного типа:

```
 struct Student MyFriend={"Nik", "5566-1", "1 may", 17 };
 struct Student *ptr;
```

тогда

ptr=&MyFriend; //присвоить адрес объекта указателю
printf("%s",ptr->Name); //печать элемента структуры через указатель

здесь стрелка означает, что берется значение элемента структуры через адрес, записанный в ptr.

```
printf("%s",ptr-> BirthDay);
ptr-> YearsOld++;
```

Рассмотрим случай, когда в объекте типа структура имеется указатель.

```
 struct Vendor {     //описание производителя
   char Name[10];
   char Addres[20];
}
 struct Goods {  //описание товара
struct Vendor *ven;
       char Name[20];
char Price;
};

struct Vendor vendor={ "фабрика", "Луна"};
struct Goods Mouse, *ptr;
ptr=&Mouse;
ptr->ven=vendor;
strcpy(ptr->ven->Name,"Завод") ;
```

Если в структуре ***Goods*** переменную ***ven*** заменить на массив ***ven[20],*** то к соответствующим полям структуры ***Vendor*** можно сослаться следующим образом:

```
 printf( ptr->ven[k]->Name );
```

В структурах можно вводить рекурсивные описания, например для записи поля списка:

```
 struct Link {
   struct Link *next;   //указатель на следующий элемент списка
   int type;            //можно хранить тип данного (это зависит от
                        //программиста)
   void *ptr;           //указатель на данные, тип void можно
                        //преобразовать к требуемому
};
```

При объявлении

struct Link *item;

и если ему будет присвоено конкретное значение адреса начала списка такой структуры, то можно записывать:

item->next //адрес следующего элемента
item->next->next //адрес второго элемента
item->next->next->next //адрес третьего и т.д.

item->type; //тип данного элемента
item->next->type; //тип следующего элемента списка

item->ptr; //указатель на данные элемента списка
item->next->ptr; //указатель на данные следующего элемента

2.11.5 Указатели на функции

В Си можно организовать указатель на функцию. Правила записи здесь следующие:

<описание типы выхода> (*<имя>) (<список параметров>)

например,

void (*func)(int x, int y); //объявить указатель ***func***

Простейший пример, показывающий работоспособность следующий:

```
void (*f)(int i);                                   //указатель на функцию
void f1(int i) { printf("%d <<---\n", 2*I); }
void f2(int i) { printf("%d <<+++\n", 3*I); }

void main()
{
 f=f1;  //присвоить указателю адрес первой функции и вызвать ее
 f(10); //через указатель f с аргументом 10
 f=f2;
//присвоить указателю адрес первой функции и вызвать ее
 f(20); //через указатель f с аргументом 20
 }
```

Возможно использование указателя как параметр в функции, например

```
void ForEach(int massiv[],int n,  void (*func)(int el))
{
 int i;
 for(i=0;i<n; i++) func(massiv[i]);
}
```

В этом примере описан прием программирования, при котором организуется просмотр элементов заранее не определенной функцией, она в этом случае воспринимается как параметр, который передается извне, при вызове.

Например,

```
void Show1(int x) { printf("вариант печати №1 %d",x); }
void Show2(int x) { printf("вариант печати №2 %d",x); }
int y[5] ={ 1, 3, 7, 9, 11};

ForEach(y,5,Show1); //вывод массива по первому варианту
ForEach(y,5,Show2); //вывод массива по второму варианту
```

2.11.6 Указатели и константы

Указатель или объект, на который указывает указатель, могут быть объявлены как константы (это делается с помощью модификатора const). Рассмотрим несколько примеров

```
int i;                          //i – целая переменная
int *pi;                        //неинициализированный указатель на int
int * const cp=&i;              //указатель константа на переменную i
const int ci=7;                 //целая константа
const int *pci;                 //не инициализированный указатель на
                                // целую  константу
const int * const cpc=&ci; //константа указатель на целую
                                // константу
```

Следующие присвоения являются правильными

```
i=ci;       //целой переменной присвоена значение константы
*cp=ci;     //присваивается значение константы по указателю-
            //константе обычному объекту i (см.выше)
++pci;      //инкремент указателя на константу
```

pci=cpc; //присвоение указателю на константу значение
//константы указателя на константу объект

Следующие присвоения являются неправильными

ci=0; //нельзя присваивать значение объекту-константе
ci++; //нельзя изменять значение объекта-константы
**pci=3;* //нельзя изменять значение объекта-константы через
//указатель
cp=&ci; //нельзя изменять адрес указателя-константы
cpc++; //нельзя изменять значение указателя-константы

2.12 Программы и подпрограммы

Мы уже дали определение алгоритма как последовательности действий. Теперь сосредоточим внимание возможности группирования действий, т.е. объединение действий как единого укрупненного действия. Если теперь рассмотреть алгоритм с позиций компьютера, это последовательность действий представлена в виде последовательности команд. Объединение последовательности команд и по именование этой последовательности привело к понятию подпрограммы. Понятие «подпрограммы» расширяет понятие «команды». Можно сказать, что подпрограмма это команда созданная пользователем-программистом. Обычно подпрограмму специальным образом оформляют: обычно вначале записывают имя подпрограммы, а в конце команду возврата из подпрограммы. Чтобы воспользоваться подпрограммой, необходимо вызвать ее. Это делается с обычно помощью команды call.

Механизм вызова следующий:

1) адрес следующей команды за call запоминается в системном стеке;

2) в PC записывается адрес подпрограммы, который хранится в call и она начинает выполняться;

3) после выполнения подпрограммы должна быть выполнена команда return, которая берет из стека адрес возврата и записывает его в PC.

Поскольку call обычная команда процессора, то вызвать подпрограмму можно из любой другой подпрограммы. Особый интерес — это вызов подпрограммы в теле самой подпрограммы. Так называемая, рекурсивная организация подпрограмм.

Рассмотрим важный вопрос для использования подпрограмм это обмен информацией между программой и подпрограммой. Обмен информацией предполагает:

1) передачу информации в подпрограмму;

2) передача информации из подпрограммы в вызывающую программу.

Рассмотрим способы передачи информации в подпрограмму.

1. Использование регистров — наиболее быстрый способ организации передачи информации в подпрограмму. В этом случае, при написании подпрограммы учитывается, что необходимая информация перед вызовом подпрограммы заносится в известные заранее регистры. Например, если подпрограмма выводит символ на экран с повторением, то заранее оговаривают, что регистр r1 содержит код символа, а регистр r2 количество повторений. В этом случае регистры r1 и r2, говорят, содержат значения параметров.

2. Использование общей памяти. В данном случае, выделяется определенный размер памяти, доступ к которой имеет как вызываемая программы, так и подпрограмма. Обычно это некоторая статическая память, управление которой не зависит от подпрограммы. В данном случае, вызывающая программа заносит значения в статическую память, а подпрограмма берет эти значения и производит вычисления.

3. Использование стековой памяти. Этот механизм в настоящее время является общепринятым и заключается в следующем: перед вызовом подпрограммы, значения параметров заносятся в системный стек, в подпрограмме значения параметров выбираются относительно вершины стека, поэтому один и тот же код может работать с различными участками памяти, тем самым возможен рекурсивный вызов подпрограмм.

Рассмотрим варианты передачи информации из подпрограммы в вызывающую программу.

1. Использование регистров. В этом случае заранее известно, что такой то регистр после выполнения подпрограммы содержит искомое значение.

2. Использование общей памяти, в данном случае подпрограмма заносит вычисляемые значения в общую память.

3. Использование стека. Существует механизм передачи информации через системный стек.

Существует еще один важный вопрос передачи параметров: передача параметров по значению и передача параметров по адресу. В первом случае передается само значение, во втором случае вместо значения передается адрес памяти, где хранится значение. В первом случае, подпрограмма будет работать в своей локальной памяти. Во втором случае подпрограмма получает доступ к памяти в вызывающей программе.

2.12.1 Функции в Си

Рассмотрим организацию подпрограмм на Си. Все подпрограммы на языке Си оформляются как функции. Например,

```
int                 //описание возвращаемого значения
 func(              //имя функции
     int i,         //описание первого параметра
     int j          //описание второго параметра
      )             //
{                   //тело функции
 return i%j;        // оператор возврата
 }                  //конец тела функции
```

Для функции, которая не возвращает значения необходимо записать описатель void.

Например,

```
void                //функция ничего не возвращает
 Show(              //имя функции
     int i,         //описание первого параметра
     int j          //описание второго параметра
      )             //
{                   //тело функции
 printf("%d
%d\n",i,j);        // вывод значения i, j
 }                  //конец тела функции, оператор return может
                    отсутствовать
```

Оператор ***return*** может быть в любом месте подпрограммы.

Вызов подпрограммы записывается следующим образом: записывается имя функции и в скобках список выражений для параметров функции, если они есть. Например,

1) func(10,5);
2) Show(i+1,k++);

В тех случаях, когда функция возвращает значение, то вызов функции можно записывать в выражении. Например,

```
x=func(2,3)+20;
```

Список параметров в функции может отсутствовать.

Рассмотрим вопросы обмена информацией между программной и подпрограммой.

Возможны следующие варианты:

1. Использование параметров.
2. Использование общей памяти.

Рассмотрим передачу информации через параметры. Заранее известно, что значения параметров в функцию передаются по значению. Это означает, что значение переменной или выражения переписывается в локальную память функции (причем локальная память выделяется из системного стека). Поэтому значения переменных, подставленные вместо параметров, не изменяются.

Например,

```
void f(int i) { i=4; }
...
int j=5;
f(j);
```

В этом примере значение j не изменится при вызове функции f.

Для того чтобы значение переменной изменилось необходимо передать адрес переменной, это можно сделать используя указатели. Например,

```
void v(int *p) { *p=4; }
...
k=5;
v(&k);
```

В этом примере значение переменной k изменится, поскольку было передано не значение переменной, а ее адрес.

Часто такой подход используют при работе со структурами.

```
typedef struct A { type1 a1;  type2 a2; ... } ASTRUCT;

ASTRUCT x, y;
FuncA(ASTRUCT *s) { s->a1=0; s->a2=0; .... }
...
FuncA(&x);
FuncA(&y);
...
```

Рассмотрим вопросы организации передачи информации через внешние переменные!

Как это организовать?

.... ***prog1.c*** //первый модуль
int x; //это статическая переменная объявленная вне функции
void func1() { x=100; }
void func2(int i) { x=i; }
...
.... prog2.c //второй модуль
extern int x;
void Show() { printf("%d\n",x); }
void func3() { return x; }

В этом примере записано два файла ***prog1.c*** и ***prog2.c***, которые имеют внешнюю память x и функции, которые читают и пишут значения в эту внешнюю память. В первом файле объявляется статическая переменная x, во втором объявляется, что переменная x описывается как переменная, которая объявлена вне данного файла.

Есть еще одна важная деталь: это порядок занесения значений параметров стек при вызове функции. Существует несколько вариантов:

1) занесение с конца списка параметров (***cdecl***);
2) занесение с начала списка параметров (***pascal***);

Функия может иметь переменное число параметров. Для записи функции с переменным число параметров необходимо использовать лексемму … (три точки). Например,

void printf(char *format,…);

Три точки стояшие вместо описания параметра указывают, что функция имеет переменное число параметров. Например,

```
int sum_vect( int n,...){
 int *vec=&n+1;
 int sum=0;
 int i;
 for(i=0; i<n; i++) sum+=vec[i];
 return sum;
}
```

Эта функция находит суммы чисел, передаваемых в качестве аргументов. Первым аргументом является количесво чисел, передаваемых в качестве аргументов.

Например,

```
int s=sum_vect(5,20,10,15,1,2); //s=20+10+15+1+2
int count=sum_vect(3,i,j,k); //count=i+j+k
```

В систем каталоге **INCLUDE** имеется заголовочный файл ***stdarg.h***, которй содержит описания макросов для выделения аргумента заданного типа из списка аргументов:

```
void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

va_list — это тип указателя на список аргументов;
ap — указатель на список аргуметов;
lastfix — это имя параметра стоящего перед … (тремя точками);
va_start — инициализация указателя на список аргуметов;
type — тип аргумента;
va_arg — выделение значения очередного значения аргумента, при этом укаазталь ***ap*** автоматически перемещается на следующий;
va_end — макрос завершения

```
#include <stdio.h>
#include <stdarg.h>

//вычислить сумму последовательности целых чисел,
заканчивающуюся на 0
void sum(char *msg, ...)
{
  int total = 0;
  va_list ap;
  int arg;
  va_start(ap, msg);
  while ((arg = va_arg(ap,int)) != 0) {
    total += arg;
  }
  printf(msg, total);
  va_end(ap);
}

int main(void) {
  sum("The total of 1+2+3+4 is %d\n", 1,2,3,4,0);
```

```
  return 0;
}


//найти суммы разнотипных чисел, используя строку формата
#include <stdio.h>
#include <stdarg.h>

double sum2(char *format, ...)
{
  double total = 0;
  va_list ap;
  int arg;
  va_start(ap, format);
  while(*format){
   if(*format=='i') total+=va_arg(ap,int);
  else
   if(*format=='d') total+=va_arg(ap,double);
   format++;
  }
  va_end(ap);
  return total;
}
int main(void) {
  printf("%f",sum2("iddi", 1,2.77,3.5,4));
  return 0;
}
```

2.13 Препроцессор

Препроцессор — специальная программа входящая в состав компилятора Си, которая производит различные преобразования исходного текста программы. Это следующие преобразования:

1) включение текста из других файлов, обычно это заголовочные файлы, хотя может быть любой текстовый файл.

2) замена одних последовательностей символов на другие, причем возможна параметризация получаемых последовательностей.

3) условная трансляция.

4) установка состояний компилятора для различных вариантов компиляции

Важно отметить, что препроцессор не знает языка Си. Просто на входе текст на выходе преобразованный текст.

Препроцессор управляется специальным набором директив (операторов), который должны начинаться с символа #. Рассмотрим основные директивы :

2.13.1 Директива *#include*

Директива #include означает включить текст и файла, имя которого записано в директиве. Имеются следующие формы записи директивы:

1) ***#include*** <имя файла>

2) ***#include*** "имя файла"

Первый вариант предназначен для включения файлов из системных каталогов компилятора. Например,

#include <stdio.h> //включить заголовочный файл для использования функций стандартного ввода/вывода

Второй вариант предназначен для включения файлов из текущего каталога пользователя, обычно это заголовочный файл прикладного программиста Например,

#include "my_header.h"

2.13.2 Директива #define

Директива ***#define*** определяет макрос, который обеспечивает механизм подстановки лексем без или со списком формальных параметров, подобно функциям. Рассмотрим макросы без параметров. Запишем ряд примеров:

```
#define HI "Какой прекрасный день!"
#define SIZE 100
#define empty

printf(HI); //подстановка в результате будет printf("Какой прекрасный день!");
for(i=0; i<SIZE; i++) //вместо SIZE будет подставлено 100
printf("empty"); //в данном случае подстановки не будет, т.к. это строка символов
```

Рассмотрим макроопределения со списком параметров. Рассмотрим на примере:

```
#define CUBE  ((x) (x)*(x)*(x))
...
```

int n,y;
n=CUBE(y);

после подстановки препроцессор сгенерирует следующий текст:

n=((y)*(y)*(y));

Скобки при написании макроса очень важны, поскольку если бы не было скобок, то можно получить такой эффект:

n=CUBE(y+1);

тогда после подстановки будет получен следующий текст:

n=y+1*y+1*y+1; //результат совершенно другой 3y+1

Необходимо отметить следующие моменты использования макроопределений:

1. Вложение скобок и команд

Например,

#define ERRMSG(x,str) Show("Ошибка",x,str)
#define SUM(x,y) ((x)+(y))
...
ERRMSG(2,"Неверный ключ"); //замещение будет нормальное
//Show("Ошибка",2, ,"Неверный ключ");
return SUM(f(i,j),g(k,l)); //замещение нормальное
//return ((f(i,j))+(g(k,l)));

2. Соединение лексем с помощью ##.

Например,

#***define VAR(i,j) (i##j)***
....
VAR(u,10) //в результате замены даст u10

3. Сторонние эффекты.

При использовании макросов необходимо соблюдать осторожность. Например, если взять макрос CUBE описанный выше то может быть следующий эффект:

a=*3;*
n=CUBE(a++);
printf("%d\n",n);

В результате подстановки будет:

n=((a++)*(a++)*(a++));

после выполнения n будет 27 a=6 .

2.13.2 Условная компиляция

Условная компиляция предназначена для генерации конкретного варианта текста программы и создается с помощью специальных директив:

#if, #ifdef, ***ifndef, #e***lse,
#endif

Рассмотрим некоторые типичные примеры:

#ifndef MyHeaderH
#define MyHeaderH
....
....
#endif

Это типичный пример создания заголовочного файла пользователя. Здесь условная компиляция позволяет не вставлять несколько раз один и тот же заголовочный файл.

Первая строка ***(#ifndef***) проверяет не определен ли макрос ***MyHeaderH,*** если нет то определяет его и вставляет текст файла. Если макрос ***MyHeaderH*** определен , то весь текст пропускается до ***#endif***, тем самым он не вставляется.

Другой пример,

#ifdef DOS
....//программный код для OS DOS
...
#elif define(WINDOWS)
.... .//программный код для OS WINDOWS
....
#elif define(UNIX)
.... .//программный код для OS UNIX

....
#else
.... .//программный код для других OS
....
#endif

В этом примере показано, как создавать программный код для разного класса операционных систем. Если при трансляции установлен макрос DOS, то будет вставлен программный код следующий за #ifdef. Если установлен макрос WINDOWS, то будет вставлен код только для WINDOWS и т.д.

2.3 Механизм реализации языков программирования

При более глубоком изучении программирования необходимо освоить механизм реализации операторов языка программирования и связь его с низкоуровневым программированием (использование языка ассемблера).

Рассмотрим механизм реализации выражений. Например, нам необходимо вычислить выражение: ***x=sin(20+a)*(y-v);***

Один из многочисленных вариантов может выглядеть так:

```
mov r1,a      ;загрузить в регистр R1 значение ячейки a
mov r2,20     ;загрузить в регистр R2 значение константы 20
add  r1,r2    ;сложить R1 и R2 результат поместить в R1
push r1       ;занести в стек значение R1 это будет параметр для
              функции sin
call sin      ;вызвать функцию sin
pull  r1      ;забрать из стека результат вычисления синуса
mov r2,y      ;загрузить в регистр R2 значение ячейки y
mov r3,v      ;загрузить в регистр R3 значение ячейки v
sub r3,r2     ;произвести вычитание, результат поместить в R3
mult r1,r3    ;выполнить умножение, результат поместить в R1
mov x,r1      ;запомнить содержимое R1 в ячейку x
stop          ;останов
```

Реализация условных операторов. Рассмотрим пример

if(k==10) x=20; else z=z+30;

Первоначально вычисляется условное выражение и если результат не равен нулю, то выполняется оператор x=20; В противном случае оператор x=x+20; Тогда программ на ассемблере приблизительно выглядит так:

```
        mov r1,k      ;загрузить в регистр R1 значение ячейки k
        mov r2,10     ;загрузить в регистр R2 значение 10
        cmp r1,r2     ;сравнить содержимое регистров и если равны то
                      установить ;флаг Z в единицу
        jne met1      ;переход на метку если Z равен нулю
        mov r1,20     ; загрузить в регистр R1 значение 20
        mov x,r1      ;запомнить содержимое R1 в ячейку x
        bra  next     ;безусловный переход на продолжение
met1    mov r1,z      ; загрузить в регистр R1 значение ячейки z
        mov r2,30     ; загрузить в регистр R2 значение 30
        add r1,r2     ;произвести сложение, результат запомнить в R1
        mov z,r1      ;содержимое R1 запомнить в ячейку z
next    ....          ;
```

Механизм реализации цикла while. Рассмотрим это механизм на примере:

```
s=0;
while(i<10) {  i++;  s=s+i;  }
```

```
        clr r3        ; очистить r3 , это будет сумма (s)
        clr r1        ;очистить r1, это будет счетчик (i)
        mov r2,10     ;  загрузить в регистр R2 значение 10
begin   cmp r1,r2     ;сравнить r1 и r2
        bge next      ;если значение r1 больше равно, то перейти на
                      метку next
        inc r1        ;увеличить значение r1 на единицу
        add r3,r1     ;выполнить сложение s=s+i
        bra begin     ;перейти по метке begin (в начало цикла)
next    mov i,r1      ;выход из цикла, запомнить значение счетчика
        mov s,r3      ;запомнить значение суммы
```

Рассмотрим механизм реализации оператора switch на следующем примере:

```
switch(ch)
{
case ‘a’: x=1; break;
case ‘b’: y=2; break;
```

```
case ‘c’: z=3; break;
default: m=10;
}
```

Тогда ассемблерный код будет следующий:

```
        mov r1.ch     ;загрузить в r1 значение ch
        cmp r1,’a’    ;сравнить с кодом символа ‘a’
        bne met1      ;если это не символ ‘a’, то смотреть следующий
                      символ
        mov r2,1      ;иначе x присвоить 1
        mov x,r2      ;
        bra next      ;выход на конец оператора switch
met1    cmp r1,’b’    ;сравнить с кодом символа ‘b’
        bne met2      ;если это не символ ‘b’, то смотреть следующий
                      символ
        mov r2,3      ;иначе y присвоить 2
        mov y,r2      ;
        bra next      ; выход на конец оператора switch
met2    cmp r1,’c’    ;сравнить с кодом символа ‘c’
        bne met3      ;если это не символ ‘c’, то смотреть следующий
                      символ
        mov r2,3      ;иначе z присвоить 3
        mov z,r2      ;
        bra next      ; выход на конец оператора switch
met3    mov r2,10     ;во всех остальных случаях
        mov m,r2      ;ячейке m присвоить значение 10.
next    ....          ;продолжение
```

2.3.1 Передача параметров

Механизм передачи параметров в Си основан на использовании системного стека:

Необходимо напомнить, что в процессоре есть специальный регистр называемый регистром стека SP (stack pointer) и две специальных команды, push — положить слово в стек и pull — взять слово из стека.

Рассмотрим пример передачи параметров. Пусть задана следующая функция:

```
int Add(int a, int b)
{
  return a+b;
}
```

и где-то в программе есть следующий код

... x=Add(i,j) ....

Тогда ассемблерный код будет следующий

```
_Add    mov bp,sp     ; взять адрес вершины стека и поместить в
                        индексный регистр
        mov           ;взять значение первого параметра
        r1,[bp+4]
        mov           ;взять значение второго параметра
        r2,[bp+6]
        add r1,r2     ;сложить
        return        ;вернуть управление и значение
        ....
        push j        ;положить в стек значение последнего параметра в
                        списке
        push i        ;положить в стек значение первого параметра
        call _Add     вызвать функцию _Add
        mov x,r1      присвоить значение суммы ячейке x
        pull r1       восстанавливаем стек
        pull r1
```

Важно помнить, что в Си вычисление значений параметров и их помещение в системный стек производится с конца списка, а для функций и процедур написанных на паскале производится с начала списка. Например,

```
#include <stdio.h>

int TYPE   Test(int a, int b)
{
  printf("%d %d\n",a,b);
  asm { bbb };
  return 1;
}
int main()
{
   int i=10;
   Test(i++, i++);
    return 0;
}
```

Если слово TYPE заменить на cdecl, то параметры будут передаваться по правилам Си и после выполнения программы на экран будет выдано 11 10. Если слово TYPE заменить на pascal, то программ выдаст экран 10 11.

2.3.2 Механизм выделения локальной памяти под переменные

Известно, что память под локальные переменные выделяется из системного стека.

Рассмотрим сам механизм выделения на следующем примере:

```
int Func(int i, int k)
{
  int l, m;
  l=10;
  m=i+k;

  ...
}
```

Тогда ассемблерный код будет таким:

```
_Func   mov bp,sp       ; взять адрес вершины стека и поместить
                        в индексный регистр
        push r1         ;сдвинуть указатель стека на слово
                        (память под l)
        push r1         ;сдвинуть указатель стека на слово
                        (память под m)
        mov r1,10       ;загрузить в r1 значение 10
        mov [bp-2],r1   ;записать адресу первого параметра
        mov r1,[bp+4]   ;загрузить в r1 значение параметра i
        mov r2,[bp+6]   ;записать адресу второго параметра
        add r1,r2       ;сложить
        mov [bp-4],r1   ;занести результат по адресу ячейки m
        ...
        pull r1         ;вернуть память взятую под локальные
                        переменные
        pull r1
        return          ;вернуть управление и значение
```

Таким образом, память под параметры выделяется из стека непосредственно перед вызовом, в момент вызова в стек заносится адрес

возврата, затем выделяется память под локальные переменные. При входе в функцию указатель стека указывает на слово, содержащее адрес возврата. Также известно, что системный стек растет от старших адресов к младшим, то параметры будут иметь смещение с плюсом, а локальные переменные с минусом.

2.3.3 Использование ассемблерных вставок

В различных версиях реализаций Си-компиляторов предусматривается возможность записи ассемблерных вставок. В некоторых случаях это является эффективным средством программирования. Особенно это касается различных оптимизаций программы. Самые трудные участки программы переписываются на ассемблере, таким образом можно повысить эффективность вычислений. Это делается с помощью оператора asm. Например,

```
int i;
i=20;
asm { mov r1,80000
      mov (r1),10
}
....
```

Конкретные правила записи ассемблерного кода в Си программе зависит от компилятора и вычислительной машины.

Еще одно отличие от ассемблеров — в операторе **asm** при описании команды вместо операндов можно записывать имена глобальных и локальных переменных.

2.3.4 Использование регистров

Переменные в Си можно объявить как переменные регистры, это означает, что для хранения значений переменной выделяется регистр процессора, а не оперативная память.

Поскольку регистров значительно меньше, чем слов в ОП, то возможно, что данной переменной не будет выделен регистр. Т.е. описание переменной как регистровой не гарантирует, что под переменную будет выделен регистр. Обычно для указания, что переменная регистровая требуется ключевое слово register. Например,

register **a, b;** //выделить переменным a м b регистры процессора.

В некоторых случаях в реализации компилятора резервируют специальные ключевые слова для обозначения регистров данного процессора. Например,

_AX, _BX, _CX в некоторых версиях компиляторов фирмы Borland.

Использование регистров в практике программирования считается дурным тоном, поскольку делает программу существенно машиннозависимой. Тем не менее , в случаях написания системных программ это допускается.

2.3.5 Связь с операционной системой

Современные компьютеры оснащены мощным программным обеспечением, обеспечивающим эффективную производительность и управление вычислительным процессом. Это означает, что когда программист создает свою программу, он должен учитывать, что его программа будет только частью более сложной программы. Обычно говорят, что прикладная программа (программа пользователя) в процессе выполнения взаимодействует с операционной системой. Операционная система это совокупность программ, создающая сервис по вводу и выводу информации на конкретные устройства, а также управление этими устройствами. Кроме того, она обеспечивает поддержку файловой системы и управление задачами. Более подробно описание операционных систем можно найти в работах / /. Здесь важно отметить, что прикладная программа сама не вводит и не выводит информацию, а использует возможности операционной системы(ОС). Обычно та часть ОС, которая отвечает за ввод/вывод информации, называют BIOS (Basic Input Output System).

Механизм взаимодействия с BIOS основан на так называемых программных прерываниях. Мы уже вскользь говорили о прерываниях (см. раздел «Модель вычислений»). Однако не рассматривали программных прерываний. Ввод/вывод обычно организован асинхронно., т.е. если устройство вводит или выводит информацию, то это делается на основе аппаратного прерывания, с которым связана некоторая программа, называемая драйвером устройства. Этот драйвер обычно пишет или читает информацию в специально выделенном участке ОП, называемый буфером. Программное прерывание это приказ BIOS прочитать или писать информацию в этот буфер. Рассмотрим пример, клавиатура — устройство посимвольного ввода информации, для нее существует специальное прерывание и драйвер. Когда пользователь нажимает клавишу, происходит аппаратное прерывание, текущая программа прерывается и выполняется

драйвер клавиатуры, который читает код нажатой клавиши из регистра данных устройства и записывает этот код в выделенный буфер, который называется кольцевым. После того как код символа помещен в кольцевой буфер, возобновляется выполнение прерванной программы. Каким образом прикладная программа может прочитать код нажатой клавиши, как раз это и делает программное прерывание. Т.е. прикладная программа дает приказ операционной системе прочитать код нажатой клавиши из кольцевого буфера и вернуть этот код прикладной программе.

Каким образом записать вызов программного прерывания? Это можно сделать. используя ассемблерную вставку. Но в настоящее время для соответствующего компилятора строятся системные библиотеки и где практически все программные прерывания записаны в виде функций.

Прототипы этих функций записываются в заголовочных файлах, поставляемых вместе компилятором. А сами функции находятся в системных библиотеках

2.3.6 Описание наиболее часто используемых функций из системных библиотек

Функция printf(char *format, ...) — это универсальная функция вывода информации на экран терминала. Первый параметр у этой функции — строка символов, описывающая какие и как выводить переменные записанные за строкой формата. Описание этой функции подробно дано в описании конкрктной системы. Здесь мы рассмотрим несколько примеров:

```
int i=10, j=30;
printf("i=%d j=%d\n",i,j);
```

В данном примере функция printf() выводит значения переменных i и j. Причем вывод будет произведен согласно строке формата: первоначально будет выведено «i=«, затем будет выведено значение переменной i (%d означает, что надо взять значение переменной из списка переменных и вывести это значение в десятичном виде) , затем будет выведено « j=« и значение переменной в целом формате. И последние символы «\n» в строке формата означают, что необходимо произвести перевод строки и возврат каретки. В результате вызова функции printf из вышестоящего примера на экране появится строка символов: i=10 j=30

Перечислим некоторые описатели формата вывода:

%s — вывод строки символов;

%c — вывод одиночного символа;

%f — вывод вещественного числа;

Например,

```
 char str[20]="привет";
 int i=20;
double t=5.234;
 printf("%f ---- %s ---- %d \n",t,str,i);
```

на экран будет выведено 5.234 ---- привет --- 20

Имеется функция sprintf, которая аналогична printf, только вывод осуществляет в специальный буфер. Например,

```
char buf[30];
int Sum=21;
sprintf("Ваше число %d",Sum);
```

В результате строка buf примет значение «Ваше число 21».

Для того чтобы использовать эти функции необходимо подключить системный заголовочный файл stdio.h.

Функция puts(char *s) выводит строку символов на экран терминала.

Функция gets(char *s) вводит значение в строку s.

Функции для работы со строками:

int strlen(char *s) — определяет длину стоки;

char *strcpy(char *d,char *s) — копирует строку s в строку d

int strcmp(char *s, char *t) производит лексикографическое сравнение двух строк, при этом результат:

-1 s меньше t;
0 s равно t;
1 s больше t.

strcat(char *dst,char *src) — это функция производит операцию конкатенации двух строк, результат помещает в dst (проще говоря строка src дописывается в конец dst).

Для того чтобы использовать эти функции необходимо подключить системный заголовочный файл string.h.

Функции преобразования

int atoi(char *s) — преобразование строки символов в целое число.

float atof(char *s) — преобразование строки символов в вещественное число.

Функции для работы с файлами

Файловая система это часть операционной системы, предназначенная для долговременного хранения данных и программ. Файловая система обеспечивает организацию каталогов и файлов.

Каталог это поименованный участок внешней памяти, в котором хранятся файлы и каталоги. Имеется начальный каталог или корневой каталог, от которого начинается разветвление. Для указания местонахождения файла в файловой системе используется путь, перечисление всех вложенных каталогов, начиная с корневого.

Файл это поименованный участок внешней оперативной памяти, в котором хранятся данные и программы. Файл имеет следующие основные атрибуты: имя, расширение, время и дату создания, размер, информация о доступности и защищенности и прочую служебную информацию. Для работы с файлом необходимо:

1) Открыть файл. Эта операция предназначена для создания специальных структур и механизмов для ускоренной работы с файлами, в частности создание специальных буферов.

2) Выполнить чтение или запись. Операция чтения производит чтение информации из заданного места файла в некоторый участок оперативной памяти. Операция записи производит запись информации из оперативной памяти в файл. Существует два варианта: запись порции информации в конец файла и перезапись порции информации в уже созданном файле.

3) Закрыть файл. Эту операцию необходимо обязательно выполнить, если этого сделано не будет, то файл будет потерян.

Есть несколько уровней программного обеспечения для работы с файлами:

1) верхний уровень описан в заголовочном файле **stdio.h**

2) средний уровень описан в заголовочном файле **io.h**

3) нижний уровень пишется на уровне вызова программных прерываний dos.

Рассмотрим функции для работы с файлами верхнего уровня.

FILE *fopen(char *filename, char *mode) **—** открыть файл, необходимо указать имя файла и моду.

Мода может иметь значения:

r Открыть для только для чтения

w Создать для записи, если файл существует то для перезаписи

a добавить , писать в конец файла если он существует, открыть для записи если не существует

r+ Открыть файл для чтения и перезаписи

w+ Создать новый файл для чтения и перезаписи, если файл с таким именем существует, то фал будут заменен на новый

a+ Открыть файл на чтение запись и перезапись

Кроме того, файл можно открыть в текстовой(t) и двоичных(b) модах. Например

"rt" — открыть файл на чтение в текстовой моде.

"wb" — открыть файл на запись в двоичной моде.

Также возможно записывать

"r+t" ,"rt+", "ab+" и т.д.

FILE — это описание специальной структуры, для хранения информации об открытом файле (такая структуру обычно называется, блок управления файлом)

При успешном выполнении операции открытия функция ***fopen*** возвращает адрес структуры типа ***FILE.*** Если открытие файла не состоялось, то функция возвращает ***NULL***.

Функция чтения данных из файла

size_t fread(void *ptr,size t size, size_t n, FILE *fp);

Функция ***fread*** читает ***n*** элементов данных, каждый длинной ***size*** байт, из открытого файла ***fp***, в блок оперативной памяти, адрес которого записан в ***ptr***.

Функция записи данных в файла

size_t fwrite(void *ptr,size t size, size_t n, FILE *fp);

Функция ***fwrite*** записывает ***n*** элементов данных, каждый длинной ***size*** байт, в открытый файла ***fp***, из блока оперативной памяти, адрес которого записан в ***ptr***.

Функция перемещения указателя файла.

int fseek(FILE *fp, long offset, int whence);

Эта функция устанавливает значение указателя файла. Это указатель предназначен для указания места в файле, где необходимо производить операции чтения записи и перезаписи. Первоначально указатель устанавливается на начала файла. Параметр ***offset*** устанавливает смещение относительно места отсчета (начало файла, текущее положение указателя,

конец файла) которое записано в параметре ***whence***. Эта переменная имеет три значения 0, 1, 2, который представлены как символические константы:

whence		Размещение в файле
SEEK_SET	(0)	Начало файла
SEEK_CUR	(1)	Текущее положение курсора
SEEK_END	(2)	Конец файла

Функция fprintf

int fprintf(FILE *fp,char *format, ...)

Эта функция аналогична функции ***printf***, описанной выше, за исключением того, что она записывает данные не на экран, а в открытый файл ***fp.***

2.4 Технология создания исполняемой программы

Современные технологии создания программ основаны на использовании интегрированной системы программирования. Обычно такая система содержит:

1) текстовый редактор для ввода текста программы;

2) компилятор, который обеспечивает перевод текстового описания программы в специальный объектный код;

3) редактор связей (компоновщик), обеспечивающий сборку выполняемой программы.

4) встроенный отладчик, позволяющий пошаговое выполнение программы с возможностью отслеживать различные классы информации.

5) развитые средства помощи;

6) вспомогательные средства и средства управления проектом.

2.4.1 Текстовый редактор

Текстовый редактор позволяет вводить и редактировать текст программы, запоминать его файл и читать этот текст из файла. Обычно файла хранящие тексты программ на Си имеют расширение .c. Например, myprog.c. Для изучения текстового редактора необходимо взять некоторую среду программирования, например, Borland C.

2.4.2 Компилятор

Компилятор Си обычно состоит из двух походов. На первом — выполняется программа препроцессор, который преобразует исходный

текст программы в соответствии с директивами, записанными в тексте программы (более подробно см. описание препроцессора). На втором — производится генерация объектного кода, который записывается в файл с расширением .OBJ. Структура программы в объектном коде приблизительно такая:

1) таблица внешних символов (внешних переменных и функций) имена, которые определены в этом исходном файле;

2) таблица внешних ссылок (внешние переменные и имена функций, к которым есть обращение в данном модуле, но нет их определений).

3) машинный код программы, с учетом того, что некоторые имена еще неопределенны (см. выше пункт 2).

Компилятор может выдавать сообщения двух типов: ошибки и предупреждения. Ошибки означают, что программа не может быть транслирована в объектный код. Например, явно не объявлена какая либо переменная, или неверно записан какой либо оператор Си. Предупреждения это сообщение, на которое программисту следует обратить внимание, но компилятор генерирует объектный код. Примерами таких сообщений является сообщение о том, что в программе объявлена переменная, но нигде эта переменная не используется. Такая ситуация может быть получена при редактировании текста программы, когда часть текста, где используется переменная, удалена или помечена как комментарий, а объявление переменной было оставлено.

Что нужно знать о компиляторе — это установка различных опций, позволяющих создавать различные режимы генерации и оптимизации объектного кода.

2.4.3 Редактор связей

Редактор связей (компоновщик, линкер) программа, обеспечивающая сборку объектных модулей, вставки функций из системных библиотек и создание исполняемой программы. Эта программа просматривает таблицы внешних символов и таблицы внешних ссылок, настраивает адреса внешних переменных и функций для каждого модуля, вставляет библиотечные функции, создает заголовок программы и создает программу, исполняемую на данном компьютере. В случае если какая либо внешняя ссылка будет неопределенна, то редактор связей сгенерирует соответствующую ошибку. Ошибки редактора связей обычно связаны с неверным написанием имени функции в операторе вызова.

2.4.4 Отладчик

Отладчик (дебугер) это режим работы системы, при котором возможно:

1) пошаговое выполнение программы (этот режим очень важно освоить при изучении языка Си)

2) установка наблюдения за отдельными переменными, т.е. как переменная изменяет свое значение, то отладчик выдает ее значение;

3) проверка конкретных переменных и выражений.

2.4.5 Помощь

Развитые средства помощи (help) — очень удобное средство, однако, как правило это средство написано на английском языке. В следствии чего, им могут воспользоваться программисты, по крайней мере читающие по английски со словарем.

2.4.6 Вспомогательные средства

К ним относятся процедура Make, которая обеспечивает следующий сервис: производит проверку даты и времени создания или последней редакции файла с исходным текстом и времени и дат соответствующего объектного файла и если объектный файл был ранее создан, то данный файл с исходным текстом компилируется, в противном случае не компилируется. Эта процедура осуществляет проверку для всех модулей из текущего проекта. После проверки компиляции производится проверка исполняемого файла программы, если дата и время создания его ранее, чем одно их объектных файлов то вызывается редактор связей и создается исполняемый файл заново.

Процедура Build автоматическую компиляцию всех модулей и создание исполняемого файла программы.

2.4.7 Средства управления проектом

Достаточно большая программа обычно состоит из некоторого множества модулей. Здесь под модулем будем понимать отдельный файл с телами функций, написанных на Си.

Обычно средства управления проектом обеспечивают:

1) создание проекта;

2) добавление или удаление модуля

3) установку различных опций для трансляции данного модуля.

Глава 3. СОБЫТИЙНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Введение

Появление операционной системы (ОС) Windows ознаменовало новый этап развития методов и технологий построения программного обеспечения. Системное программное обеспечение существенно усложнилось, проявился новый слой системного программного обеспечения — графический интерфейс пользователя. Благодаря этому произошел качественный скачок в применении компьютеров — компьютеры стали использоваться повсеместно неквалифицированными пользователями. Однако чтобы создать прикладные программы для ОС Windows необходимо изменить парадигму программирования. От безраздельного господства прикладного программиста над компьютером произошел переход к программированию, основанному на взаимодействии (координации действий). Появился своеобразный диалог между операционной системой и прикладной программой, когда в получении заданного результата взаимодействовали оба слоя программного обеспечения. Для создания механизма такого взаимодействия был применен аппарат обработки событий. Этот аппарат предполагает, что ОС и прикладная программа посылает друг другу сообщения, в которых уведомляют друг друга о наступлении какого либо события. Например, ОС уведомляет прикладную программу о том, что пользователь нажал клавишу на клавиатуре или мыши, или передвинул указатель мыши, или изменил размеры окна и т.д. Прикладная программа сообщает ОС о том, что она завершила некоторую операцию, например перерисовку окна, обработку некоторого сообщения и т.д.

Таким образом, можно предложит следующую схему взаимодействия (см. рис. 12) Пользователь, используя устройства ввода (мышь, клавиатуру и пр.) желает изменить состояние прикладной программы. ОС Windows анализирует эти действия и преобразует их в форму сообщений. Например, Сообщение «Нажата клавиша на клавиатуре», «значение», «параметры». И передает это сообщение в очередь сообщений для прикладной программы. Прикладная программа анализирует эту очередь, и если есть сообщение, то производит обработку этого сообщения. В свою очередь прикладная программа может сама сформировать сообщение, однако для этого используется специально разработанный интерфейс — программный интерфейс приложения (API — application program interface).

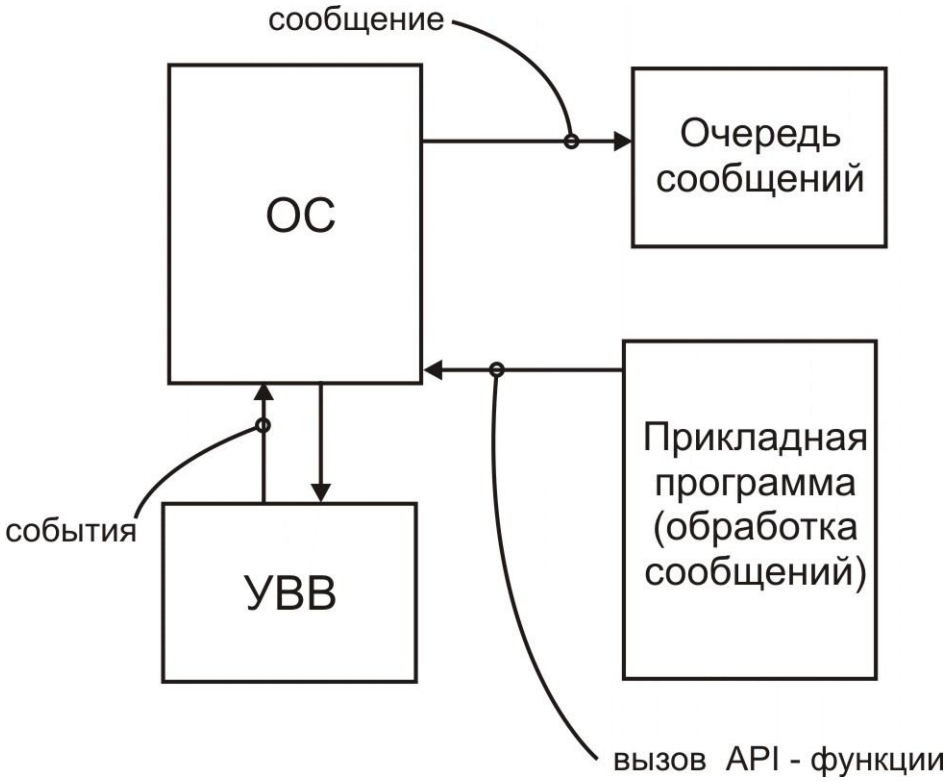


Рис. 12 — Основная схема взаимодействия ОС и прикладной программы

3.1 Графический интерфейс пользователя (GDI)

Графический интерфейс пользователя — это слой программного обеспечения операционной системы Windows, обеспечивающий эффективное взаимодействие пользователя с прикладными программами.

GDI состоит из множества объектов и функциями манипулирующими этими объектами. Важнейшими объектами являются: окна, контексты устройств, сообщения и т.д.

Окно является фундаментальным объектом ОС Windows. Под окном в ОС понимается, контейнер в котором можно хранить, обрабатывать и отображать информацию на экране. Внешний вид окна стандартен, и зависит от версии операционной системы. Оконный интерфейс имееют все объекты Windows занимающую определенную область экрана: кнопки, поля, бегунки и т.д.

Рассмотрим внешний вид окна (отображение объекта «окна» на экране компьютера»). Под внешним окна понимается выделенная прямоугольная часть крана, в которой имеется некоторая информация. Перечислим основные стандартные элементы окна (см. рис. 13): рамка, заголовок окна, рабочая область, специальные кнопки, меню, линейки прокрутки. Как правило, окно имеет выделенную границу в виде некоторой рамки, которая определяется цветом и толщиной линии. Заголовок — это прямоугольник в верхней части окна, где помещена некоторая строка символов

(обычно это название программы). Рабочая область — это прямоугольная область окна, куда помещается учебная информация. Для рабочего окна необходимо указать цвет фона. Кнопка — специальная пиктограмма, изображающая кнопку с нарисованной на ней некоторой картинкой. Например, кнопка со стрелкой.

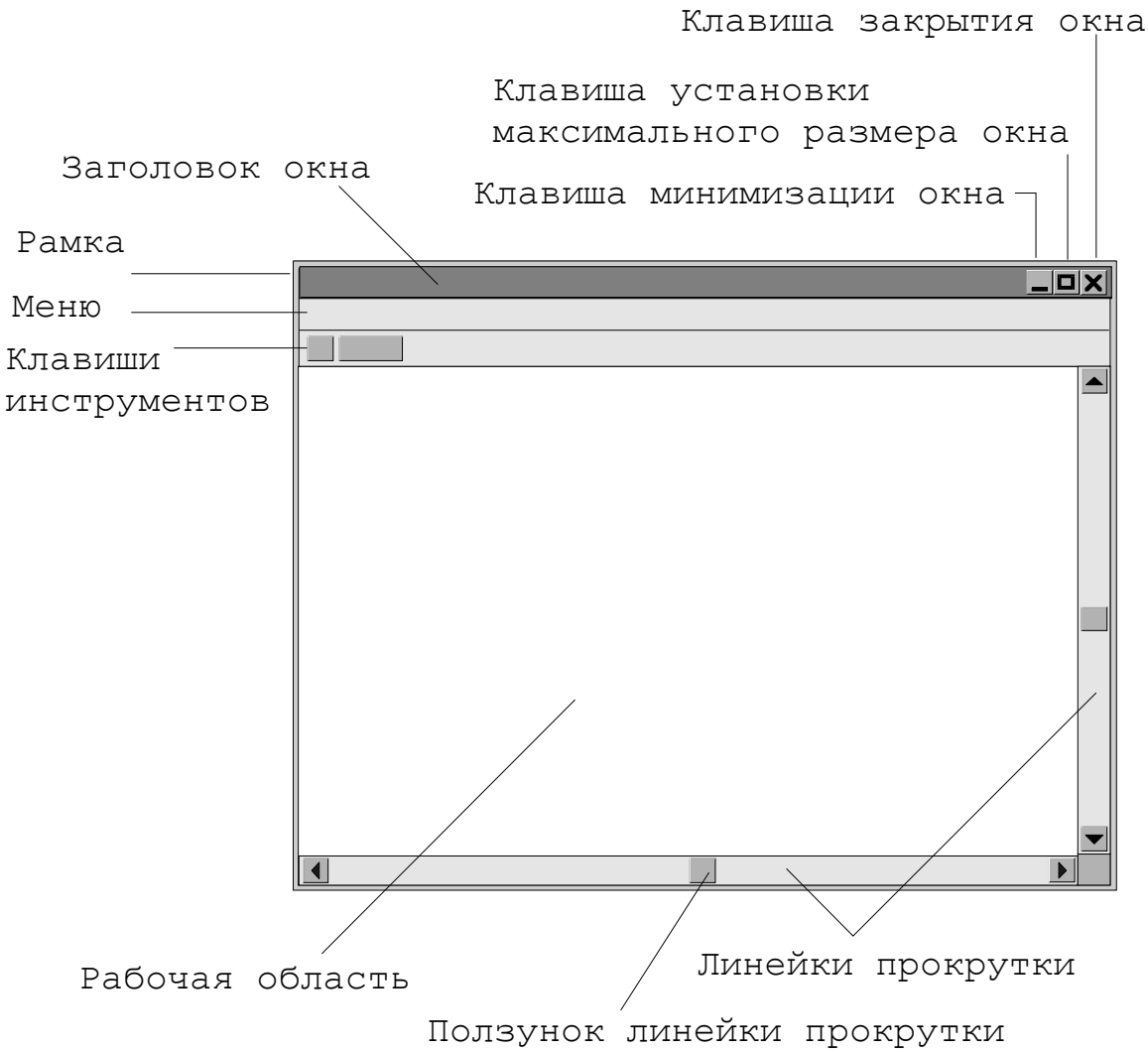


Рис. 13 — Основные элементы окна

Кнопки предназначены для инициирования некоторого действия. Например, для вызова системного меню, для изменения размеров окна и т.д. Линейки прокрутки — это специальные прямоугольники с кнопками, предназначенные для просмотра информации, большей по объему, чем может входить в рабочую область окна. Линейки прокрутки имеют две неподвижные кнопки по краям и одну кнопку-бегунок, который перемещается от одной крайней кнопки до другой. Бегунок, как правило, показывает текущее расположение выведенной в рабочей области окна порции информации ко всему объему. Линейки прокрутки могут быть вертикальными и горизонтальными. Вертикальные линейки прокрутки позволяют пе-

ремещать информацию в вертикальном направлении. Горизонтальные линейки — в горизонтальном. Для вертикальных линеек крайние кнопки имеют изображения стрелок вверх и вниз, для горизонтальной — стрелки вправо и влево. Меню окна — это прямоугольная область окна, предназначенная для организации выбора. Область меню может быть организована в верхней части, под заголовком, внизу, справа и слева. Обычно меню организовано под заголовком окна.

Рассмотрим теперь операции, которые моежт производить пользователь, с окном. Это следующие операции: переместить окно; изменить размеры окна; закрыть окно; минимизировать окно; максимизировать окно.

Рассмотрим некоторые специальные окна. Прежде всего, это диалоговые панели. Диалоговые панели — это окна, предназначенные для организации передачи небольшой порции информации между пользователем и программой. Как правило, они появляются на фоне главного окна. Примером диалоговых панелей является окно, в котором требуется подтверждение о выполнении некоторой операции (например, удаление) или ввести имя файла, или указать один из возможных параметров для некоторого режима работы КУП и т.д. Диалоговые окна могут работать в двух режимах: в модальном и немодальном.

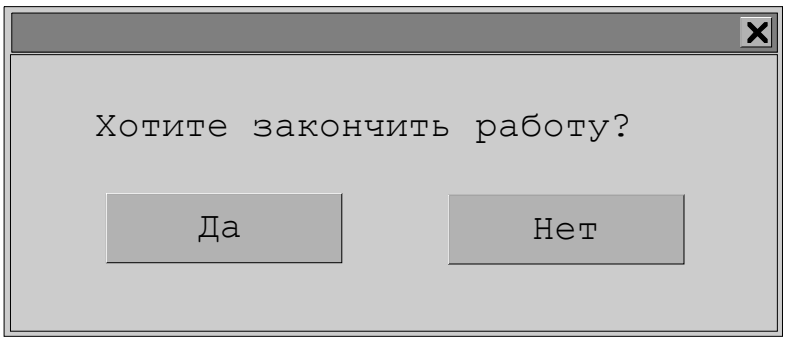


Рис. 14 — Пример стандартного диалогового окна

В модальном режиме все внимание диалоговая панель фокусирует на себя, т.е. нельзя переключиться на другое окно, пока не будет закрыта данная диалоговая панель. В немодальном режиме помимо диалоговой панели можно работать с другими окнами, например вызвать меню в главном окне.

Для диалоговых панелей характерно использование стандартных элементов управления, которые позволяют пользователю представить информацию в ясной и легко воспринимаемой форме. К ним относятся:

– статические поля;
– перечни;
– редактируемые поля;
– отмеченные блоки;
– кнопки-селекторы;
– группирующие рамки;
– кнопки.

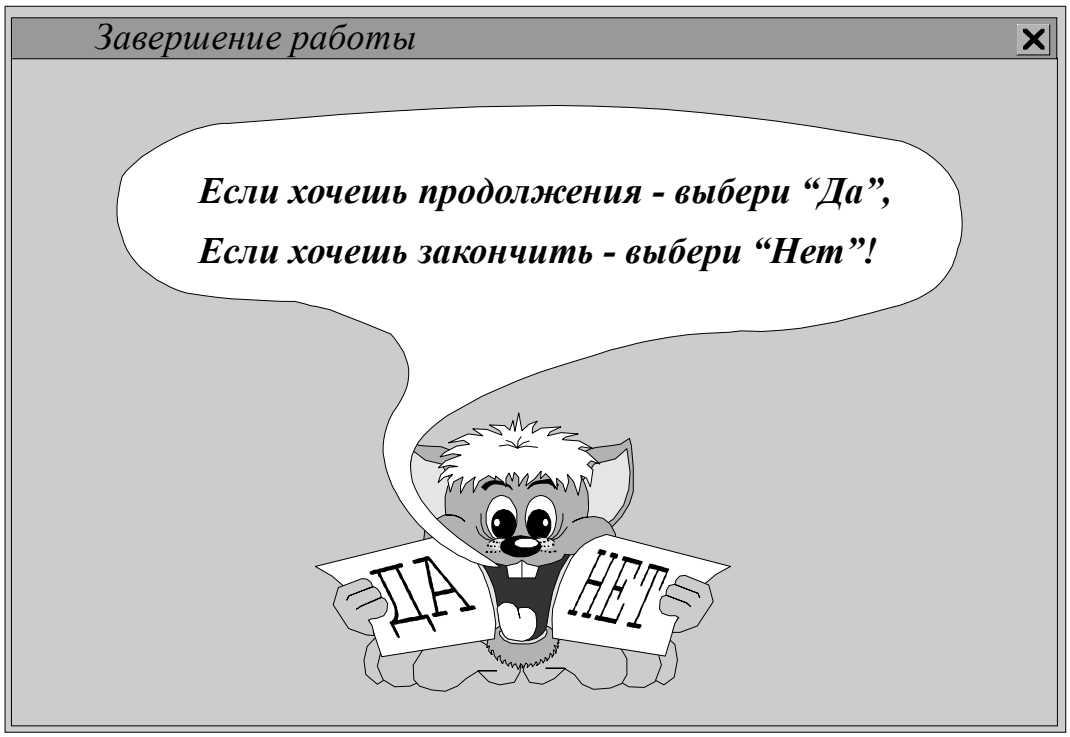


Рис. 15 — Организация диалогового окна с использованием комиксов

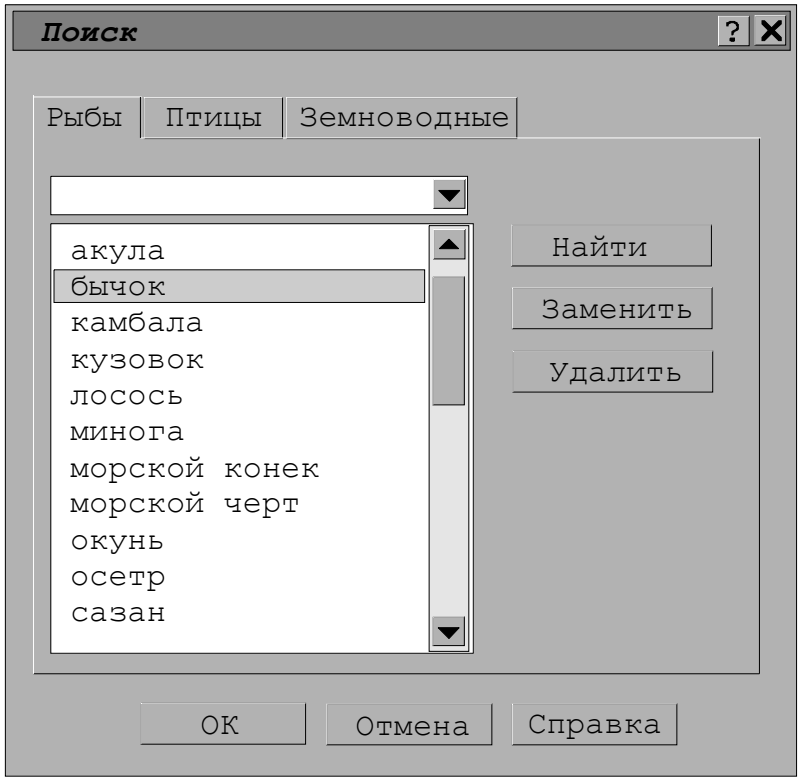


Рис. 16 — Организация поиска информации в диалоговом окне
с использованием стандартных элементов управления

Статические поля — это окна, которые в диалоговой панели предназначены для представления некоторой информации, например, некоторые имя файла или тип файла и т.д. Местоположение и размеры статических элементов не меняются. Перечни — это окна, предназначенные для вывода и просмотра списка строк символов. Строки расположены вертикально и существует механизм прокрутки для больших списков, используя вертикальную линейку прокрутки. Редактируемые поля — это окна, предназначенные для ввода и редактирования некоторой строки символов. Для ввода и редактирования длинных строк символов редактируемые поля могут иметь горизонтальные линейки прокрутки. Кнопки-селекторы предназначены для выбора одной из нескольких взаимоисключающих ситуаций. Группирующая рамка объединяет совокупность кнопок-селекторов в некоторое целое. Отмечаемые блоки — это небольшое окно с рамкой и присоединенным к нему именем. Если в этом окне записан крестик, или круг, или еще какой-либо символ или изображение, то считается, что блок отмечен, в противном случае — не отмечен. Кнопки — специальные окна с изображением кнопки. Как правило, кнопка имеет объемный вид с некоторой строкой символов или изображением и предназначена для указания некоторого действия. Например, «закрыть диалоговую панель». Для выполнения действия необходимо «нажать» данную кнопку, т.е. курсор мышки установить на кнопку и нажать левую клавишу мышки. При нажатии левой клавиши мышки кнопка диалоговой панели также «нажимается» и далее производится выполнение некоторого действия.

3.2 Программирование функциональности окна

Поскольку Windows отвечает за внешний вид окна, мы можем только формировать внешний вид окна из набора стандартных элементов. С другой стороны мы должны обеспечить функциональность приложения, т.е. наполнить окна смыслом, для чего их применять и какую информацию вводить, выводить и обрабатывать. Эту функциональность обеспечивает оконная процедура — специальным образом оформленная функция, которая связывается окном. Поскольку подобных окон в ОС Windows может быть некоторое множество, то говорят об оконном классе. Таким образом, оконный класс можно определить как структуру, содержащую следующую информацию: описание внешнего вида окна и описание функциональности.

Общение Windows с окном производится путем отправки сообщения в оконную процедуру. Общение оконной процедуры с ОС производится с помощью API-функций. Общение оконной процедуры с другими окнами производится также с помощью отправки сообщений.

API можно разделить на 7 групп функций.

1. Базовый сервис. Включает функции файловой системы, управления процессами и потоками, управление объектами ядра, работу с реестром и обработку ошибок.

2. Графический интерфейс устройств. Функции для работы с терминалом, принтером и другими устройствами вывода.

3. Интерфейс пользователя. Все функции для работы с окнами, как с контейнерами.

4. Библиотека стандартных диалоговых окон. Функции для работы с диалоговыми окнами: открыть файл, сохранить, выбор цвета и шрифта и т.п.

5. Библиотека стандартных управляющих элементов. Такие как кнопки, статические поля, линейки инструментов и пр.

6. Компонент для работы с операционной системой в командном режиме (operating system shell).

7. Сетевой сервис. Функции NetBIOS, Winsock, NetDDE, RPC.

В настоящее время имеются функции для работы с Web (Internet Explorer), мультимедиа, интерфейсы OLE,DDE,COM.

3.3 Создание, идентификация и удаление объектов Windows

Для организации правильной работы приложения необходимо понимать, как и когда создаются, идентифицируются и удаляются объекты Windows. Как правило, внутренние механизмы работы Windows не раскрываются, однако на уровне интерфейсов можно предполагать наличие того или иного внутреннего механизма. Большинство объектов Windows создаются явно, выполнив вызов некоторой API-функции CreateN (здесь под N понимается имя объекта). Если создание произошло успешно, то возвращается некоторая целая величина, характеризующая это объект (дескриптор). Это может быть некоторой адрес, в адресном пространстве ядра, это может быть номер строки в некоторой таблице объектов. Как правило, это целая беззнаковая величина имеющая описание HANDLE (ручка объекта, имея которую можно манипулировать объектом). После того, как объект в приложении не нужен его следует удалить. Почти для всех объектов Windows имеется процедура удаления объекта.

Необходимо помнить, что если вовремя не удалять объекты, то ресурсы Windows быстро расходуются и используются неэффективно. С другой стороны надо очень осторожно сохранять значения дескриптора в локальную память, поскольку легко потерять значение дескриптора. В некоторых случаях, можно получить значение дескриптора объекта, используя дополнительные механизмы. Например, можно получить доступ к списку окон приложения или дочерних окон и т.д.

3.4 Основные типы Windows

Для записи прикладных программ в Windows имеется ряд соглашений и правил.

1. Венгерская нотация. В названии переменных в виде префикса из маленьких букв записывается тип переменной. Например,

hInstance, szCmdLine, wParam.

2. Основные типы

Для более точного описания Windows вводит свои собственные типы переменных. Как правило, это делается с помощью описателя typedef, например,

```
typedef unsigned long       DWORD;
typedef void far            *LPVOID;
```

Более подробную информацию можно найти просматривая файлы windows.h или windef.h. Перечислим основные типы Windows

LPVOID — указатель на void объект. Это тип указателя на любой объект (или пока неопределенный). Важно знать, что данный указатель принимает значение NULL.

STR — строковый тип данных, в котором память под строку распределена и содержит некотрую строку. Иногда добавляют префикс Z, это означает, что строка заканчивает нулем.

LPSTR — указатель на строковый тип данных.

TCHAR — тип данных описания символа ASCII 1-байт, UNICODE — 2 байта. Значение TCHAR зависит от установок в программе.

TSTR — строка символов типа TCHAR.

LPTSTR — указатель типа TSTR.

DWORD — целая тип, если переменная имеет префикс:

1) "i" — переменная принимает целые значения;
2) "c"— переменная является счетчиком;
3) "f"— переменная хранит битовые флажки;
4) "d" — переменная имеет тип DWORD.

WORD — целый тип размером 2 байта.

BYTE — целый тип размером в один байт.

UINT — целый беззнаквый тип.

LONG — целый длинный беззнаковый тип.

HANDLE — тип описывающий дескриптор объекта Windows.

HWND — тип описывающий дескриптор окна.

HINSTANCE — тип описывающий дескриптор приложения.
HMENU — тип описывающий дескриптор меню.
WPARAM — тип описывающий первый параметр сообщения.
LPARAM — тип описывающий второй параметр сообщения.

3.5 Структура приложение

Приложение в Windows является исполняемой программой с расширением EXE. При загрузке приложения в оперативную память ОС присваивает ему некоторый идентификатор, этот идентификатор имеет тип HINSTANCE. Поскольку приложение является также объектом Windows, то для работы с приложением, например, для запроса в windows некоторой информации о приложении необходимо знать дескриптор приложения.

Дескриптор приложения передается в точку входа программы, первым параметром. Кроме дескриптора приложения ОС формирует входные параметры для точки входа и устанавливает переменные окружения. Более подробно описание процесса запуска приложения можно найти в работе[Рихтер]. Приложение состоит из следующих основных элементов:

1. Главная функция (WinMain).
2. Списка оконных процедур (WndProc).
3. Ресурсы (rc).

3.6 Главная функция (WinMain)

Точкой входа в программу является вход в функцию WinMain, которое имеет следующее описание:

```
int WINAPI WinMain(
 HINSTANCE hInst,  //дескритор приложения
 HINSTANCE,        // не используется
 LPSTR szCmdLine,  //командная строка
 int iCmdShow      // описание размера первоначального окна
)
```

Параметр szCmdLine указателем на строку, в которой находится командная строка при вызове приложения.

Параметр iCmdShow задает вид и размеры главного окна при запуске приложения. И имеет следующие значения:

SW_SHOW — Активизировать окно и отобразить в текущей позиции с текущими размерами;

SW_SHOWMAXIMIZED — Активизировать окно и отобразить с максимальными размерами;

SW_SHOWMINIMIZED — Активизировать окно и отобразить в форме иконки.

Выполняет три основных действия:

1. Регистрация всех оконных классов.
2. Создание основного окна
3. Организация цикла обработки сообщений.

Регистрация оконного класса производится в два этапа:

1. Заполнение полей структуры WNDCLASS
2. Вызов функции RegisterClass и передачу адреса заполненной структуры.

Ниже описана структура оконного класса Windows

```
typedef struct {
   UINT style;
   WNDPROC lpfnWndProc;
   int cbClsExtra;
   int cbWndExtra;
   HINSTANCE hInstance;
   HICON hIcon;
   HCURSOR hCursor;
   HBRUSH hbrBackground;
   LPCTSTR lpszMenuName;
   LPCTSTR lpszClassName;
} WNDCLASS, *PWNDCLASS;
```

style — описание стиля оконного класса;

lpfnWndProc — адрес оконной процедуры;

cbClsExtra — указание на дополнительный объем ОП при создании оконного класса;

cbWndExtra — указание на дополнительный объем ОП при создании окна;

hInstance — дескриптор приложения;

hIcon — дескриптор иконки приложения;

hCursor — дескриптор курсора;

hbrBackground — дескриптор кисти;

LPCTSTR — имя ресурса меню;

lpszClassName — имя класса.

Создание основного окна производится с помощью функции CreateWindow. Эта функция создает объект Windows «Окно» и возвращает дескриптор HWND. Для того чтобы окно появилось на экране, необходимо следом за CreateWinodow запустить функцию ShowWindow.

Затем необходимо организовать цикл обработки сообщений.

```
MSG msg;
while( GetMessage( &msg, NULL, 0, 0 )) != 0)
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

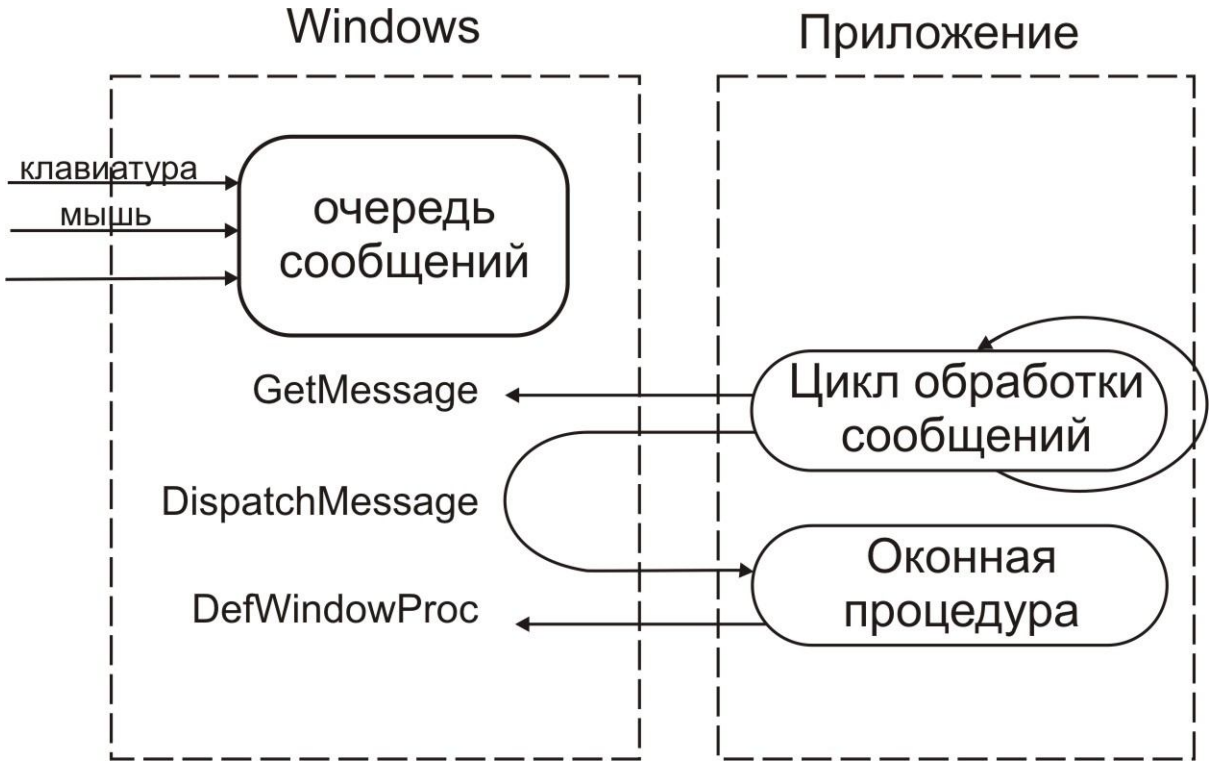


Рис. 17 — Основная схема взаимодействия Windows и приложения

Функция GetMessage обращается к очереди сообщений приложения и выбирает текущее сообщение. Если это сообщение WM_QUIT (завершить работу цикла) то функция вернет значение 0. Если это сообщение другое сообщение, то будет выполнено тело цикла.

Функция TranslateMessage(&msg) некоторые сообщения преобразует к определенному виду. Затем преобразованное сообщение передается в функцию DiapatchMessage(), которая анализирует сообщение, определяет оконную процедуру в которую необходимо передать это сообщение, вызывает эту оконную процедуру и ждет завершения обработки.

3.7 Оконная процедура

Оконная процедура вызывается функциями Windows, поэтому она имеет стандартный вид:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT msg, WPARAM
wParam, LPARAM lParam);
```

Функция WndProc возвращает 32-разрядное беззнаковое число типа LRESULT, которое интерпретируется Windows в зависимости от ситуации. Это может быть код ошибки или ее отсутствие, это может быть некоторый адрес и т.п.

CALLBACK описатель, задающий правила передачи параметров через стек.

Функция принимает следующие параметры:

hwnd — дескриптор окна, которое получает сообщение.

msg — идентификационный номер сообщения;

wParam — первый параметр сообщения;

lParam — второй параметр сообщения.

Структура оконной процедуры следующая:

```
LRESULT CALLBACK MainWndProc(HWND hwnd,UINT uMsg, WPARAM
wParam,LPARAM lParam)    {
   switch (uMsg)
   {
      case WM_CREATE:
         // Инициализация
         return 0;
      case WM_PAINT:
         // Вывод в клиентскую часть окна
         return 0;
      case WM_SIZE:
         // установка позиции и размеров окна.
         return 0;
      case WM_DESTROY:
         // удаление объектов созданных в данной процедуре
         return 0;
      //
      // Обработка других сообщений
      //
      default:
         return DefWindowProc(hwnd, uMsg, wParam, lParam);
   }
   return 0;
}
```

Довольно часто в 32-разрядные параметры wParam и lParam упаковываются несколько передаваемых значений. В этом случае можно воспользоваться макросами Windows:

LOWORD(x) — вернуть младшие 16-бит 32 разрдного числа x;

HIWORD(x) — вернуть старшие 16-бит 32 разрдного числа x;

```
LOBYTE(x)   — вернуть младшие 8 бит 16 разрядного числа x;
HIBYTE(x)   — вернуть младшие 8 бит 16 разрядного числа x.
```

3.7.1 Процесс создания окна

Для создания объекта Windows «окно» используется API-функция CreateWindow, для которой необходимо задать:

1. Имя оконного класса.
2. Текст заголовка окна.
3. Флажки, уточняющие внешний вид и поведение окна.
4. Размеры и позицию окна на экране.
5. Дескриптор родительского окна, если окно является дочерним.
6. Дескриптор дочернего окна.
7. Адрес данных для передачи в WndProc.

После вызова функции CreateWindow ОС проверяет указанные данные, распределяет память под объект «Окно», инициализирует его, создает дескриптор окна, формирует поля структуру CREATESRUCT. Вызывает оконную процедуру WndProc, в соответствии с именем оконного класса и передает в оконную процедуру:

1) дескриптор окна hwnd;
2) идентификационный номер WM_CREATE;
3) нулевой параметр wParam;
4) lParam содержит адрес структуры CREATESTRUCT.

Оконная процедура обрабатывает сообщение WM_CREATE и возвращает значение. Если значение равно нулю, то нормальное завершение работы WndProc и функция CreateWindow возвращает значение дескриптора окна. Если результат не равный нулю, то CreateWindow не создает окно и возвращает нулевое значение дескриптора окна.

3.7.2 Процесс отображения окна

После создания окна в WinMain помещается следующий код для отображения окна на экране терминала:

```
ShowWindow(hwnd, iCmdShow);
UpdateWindow(hwnd);
```

При вызыве *ShowWindow*, Windows отправляет оконной процедуре сообщения WM_SIZE и WM_SHOWWINDOW. При вызыве *UpdateWindow*, Windows отправляет оконной процедуре сообщение WM_PAINT. После этого на экране появится окно данного приложения. Более подробное обсуждения этих сообщений будет дано в разделе -.

3.7.3 Процесс завершения работы приложения

Если пользователь решил завершить работу приложения, выбрав нажатие комбинации клавиш <alt><F4> или нажав в системном меню кнопку на кнопку close, то Windows начнет удаление объектов, связанных с этим окном. При этом в процессе удаления она пошлет в оконную процедуру сообщение WM_DESTROY. В ответ на это сообщение WndProc должна освободить ресурсы и уничтожить объекты, которые были созданы во время ее работы. А затем вызвать API-функцию PostQuitMessage(0), которая ставит в очередь сообщение WM_QUIT, для завершения цикла обработки сообщений.

3.7.4 Процесс обработки сообщений по умолчанию

При работе приложения оконная процедура может получать огромное число сообщений от Windows, обработка которых не предусмотрена данной оконной процедурой. Для того чтобы необработанные сообщения не нарушили работу системы, необходимо их обрабатывать API-функцией DefWindowProc, которая знает, как обрабатывать все стандартные сообщения Windows.

DefWindowProc(hwnd, msg, wParam, lParam);

Параметры DefWindowProc теже, что и в оконной процедуре DefWindowProc.

Пример создания простейшего приложения

```
#include <windows.h>

LPSTR szClassName = "KruClass"; //имя класса
HINSTANCE hInstance;            //дескриптор приложения
//описание оконной процедуры
LRESULT CALLBACK MyWndProc(HWND, UINT, WPARAM, LPARAM);

//точка входа в приложение
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInstance,
LPSTR szCmdLine, int iCmdShow)
{
   WNDCLASS wnd;
   MSG msg;
   HWND hwnd;

   hInstance = hInst;
      // заполнение структуры оконного  класса
   wnd.style = CS_HREDRAW | CS_VREDRAW; //флажки перерисовки
   wnd.lpfnWndProc = KruWndProc;  //установка указателя на
оконную процедуру
   wnd.cbClsExtra = 0;
   wnd.cbWndExtra = 0;
```

```
   wnd.hInstance = hInstance;
   wnd.hIcon = LoadIcon(NULL, IDI_APPLICATION); //Стандартная
иконка
   wnd.hCursor = LoadCursor(NULL, IDC_ARROW);   //Стандартный
курсор
    //Стандартная кисть
   wnd.hbrBackground = (HBRUSH)(GetStockObject(WHITE_BRUSH));
   wnd.lpszMenuName = NULL;                     //Без меню
   wnd.lpszClassName = szClassName;   //Присвоение имени окон-
ному классу

   if(!RegisterClass(&wnd))           //Регистрация
   {
       MessageBox(NULL, "Ошибка регистрации","Error", MB_OK);
       return 0;
   }
//Создание главного окна
   hwnd = CreateWindow(szClassName,
          "Мое первое окно",
     WS_OVERLAPPEDWINDOW, //стандартный вид и поведение
     CW_USEDEFAULT,
           CW_USEDEFAULT,       //размеры и координаты по
умолчанию
           CW_USEDEFAULT,
           CW_USEDEFAULT,
           NULL,                //родительского окна нет
           NULL,                //меню нет
           hInstance,
           NULL);               //нет параметров для передачи
в KruWndProc
   ShowWindow(hwnd, iCmdShow);     //Отобразить окно
   UpdateWindow(hwnd);             //Послать сообщение
WM_PAINT
   while(GetMessage(&msg, NULL, 0, 0))      //Цикл обработки
сообщений
   {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
   }
   return msg.wParam;
}
//оконная процедура
LRESULT CALLBACK KruWndProc(HWND hwnd, UINT msg, WPARAM
wParam, LPARAM lParam)
{
   switch(msg)
   {
       case WM_DESTROY: //обработать сообщение об завершении
приложения
           PostQuitMessage(0); //остановить цикл обработки со-
общений
```

```
            return 0;
    }
    //остальные сообщения обрабатывать по умолчанию
    return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

3.7.5 Процесс перерисовки окна

Окна в ОС Windows можно передвигать по экрану, изменять их размер, перекрывать другими окнами и т.д. Поскольку Windows не отвечает за информацию, находящуюся в клиентской части окна, то при перерисовке окна, ОС выдает в оконную процедуру специальное сообщение WM_PAINT, которое требует от оконного окна перерисовки клиентской части.

3.8 Контекст устройства

Контекст устройства является специальной структурой, в которой определено множество графических объектов, их атрибутов и свойств обеспечивающих вывод информации на устройства вывода. Такие как принтеры, дисплеи, плоттеры и пр. Основным достоинством контекста устройства является независимость приложений от конкретного устройства вывода. Приложение, работающее под Windows, может производить вывод на различные классы принтеров и дисплеев. При этом это свойство распространяется на новые устройства, т.е. для случая, когда приложение создано ранее, чем устройство отображения. Таким образом, контекст устройства это программный интерфейс, обеспечивающий стандартный вывод информации: текстовой и графической. Контекст устройства определяется набором графических объектов и API-функций.

Графические объекты включают перо (pen) для рисования линий, кисть(brush) для закраски и заливки прямоугольников, эллипсов и прочих фигур, картинка (bitmap) необходимая для копирования и просмотра изображений, палитра для задания множества цветов, область для операций вырезания и прочих, и контур (path) для задания контурных фигур.

Объект	Атрибуты объекта
Bitmap — изображение	Размер, в байтах. Размер в пикселах. Палитра цветов. Схема сжатия
Brush — кисть	Стиль, цвет, и т.д.
Palette — палитра	Цвета и число цветов
Font — шрифт	Гарнитура, толщина,

Объект	Атрибуты объекта
	высота, ширина, множество символов и т.д.
Path — контур	контурная фигура
Pen — перо	стиль, ширина, цвет
Region — область	Размещение и размерность

Windows поддерживает пять графических мод, которые разрешают приложению указывать как цвета могут быть смешаны, где может быть произведен вывод, как производить масштабирование и т.д. Основные соды представлены в следующей таблице:

Графическая мода	Описание
Background mode	Определяет как цвет фона будет скомбинирован с цветом окна, экрана и т.д.
Drawing mode	Определяет как цвет пера, кисти, рисунка будет скомбинирован с цветом фонового изображения окна.
Mapping mode	Определяет как будет графический вывод будет отмапирован в логическое пространство окна, принтера и пр.
Polygon-fill mode	Определяет каким образом кисть будет закрашивать сложные области
Stretching mode	Определяет способ сжатия графического вывода

Для определения текущей моды используются следующие API-функции:

Графическая мода	Функция
Background mode	GetBkMode
Drawing mode	GetROP2
Mapping mode	GetMapMode
Polygon-fill mode	GetPolyFillMode
Stretching mode	GetStretchBltMode

Для изменения графической моды используются следующие функции:

Графическая мода	Функция
Background mode	SetBkMode
Drawing mode	SetROP2
Mapping mode	SetMapMode
Polygon-fill mode	SetPolyFillMode
Stretching mode	SetStretchBltMode

Windows обеспечивает четыре типа контекста: дисплей, принтер, память и информация. Каждый из которых имеет специфические цели, описанные в следующей таблице:

Контекст устройства	Цели
Display	Поддержка операций вывода на экран
Printer	Поддержка операций вывода на принтер или плоттер
Memory	Поддержка операций вывода изображение, представленное в оперативной памяти
Information	Поддержка поиска информации об устройстве вывода

3.8.1 Контекст устройства дисплея

Контекст устройства дисплея можно получить с помощью API-функций BeginPaint и GetDC. Первая функция вызывается только при обработке сообщения WM_PAINT и имеет следующее описание:

```
HDC BeginPaint(HWND hwnd,  LPPAINTSTRUCT lpPaint);
```

где hwnd — дескриптор окна;

lpPaint — указатель на структуру, содержащую информацию о контексте устройства PAINTSTRUCT. Функция возвращает дескриптор контекста устройства.

Во всех остальных случаях необходимо использовать функцию GetDC, которая имеет следующее описание:

```
HDC GetDC( HWND hwnd );
```

hwnd — дескриптор окна. Функция возвращает дескриптор контекста устройства.

После того, как контекст устройства получен в него можно производить вывод текста, линий, графических фигур, изображения и т.д. Это

производится с помощью специальных функций, где первым параметром является дескриптор контекста устройства (описание см. в следующих разделах)

В некоторых случаях необходимо изменять параметры вывода: толщину и цвет линий, цвет фона и прочее. Для этого используют механизм замены объектов в контексте устройства. Замена производится с помощью функции SelectObject, которая производит замену графического объекта в контексте устройства на новый. Причем новый объект должен быть заранее создан. Описание функции следующее:

```
HGDIOBJ SelectObject( HDC hdc,  HGDIOBJ hgdiobj );
```

где hdc — дескриптор контекста устройства;
hgdiobj — дескриптор нового объекта.
Функция возвращает дескриптор старого объекта.

Для закрытия контекста устройства используются функции EndPaint и ReleaseDC. Первая функция используется для случая открытия контекста с помощью функции BeginPaint, вторая — для GetDC.

Обычная схема работы для вывода в контекст устройства следующая:

1) открыть или создать контекст устройства и получить дескриптор;
2) создать графические объекты для вывода информации (pen,brush,font, bitmap и пр.)
3) заменить объекты в контексте устройства, сохранить дескрипторы старых объектов;
4) произвести вывод информации;
5) восстановить старые объекты в контексте устройства;
6) удалить созданные графические объекты.
7) закрыть или удалить контекст устройства.

3.8.2 Контекст устройства для работы с принтером

Для работы с принтером контекст устройства создается с помощью функции CreateDC, описание функции следующее:

```
HDC CreateDC(
   LPCTSTR lpszDriver,// строка содержащая имя драйвера
   LPCTSTR lpszDevice,//строка содержащая имя устройства
   LPCTSTR lpszOutput,// NULL
   CONST DEVMODE *lpInitData //параметры принтера
  );
```

После вывода на принтер необходимо удалить контекст устройства с помощью функции DeleteDC.

3.8.3 Контекст устройства для работы с памятью

Windows в оперативной памяти может создать битовую карту, аналог растрового изображения, и весь графический вывод производить в эту карту. После того как графическое изображение получено в памяти, его можно вывести на устройство отображения. Такой подход может быть более эффективным, поскольку скорость вывода в ОП в разы выше скорости работы с видеопамятью. Для работы с контекст устройства создается с помощью функции CreateCompatibleDC, описание функции следующее:

HDC CreateCompatibleDC(HDC hdc);

где hdc — контекст устройства, на который будет осуществляться вывод изображения. Функция вернет дескриптор контекста устройства для вывода в изображение, находящееся в оперативной памяти.

Для удаления контекста устройства необходимо использовать функцию DeleteDC.

3.9 Шрифты и вывод текста

Под текстом будем понимать последовательность предложений, слов и знаков, построенную согласно правилам некоторого данного языка, данной знаковой системы и образующую некоторое сообщение [52]. Другое определение «текст — форма предъявления информации с помощью знаков определенной письменности того или иного естественного языка. Восприятие текста непосредственно сопровождается его трансформацией на естественный, «живой» язык» [53]. Текст записывается с помощью символов, эти символы можно разделить на буквы алфавита, цифры, знаки пунктуации и специальные знаки (математические, химические и т.д.).

Накоплен огромный опыт организации и использования текстовой информации. Этот опыт широко используется при создании газет, журналов и книг, в том числе и при создании разнообразных учебников. Книга до недавнего времени являлась основным источником хранения и передачи текстовой информации. В настоящее время с появлением компьютерных сетей происходит переход к компьютерным технологиям хранения и передачи текстовой информации. Тем не менее, этот опыт имеет огромное значение и для компьютерных технологий хранения и передачи текстовой информации, поскольку меняется только носитель — вместо бумаги используется электронный экран. Для компьютерного представления и предъявления текстовой информации необходимо рассмотреть следующее:

1) вопросы компьютерного представления текста;

2) вопросы организации и использования шрифтов;
3) способы организации и записи текста;
4) способы ввода и преобразования текста.

3.9.1 Компьютерное представление текста

В памяти компьютера текст представляется в виде последовательности двоичных чисел (кодов). Каждый символ занимает один байт, в который можно записать всего 256 возможных значений двоичных чисел. Поэтому в памяти компьютера текст представляется не более 256 различными символами. В настоящее время существуют стандартные таблицы для представления и передачи текстовой информации. В этих таблицах каждому символу из некоторого множества соответствует некоторое двоичное число — код. Эти таблицы лежат в основе разработки устройств ввода, обработки и вывода текстовой информации (клавиатура, терминал, печать и др.) Наиболее стандартные — ASCII (Американский стандартный код для обмена информации), EBCDIC (код, разработанный фирмой IBM), ДКОИ (двоичный код для обмена информацией) и др. Современные персональные компьютеры имеют стандартный ASCII код. Необходимо отметить, что этот код семиразрядный и соответственно в кодовой таблице компьютера занимает 128 первых позиций. Остальные 128 позиций кодовой таблицы могут меняться.

В настоящее время создан еще один стандарт Unicode. В этом стандарте для представления символа используется 16 бит. Это позволяет присвоить коды символов для всех известных языков. Тогда текст, представленный в Unicode, может быть мультиязыковым, написанным одновременно на разных языках.

3.9.2 Шрифт

Основные определения

Шрифт является важнейшим элементом предъявления текстовой учебной информации. Дадим ряд определений [54]:

1. ***Графема*** — линейно осевая структура буквенного знака, определяющая его основной отличительный признак относительно других единиц алфавита.

2. ***Буква*** — графический знак, который используется для обозначения на письме фонем, их основных вариантов или их типичных последовательностей.

3. ***Алфавит*** (азбука) — совокупность расположенных в определенном порядке графических знаков, сложившихся в определенный исторический период и использующихся для фиксации и передачи на письме данного языка.

4. ***Шрифт*** — система букв, цифр и других графических знаков, имеющих общую закономерность, обусловленную конкретной языковой ситуацией или художественной целесообразностью.

Шрифты могут быть рисованными, типографскими и компьютерными. Рисованный шрифт — шрифт, нарисованный художником. Шрифт типографский — комплект букв, цифр и знаков в виде литер или изображений, используемых для набора текста. Шрифты различаются по характеру рисунка. Шрифты, объединенные общностью рисунка, называются ***гарнитурой.*** Внутри гарнитуры шрифты подразделяются по начертанию и размеру. Начертание шрифта зависит от положения ***очка*** букв и знаков. Различают:

1. Прямое начертание — основные штрихи расположены вертикально.
2. Курсивное начертание — основные штрихи букв и знаков имеют наклон 15°, в большинстве случаев вправо. Начертание имитирует рукописные изображения букв.

Шрифты также различают по насыщенности штрихов букв и знаков, которая зависит от отношения ширины основного штриха буквы нормального начертания к ее внутреннему просвету:

1. Светлое начертание — внутри буквенный просвет в 2–4 раза больше ширины основного штриха.
2. Полужирное начертание — с примерно равным внутри буквенным просветом и толщиной основного штриха.
3. Жирное начертание — внутри буквенный просвет уже толщины основного штриха.

По начертанию также выделяют ***капитель*** — шрифт, в котором буквы по высоте равны строчным, но имеют рисунок прописных букв.

Размер шрифта определяется кеглем, измеряемым в пунктах (1 пункт равен 0,375 мм, или 1/72 дюйма). Названия шрифтов в зависимости от размера приведены в следующей таблице:

Название	**Размер**
Бриллиант	3
Диамант	4
Перл	5
Нонпарель	6
Миньон	7
Петит	8
Боргес	9
Корпус	10
Цицеро	11
Миттель	14
Терция	16
Такт	20

Компьютерные шрифты

Компьютерные шрифты существенно зависят от устройств вывода текстовой информации, прежде всего от терминалов и принтеров. Первоначально шрифты были жестко встроены в соответствующие устройства вывода, при этом они содержали строчные буквы С появлением устройств вывода с графическими режимами стало возможным выводить на экран терминала и печатать различными шрифтами. Чем выше разрешающая способность устройства вывода, тем больше возможностей для использования разнообразных шрифтов. В настоящее время различают следующие типы компьютерных шрифтов:

1) растровые;
2) векторные;
3) шрифты TrueType.

РАСТРОВЫЕ ШРИФТЫ

Каждый символ растрового шрифта представляет собой некоторую матрицу, где элемент матрицы a_{ij} определяет наличие точки в изображении данного символа. Если элемент равен 1, то имеется точка в изображении символа, если 0 — нет. Размер матрицы, как правило, для всех символов одинаков, поэтому растровые шрифты иногда измеряются размерностью матрицы. Например, шрифты 8×8, 8×10, 10×14, 32×32 и т.д.

ВЕКТОРНЫЕ ШРИФТЫ

В векторном шрифте каждый символ представлен в виде отрезков прямых. Каждый отрезок задается в виде пары точек $(x_0,y_0),(x_1,y_1)$. Используется также и другое представление — в виде последовательности команд: «установи карандаш» и «рисуй отрезок». Например:

установи x_0,y_0
рисуй x_1,y_1
рисуй x_2,y_2
и т.д.

Первая команда устанавливает начальную точку, а следующие команды рисуют отрезки. Такое представление используется для векторных шрифтов фирмы Borland. Заданный таким образом шрифт можно использовать для устройств отображения с векторной графикой, например перьевой графопостроитель. Достоинством векторных шрифтов является возможность аффинных преобразований (изменение масштаба, различные по-

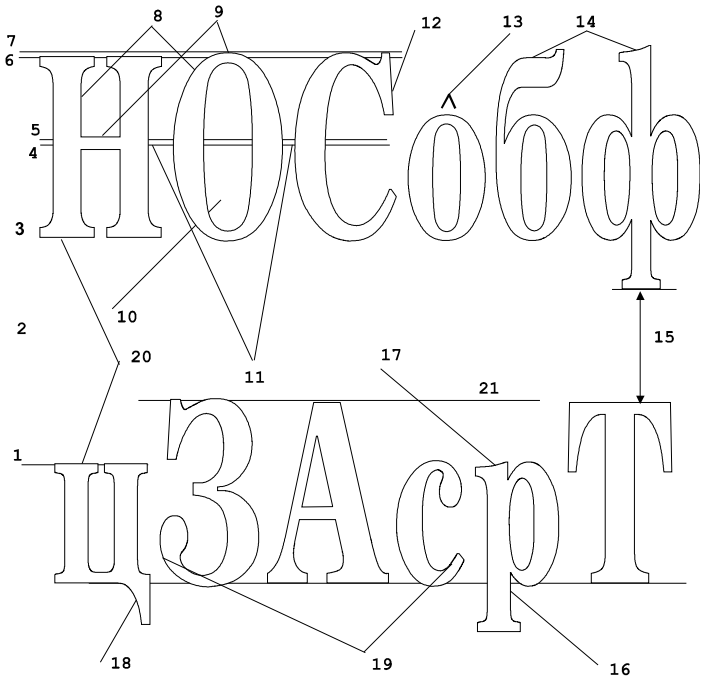
вороты и наклоны). Кроме того, при выводе можно менять толщину и вид линий.

ШРИФТЫ TrueType

Шрифты TrueType организованы таким образом, что они сочетают в себе достоинства векторных и растровых шрифтов. Это достигается следующим образом: каждый символ в шрифте представлен совокупностью точек и алгоритмов преобразования; при выводе на экран или принтер символ предварительно преобразуются в растровую картинку. Используя данный подход, стало возможным иметь шрифты, символы которых выглядят одинаково как на экране, так и на принтере.

Основные параметры компьютерных шрифтов

Ниже даны элементы буквенных знаков и их названия [78].



1. Верхняя линия строчных букв.
2. Междустрочие.
3. Линия шрифта.
4. Математическая линия.
5. Оптическая линия.
6. Верхняя линия прописных букв.
7. Верхняя линия округлых прописных букв.
8. Основные штрихи.
9. Соединительные штрихи.

10. Внутрибуквенные просветы.
11. Межбуквенные расстояния.
12. Вертикальные засечки.
13. Надбуквенные засечки (акценты).
14. Выступающие элементы.
15. Наименьшее междустрочие.
16. Свисающий элемент.
17. Односторонняя засечка.
18. Подбуквенные значки.
19. Росчерки.
20. Засечки.
21. Линия остроконечных прописных букв.

Для компьютерного представления шрифта необходимо знать следующие параметры:

1) высоту символа;
2) ширину символа;
3) высоту свисающего элемента для каждого символа.
4) величину наименьшего междустрочия;
5) начало системы координат;
6) способ представления символов;
7) величину межбуквенных пробелов.

Виды компьютерных шрифтов

Обычно компьютерные шрифты содержат:

– прописные и строчные буквы латинского алфавита;
– цифры;
– знаки препинания;
– прописные и строчные буквы кириллицы;
– разделители и специальные символы.

Кроме этого существуют специальные шрифты:

1) шрифты для других языков (греческий, арабский, готический, древнерусский и т.д.);
2) шрифты математические;
3) шрифты, содержащие различные знаки.

В настоящее время существует огромное множество разнообразных компьютерных шрифтов. Необходимо помнить, что доступны, как правило, шрифты, имеющиеся в данной операционной системе. Для них имеется специальная библиотека подпрограмм, обеспечивающая возможность использования их в разрабатываемых КУП. Например, в операционной системе MS WINDOWS имеются специальные подпрограммы для работы со шрифтами [79]. Если используется некоторая инструментальная система, то доступны ее шрифты. Для использования шрифтов различных тексто-

вых редакторов необходимо знать их формат шрифтов, а также форматы хранения текста.

Специальные шрифты

Рассмотрим специальные шрифты на примере математических шрифтов. Особенность записи математических выражений и формул заключается в необходимости масштабирования некоторых знаков в зависимости от длины выражения. Например, корень квадратный: $\sqrt{2}$ или $\sqrt{x^2+y^2}$ — ширина знака «корень квадратный» разная и зависит от подкоренного выражения. Поэтому возникают проблемы использования шрифтов с фиксированной длиной символов для записи математических выражений. Например, используя шрифт с фиксированный длиной символов, можно записать выражение для корня квадратного, а вертикальную линию далее нарисовать требуемой длины, но проблема появляется при редактировании такого выражения (увеличение и уменьшение длины подкоренного выражения). Поэтому в настоящее время разрабатываются специальные редакторы для записи математических выражений, в которых используется шрифт математических символов с эффектом масштабирования. Например, редактор Equation II, который имеется для текстового редактора MicrosoftWord/80/.

Шрифты в Windows

В Windows имеется большое число разнообразных возможностей для работы с шрифтами. Основные шаги для использования шрифта следующие:

1) создание шрифта;
2) загрузка в контекст устройства;
3) определение параметров шрифта;
4) вывод текста;
5) восстановление старого шрифта в контекста устройства.
6) удаление шрифта.

Создание шрифта производится с помощью функций CreateFont и CreateFontIndirect. Первая функция требует указать 14 параметров, вторая требует на входе указать адрес структуры, содержащий эти 14 параметров.

Перечислим эти параметры:

int nHeight,	Высота шрифта
int nWidth,	Ширина шрифта
int nEscapement,	Угол наклона шрифта
int nOrientation,	Ориентация
int fnWeight,	Жирность шрифта

DWORD fdwItalic,	Курсив
DWORD fdwUnderline,	Подчеркивание символов
DWORD fdwStrikeOut,	Зачеркивание символов
DWORD fdwCharSet,	Тип символьного набора
DWORD fdwOutputPrecision,	Точность вывода символов
DWORD fdwClipPrecision,	Точность обрезки символов
DWORD fdwQuality,	Точность соответствия между требуемым шрифтом и имеющимся
DWORD fdwPitchAndFamily,	Определяет шрифт близкий к заданному
LPCTSTR lpszFace	Имя гарнитуры шрифта

Описание функции CreateFont

```
HFONT CreateFont(int nHeight, int nWidth, int nEscapement, int nOrientation,
        int fnWeight, DWORD fdwItalic, DWORD fdwUnderline, DWORD
fdwStrikeOut,
DWORD fdwCharSet, DWORD fdwOutputPrecision, DWORD fdwClipPreci-
sion,   DWORD fdwQuality, DWORD fdwPitchAndFamily, LPCTSTR
lpszFace);
```

Описание функции CreateFontIndirect

HFONT CreateFontIndirect(CONST LOGFONT *lplf); где
LOGFONT имеет следующую структуру:

```
typedef struct tagLOGFONT { // lf
  LONG lfHeight;
  LONG lfWidth;
  LONG lfEscapement;
  LONG lfOrientation;
  LONG lfWeight;
  BYTE lfItalic;
  BYTE lfUnderline;
  BYTE lfStrikeOut;
  BYTE lfCharSet;
  BYTE lfOutPrecision;
  BYTE lfClipPrecision;
  BYTE lfQuality;
  BYTE lfPitchAndFamily;
  TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT;
```

Описанные выше функции возвращают дескриптор созданного шрифта (HFONT).

Подробную информацию о шрифте в Windows можно получить, используя функцию GetTextMetrics, описание которой следующее:

```
BOOL GetTextMetrics(HDC hdc, LPTEXTMETRIC lptm);
```

где hdc — дескриптор контекста устройства;
lptm — адрес структуры TEXTMETRIC.

```
typedef struct tagTEXTMETRIC { // tm
   LONG tmHeight;
   LONG tmAscent;
   LONG tmDescent;
   LONG tmInternalLeading;
   LONG tmExternalLeading;
   LONG tmAveCharWidth;
   LONG tmMaxCharWidth;
   LONG tmWeight;
   LONG tmOverhang;
   LONG tmDigitizedAspectX;
   LONG tmDigitizedAspectY;
   BCHAR tmFirstChar;
   BCHAR tmLastChar;
   BCHAR tmDefaultChar;
   BCHAR tmBreakChar;
   BYTE tmItalic;
   BYTE tmUnderlined;
   BYTE tmStruckOut;
   BYTE tmPitchAndFamily;
   BYTE tmCharSet;
} TEXTMETRIC;
```

Рисунок

В некоторых случаях удобная функция GetTextExtentPoint32, которая определяет размеры прямоугольной области, в которую вписывается строка символов. Описание этой функции следующее:

```
BOOL GetTextExtentPoint32(
   HDC hdc, //дескриптор контекста устройства
   LPCTSTR lpString,//адрес строки символов
   int cbString,//число символов в строке
   LPSIZE lpSize //адрес структуры SIZE
  );
```

Структура SIZE опередеяет размеры прямоугольной области

```
typedef struct tagSIZE {
   LONG cx;  //ширина
   LONG cy; //высота
} SIZE;
```

3.9.3 Вывод текста в Windows

Имеется две основные функции вывод текста в Windows: DrawText и TextOut. Простейшая из них TextOut, описание которой дано ниже:

```
BOOL TextOut(   HDC hdc,//дескриптор контекста устройства
   int nXStart,// x-координата для вывода строки
   int nYStart,// y- координата для вывода строки
   LPCTSTR lpString,// адрес строки
   int cbString //число символов в строке
  );
```

Функция выводит строку символов в контекст устройства с заданными объектами (цвет, шрифт, фон и т.д.) и координатами левого верхнего угла.

Функция DrawText позволяет организовать форматный вывод строки символов в прямоугольную область контекста устройства. Описание функции следующее:

```
int DrawText(HDC hDC, //дескриптор контекста устройства
   LPCTSTR lpString,//адрес строки символов
   int nCount,  // число символов
   LPRECT lpRect, //адрес структуры, описывающей область вывода
   UINT uFormat   // флажки с указанием способа форматирования
  );
```

Функция осуществляет форматирование текста, записанного в строке по адресу lpString, длинной nCount и вывод в прямоугольную область контекста, заданную в структуре по адресу lpRect. Если nCount равен -1, то функция вычисляет длину самостоятельно. Если прямоугольную область необходимо вычислить, то DrawText производит вычисление этой области. Если строка символов не содержит символы перевода строки, то рассматривают случай SINGLELINE, иначе случай MULTILINE. uFormat — является комбинацией значений флажков, значения некоторых из них записано в следующей таблице:

DT_BOTTOM	Вывод строки с выравниванием относительно нижней линии прямоугольника. При это должен быть установлен флажок DT_SINGLELINE
DT_CALCRECT	При установке этого флага функция не производит вывод строки, а вычисляет параметры прямоугольной области. В случае SINGLELINE вычисляется и ширина и высота прямоугольника, адрес которого задан в lpRect. В случае MULTILINE в прямоугольнике должна быть задана ширина
DT_CENTER	Вывод текста относительно центра прямоугольной области
DT_EXPANDTABS	Вывод с учетом табуляции
DT_LEFT	Выравнивание относительной левой границы прямоугольной области
DT_RIGHT	Выравнивание относительно правой границы прямоугольной области
DT_SINGLELINE	Воспринимает текст как однострочный, игнорируя символы перевода строки и возврата каретки
DT_TOP	Вывод текста относительно верхней границы прямоугольной области
DT_VCENTER	Вывод текста в центре относительно вертикальной линии
DT_WORDBREAK	Автоматический переход на новую строку, если слова не помещается в текущей строке

Рассмотрим пример использования DrawText

```
 HDC hdc; //дескриптор контекста
PAINTSTRUCT ps; //структура для BeginPain
RECT rectClient; //структура для прямоугольника
int nWidth, nHeight;. //ширина и высота клиентской части окна
hdc = BeginPaint( hwnd, &ps ); //определяем дескриптор контекста устройства
GetClientRect( hwnd , &rectClient ); .//определяем размер клиентской части окна
nWidth = rectClient.right;    // вычисляем ширину(rectClient.left равен 0)
nHeight = rectClient.bottom;  //вычисляем высоту (rectClient.top равен 0)
rectClient.left += (nWidth / 4);//вычисляем новые координаты прямоугольника
rectClient.top  += (nHeight / 4);
rectClient.right -= (nWidth / 4);
```

```
SetBkMode( hdc, TRANSPARENT ); //устанавливаем моду для вывода текста
//выводим текст в заданный прямоугольник по центу с переносом слов
DrawText( hdc, lpszClientAreaText, lstrlen( lpszClientAreaText ),
  &rectClient, DT_CENTER | DT_VCENTER | DT_WORDBREAK );
EndPaint( hwnd, &ps ); //закрываем контекст устройства
```

3.10 Вывод фигур и линий

В Windows имеется набор API-функций, который обеспечивает вывод фигур и линий. Можно выводить: прямоугольники, эллипсы, сектора, многоугольники, дуги, конечные точки которых соединены прямой. Все указанные фигуры выводятся в контекст устройства с установленными там пером и кистью.

Перо объект контекста устройства, который задает свойства линий и границ при рисовании графических фигур. Перо имеет три атрибута: толщина, цвет и стиль. Толщина и цвет задают толщину и цвет линии рисования. Стиль задает разные способы рисования линий. Имеется следующие стили:

PS_SOLID	Сплошная линия
PS_DASH	Пунктир черточками
PS_DOT	Пунктир точками
PS_LASHDOT	Штрихпунктирная линия
PS_INSIDEFRAME	Линия внутри рамки замкнутой фигуры

Для создания пера имеется следующие функции CreatePen и CreatePenIndirect.

```
HPEN CreatePen(
  int fnPenStyle,// стиль пера
  int nWidth,// толщина пера
  COLORREF crColor //цвет пера
 );

HPEN CreatePenIndirect(
  CONST LOGPEN *lplgpn // указатель на структуру пера
 );
```

Описание структуры пера следующее:

```
typedef struct tagLOGPEN {
  UINT     lopnStyle;  //стиль пера
  POINT    lopnWidth;//толщина пера
```

```
   COLORREF lopnColor; //цвет пера
} LOGPEN;
```

Цвет пера задается специальным макросом RGB, сочетание цветов: красный, зеленый, голубой.

```
COLORREF RGB(
   BYTE bRed,// компонента красного цвета (0-255)
   BYTE bGreen,// компонента зеленого цвета (0-255)
   BYTE bBlue// компонента голубого цвета (0-255)
  );
```

сам макрос определен в Windows.h как

```
#define RGB(r, g ,b)  ((DWORD) (((BYTE) (r) | \
   ((WORD) (g) << 8)) | \
   (((DWORD) (BYTE) (b)) << 16)))
```

Пример,

```
HPEN hpen, hpenOld;
   // Создание сплошного пера, толщиной 10, зеленого цвета
   hpen = CreatePen(PS_SOLID, 10, RGB(0, 255, 0));
   // выбрать новое перо для контекста устройства, старое сохранить
   hpenOld = SelectObject(hdc, hpen);
   //Вывести фигуру с заданным пером
   Rectangle(hdc, 100,100, 200,200);

//Восстановление старого пера в контексте устройства
   SelectObject(hdc, hpenOld);
//удаление нового пера
          DeleteObject(hpen);
```

Кисть(brush) в Windows обеспечивает закраску замкнутой фигуры и могут быть: сплошными (solid), штриховыми (hatch), заданными прикладным программистом (pattern). Соответственно для каждого типа кисти существует своя функция создания CreateSolidBrush, CreateHatchBrush, CreatePatternBrush. Описание первой следующее:

```
HBRUSH CreateSolidBrush(
   COLORREF crColor //цвет закраски
  );
```

Описание функции создания штриховой кисти следующее:

```
HBRUSH CreateHatchBrush(
   int fnStyle,// стиль штриха
   COLORREF clrref // цвет
  );
```

Параметр fnStyle принимает следующие значения:
HS_BDIAGONAL
HS_CROSS
HS_DIAGCROSS
HS_FDIAGONAL
HS_HORIZONTAL
HS_VERTICAL

Прикладной программист может создать свою собственную кисть, для этого надо использовать функцию CreatePatternBrush. Описание первой следующее:

```
HBRUSH hbrush, hbrushOld;
   // Создание сплошной кисти зеленого цвета
   hbrush = CreateSolidBrush( RGB(0, 255, 0));
   // выбрать новое породля контекста устройства, старое сохранить
   hbrushOld = SelectObject(hdc, hbrush);
   //Вывести фигуру с заданной кистью
   Ellipse(hdc, 100,100, 200,200);

//Восстановление старой кисти в контексте устройства
   SelectObject(hdc, hdrushOld);
//удаление новой кисти
   DeleteObject(hbrush);
```

3.10.1 Функции вывода фигур

Основные функции вывода фигур следующие: Chord, Ellipse, Pie, Polygon, Rectangle. Запишем описания этих функций:

```
BOOL Chord(
   HDC hdc,// контекст устройства
   int nLeftRect,// x-координата левого верхнего угла прямоугольника
   int nTopRect,// y- координата левого верхнего угла прямоугольника
   int nRightRect,// x- координата правого верхнего угла прямоугольника
   int nBottomRect,// y- координата правого верхнего угла прямоугольника
   int nXRadial1,// x- координата точки начала хорды
   int nYRadial1,// y- координата точки начала хорды
   int nXRadial2,// x- координата конечной точки  хорды
   int nYRadial2 // y- координата конечной точки  хорды
  );

BOOL Ellipse(
   HDC hdc,//  контекст устройства
   int nLeftRect,//x-координата левого верхнего угла прямоугольника
   int nTopRect,// y- координата левого верхнего угла прямоугольника
```

```
   int nRightRect,// x- координата правого верхнего угла прямоугольника
   int nBottomRect // y- координата правого верхнего угла прямоугольника
  );

BOOL Pie(
   HDC hdc,// контекст устройства
   int nLeftRect,// x-координата левого верхнего угла прямоугольника
   int nTopRect,// y- координата левого верхнего угла прямоугольника
   int nRightRect,// x- координата правого верхнего угла прямоугольника
   int nBottomRect,// y- координата правого верхнего угла прямоугольника
   int nXRadial1,// x- координата точки начала хорды
   int nYRadial1,// y- координата точки начала хорды
   int nXRadial2,// x- координата конечной точки  хорды
   int nYRadial2 // y- координата конечной точки  хорды
  );

BOOL Polygon(
   HDC hdc, // контекст устройства
   CONST POINT *lpPoints,//массив точек вершин многоугольника
   int nCount // число вершин у многоугольника
  );
```

где POINT описана как

```
typedef struct tagPOINT { // pt
   LONG x;
   LONG y;
} POINT;

BOOL Rectangle(
   HDC hdc, // контекст устройства
   int nLeftRect,// x-координата левого верхнего угла прямоугольника
   int nTopRect,// y- координата левого верхнего угла прямоугольника
   int nRightRect,// x- координата правого верхнего угла прямоугольника
   int nBottomRect,// y- координата правого верхнего угла прямоугольникar
  );
```

3.10.2 Вывод линий

Вывод линий в Windows можно осуществить с помощью функций LineTo, PolyLine, Arc, PolyBezier.

Для вывода прямых линий имеются функции: MoveToEx, LineTo и PolyLine. Первые две работают в паре. Функция MoveToEx устанавливает текущую позицию пера, а функция LineTo выводит прямую линию от те-

кущей позиции до конечной позиции, причем после вывода прямой, текущей позицией становится последняя точка, указанная в параметрах LineTo. Описание функций следущее:

```
BOOL MoveToEx(
HDC hdc, //дескриптор контекст устройства
int X, //x координата начальной точки
int Y,//y координата начальной точки
LPPOINT lpPoint//указатель на структуру POIN предыдущей текущей позиции
);

BOOL LineTo(
HDC hdc,  // дескриптор контекст устройства
int nXEnd, // x-координата конечной точки
int nYEnd  //y-координата конечной точки
);
```

Функция PolyLine обеспечивает вывод ломанной линии, состоящей из отдельных прямых отрезков. Координаты отрезков записаны в массиве POINT. Описание функции следующее:

```
BOOL Polyline(
   HDC hdc,// дескриптор контекст устройства
   CONST POINT *lppt,// адрес массива точек ломанной линии.
   int cPoints // число точек в массиве
  );
```

3.10.3 Вывод кривых (curves)

Вывод кривых в Windows осуществляется с помощью Arc и PolyBezier.

Функция Arc вывод эллептической дуги. Описание функции следующее:

```
BOOL Arc(
   HDC hdc,//дескриптор контекста устройства
    int nLeftRect,// x-координата левого верхнего угла прямоугольника
   int nTopRect,// y- координата левого верхнего угла прямоугольника
   int nRightRect,// x- координата правого верхнего угла прямоугольника
   int nBottomRect,// y- координата правого верхнего угла прямоугольника
   int nXStartArc,// x- координата точки начала дуги
   int nYStartArc,// y- координата точки начала дуги
```

```
  int nXEndArc,// x- координата конечной точки дуги
  int nYEndArc// y- координата конечной точки  дуги
);
```

Кривые Безье (подробно см. на www.Wikipedia.org) удобный инструмент параметрического задания кривых. Кривая Безье задается четырьмя точками (см. рис. 18): p0, p1, p2, p3.

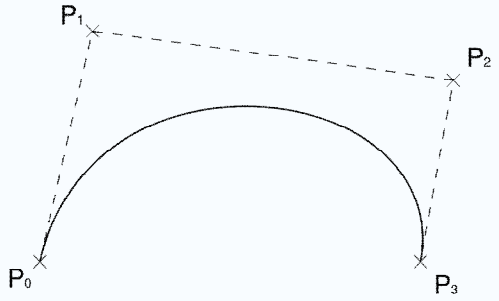


Рис. 18 — Кривая Безье

Точки p0 и p3 задают концы кривой, а точки p1 и p2 задают направления и кривизну кривой соединяющей точки p0 и p3. Изменяя положения точек p1 и p2 можно добиться построения нужной кривой. Для построения такой кривой в Windows используется функция PolyBezier. Описание этой функции следующее:

```
BOOL PolyBezier(
  HDC hdc,// дескриптор контекста устройства
  CONST POINT *lppt,//указатель на массив точек
  DWORD cPoints //число точек
);
```

Функция PolyBezier обеспечивает рисование кривой с установленным пером в контексте устройства. Число точек может быть больше чем 4, однако необходимо учитывать, что количество точек должно быть 4+3+…+3

3.11 Битовые карты (bitmap)

Битовые карты мощный инструмент Windows, предназначенный для представления и обработки растровых изображений. Битовая карта является одним из объектов контекста устройства. С точки зрения пользователя битовая карта представляет прямоугольник точек (pixel), которые формируют визуальное изображение. С точки зрения программиста это набор структур, состоящий из:

1) заголовка, в котором описано: разрешающая способность устройства отображения, размер прямоугольной области в пикселах, размер пиксела, размер массива и т.д.;

2) цветовая палитра;

3) массив бит, содержащих пикселы.

Существует два типа битовых карт: DDB — битовые карты, зависящие от устройства отображения и DIB- не зависящие от устройства. Рассмотрим структуры битовой карты формата DIB:

1) цветовая гамма устройства, в котором было создано прямоугольной изображение;

2) разрешение устройства, в котором создано изображение;

3) палитра устройства, в котором создано изображение;

4) массив (RGB) значений пикселов изображения;

5) указание на способ сжатия изображения.

Данная информация заносится в структуру BITMAPINFO, которая состоит из заголовка BITMAPINFOHEADER и нескольких массивов RGBQUAD. BITMAPINFO имеет описание

```
typedef struct tagBITMAPINFO { // bmi
  BITMAPINFOHEADER bmiHeader;
  RGBQUAD          bmiColors[1];
} BITMAPINFO;
```

Заголовок имеет следующее описание

```
typedef struct tagBITMAPINFOHEADER{ // bmih
  DWORD  biSize;  //размер структуры в байтах
  LONG   biWidth;   //ширина изображения в пикселах
  LONG   biHeight;  //высота изображения в пикселах
  WORD   biPlanes;  //описывает число плоскостей для EGA
  WORD   biBitCount //число бит на один пиксел
  DWORD  biCompression; //способ сжатия изображения
  DWORD  biSizeImage;  //размер изображения в байтах
  LONG   biXPelsPerMeter; //горизонтальное разрешение
  LONG   biYPelsPerMeter;  //вертикальное разрешение
  DWORD  biClrUsed;
  DWORD  biClrImportant;
} BITMAPINFOHEADER;
```

Структура RGBQUAD имеет описание:

```
typedef struct tagRGBQUAD { // rgbq
   BYTE    rgbBlue;  //голубой
   BYTE    rgbGreen; //зеленый
```

```
   BYTE    rgbRed;   //красный
   BYTE    rgbReserved; //зарезервировано
} RGBQUAD;
```

Описание структуры BITMAP

```
typedef struct tagBITMAP {  // bm
  LONG   bmType;  //0
  LONG   bmWidth; //Ширина изображения в пикселах
  LONG   bmHeight; //Высота изображение в пикселах
  LONG   bmWidthBytes; //число байт на сканированную строку точек
  WORD   bmPlanes; //число плоскостей
  WORD   bmBitsPixel; //число бит на пиксел
  LPVOID bmBits; //скан строки пикселов
} BITMAP;
```

Функция BitBlt производит копирование битовой карты из одного контекста устройства в ругой. Описание функции следующее:

```
BOOL BitBlt(    HDC hdcDest,// дескриптор контекста приемника
   int nXDest,// x- координата левого верхнего угла приемника
   int nYDest,// y- координата левого верхнего угла приемника
   int nWidth,// ширина прямоугольной области в пикселах
   int nHeight,//высота прямоугольной области в пикселах
   HDC hdcSrc,// дескриптор контекста источника
   int nXSrc,// x- координата левого верхнего угла источника
   int nYSrc,// y- координата левого верхнего угла источника
   DWORD dwRop //код растровой операции
  );
```

Код растровой операции может иметь следующие значения:

BLACKNESS
DSTINVERT
MERGECOPY
MERGEPAINT
NOTSRCCOPY
NOTSRCERASE
PATCOPY
PATINVERT
PATPAINT
SRCAND
SRCCOPY

SRCERASE
SRCINVERT
SRCPAINT

3.12 Работа с клавиатурой

Windows обеспечивает поддержку работы с клавиатурой независимо от производителя устройства и используемого языка. Это достигается за счет организации драйвера и специального интерфейса клавиатуры.

При нажатии клавиши драйвер читает код нажатой клавиши и передает в интерфейс клавиатуры. Последний на основе данного кода формирует виртуальный код клавиши, создает ряд сообщений и посылает их в очередь приложения.

Важным элементом взаимодействия интерфейса клавиатуры приложениями является фокус. Фокус определяет окно, которому будет переданы сообщения от клавиатуры. Фокусом клавиатуры обычно обладает активное окно, с которым работает пользователь. Активным окно становится при запуске приложения, при щелканье мышкой на заданном окне. При переключении между окнам с помощью комбинации клавиш ALT+TAB или ALT+ESC.

Когда фокус клавиатуры передается от одного окна к другому, то в первое окно передается сообщение WM_KILLFOCUS, а второму — WM_SETFOCUS.

При нажатии клавиш Windows создает сообщения WM_SYSKEYDOWN или WM_KEYDOWN.

Сообщение WM_SYSKEYDOWN передается в том случае, если была нажата комбинация клавиш ALT и другая клавиша, например, ALT F4, или ALT x и т.д. При обработке этого сообщения wParam содержит код виртуальной клавиши, а lParam — параметры драйвера клавиатуры (счетчик повторений, сканкод, флажки состояний и др.)

Сообщение WM_KEYDOWN передается в том случае, если была нажата клавиша или комбинация клавиш без ALT. В этом случае wParam содержит код виртуальной клавиши, а lParam — параметры драйвера клавиатуры (счетчик повторений, сканкод, флажки состояний и др.)

Сообщение WM_KEYUP передается в том случае, если была нажатая клавиша или комбинация клавиш без ALT пришла в исходной состояние (отпущена после нажатия). В этом случае wParam содержит код виртуальной клавиши, а lParam — параметры драйвера клавиатуры (счетчик повторений, сканкод, флажки состояний и др.)

Сообщение WM_SYSKEYUP передается в том случае, если комбинация клавиш с ALT пришла в исходной состояние (отпущена после нажатия). В этом случае wParam содержит код виртуальной клавиши, а

lParam — параметры драйвера клавиатуры (счетчик повторений, сканкод, флажки состояний и др.)

Сообщение WM_CHAR передается в том случае, если нажатая клавиша является символом (в отличии от управляющих клавиш DEL, HOME, INSERT, и т.д.) Тогда wParam содержит ASCII код символа, lParam — параметры драйвера клавиатуры (счетчик повторений, сканкод, флажки состояний и др.)

Ниже представлены два примера. Первый пример показывает, как использовать сообщение WM_KEYDOWN для обработки нажатий не символьных клавиш.

```
case WM_KEYDOWN:
        switch (wParam) {
           case VK_LEFT:    // Обработка клавиши "стрелка влево"

              break;

           case VK_RIGHT:  // Обработка клавиши "стрелка вправо"                .

              break;

           case VK_UP: :  // Обработка клавиши "стрелка вверх"

              break;

           case VK_DOWN:  // Обработка клавиши "стрелка вниз"

              break;

           case VK_F1:  // Обработка клавиши "F1"
              Help();
              break;
            .
           . //  Обработка других не символьных кодов
           .
            default:
              break;
        }
```

Этот пример показывает использование сообщения WM_CHAR для обработки ввода символов.

```
case WM_CHAR:
        switch (wParam) {
           case 0x08:  //Обработка клавиши "забой"
               break;
           case 0x0A:  // Обработка клавиши "перевод строки"
```

```
            break;

        case 0x1B:  // Обработка клавиши "ESC"

            break;

        default:  //обработка других символов ASCII кода
            break;
    }
```

3.13 Работа с мышкой

Манипулятор типа «Мышь» (далее мышка) является удобным устройством для работы с графическим интерфейсом ОС «Windows». Также как с клавиатурой, Windows для мышки имеет драйвер и программный интерфейс, которой не зависит от производителя устройства. Обычно, мышка имеет три клавиши (левая, средняя и правая). Передвигая мышку по поверхности (коврика, стола и пр.) мы перемещаем специальный курсор по экрану терминала.

Для создания приложения работающего с мышкой необходимо знать положение курсора, а также, какая клавиша мышки при этом нажата. Для этого в Windows предусмотрены соответствующие сообщения.

1. WM_MOUSEMOVE

Это сообщение передается в приложение когда в окне приложение находится курсор и он изменяет сое положение. При этом wParam содержит флажки, а lParam содержит положение курсора. Причем:

```
xPos = LOWORD(lParam);  // горизонтальная позиция курсора
yPos = HIWORD(lParam);  // вертикальная позиция курсора
```

Значение wParam может быть комбинацией следующих флажков:
MK_CONTROL установлен, если нажата клавиша CTRL.
MK_LBUTTON установлен, если нажата левая клавиша мышки.
MK_MBUTTON установлен, если нажата средняя клавиша мышки.
MK_RBUTTON установлен, если нажата правая клавиша мышки.
MK_SHIFT установлен, если нажата клавиша SHIFT.

Ниже представлена таблица сообщений для обработки нажатий и отжатий клавиш мышки.

WM_LBUTTONDBLCLK	Двойной щелчок левой клавиши
WM_LBUTTONDOWN	Нажата левая клавиша мышки
WM_LBUTTONUP	Левая клавиша мышки отжата
WM_MBUTTONDBLCLK	Двойной щелчок средней клавиши
WM_MBUTTONDOWN	Нажата средняя клавиша мышки
WM_MBUTTONUP	Средняя клавиша мышки отжата

WM_RBUTTONDBLCLK	Двойной щелчок правой клавиши
WM_RBUTTONDOWN	Нажата правая клавиша мышки
WM_RBUTTONUP	Правая клавиша мышки отжата

В каждом перечисленном сообщении wParam содержит флажки, описанные выше (см. WM_MOUSEMOVE). А lParam содержит положение курсора.

```
xPos = LOWORD(lParam);  // горизонтальная позиция курсора
yPos = HIWORD(lParam);  // вертикальная позиция курсора
```

Примечание. Для того чтобы обрабатывать двойные щелчки надо в оконном классе необходимо установить стиль CS_DBLCLKS.

3.13.1 Пример рисование линий (эффект резиновой нити)

Рассмотрим пример рисование линий с помощью курсора мышки. Когда курсор мышки передвинут в клиентскую часть окна и нажата левая клавиша, в оконную процедуру посылается сообщение WM_LBUTTONDOWN. Обрабатывая это сообщение, происходи захват мышки, координаты курсора сохраняются, которые становятся координатами начальной точки рисования линии.

При движении курсора с «придавленной» левой клавишей в оконную процедуру посылается сообщение WM_MOUSEMOVE. Если это сообщение принимается первый раз, то рисуется линия между начальной и текущей точками и координаты текущей точки запоминаются. В последующие изменения координат мышки

предыдущая линия затирается и рисуется линия с новыми координатами курсора. Таки образом, линия всегда следует за курсором и тем самым достигается эффект рисования резиновой нити.

```
LRESULT APIENTRY MainWndProc(HWND  hwndMain, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
   HDC hdc;                  //  дескриптор контекста устройства
   RECT rcClient;              // размеры клиентской части окна
   POINT ptClientUL;             // координаты левого угла клиентской части окна
   POINT ptClientLR;             // координаты правого угла клиентской части окна

   static POINTS ptsBegin;       //начальная точка

   static POINTS ptsEnd;         // новая текущая точка
   static POINTS ptsPrevEnd;     / / предыдущая текущая точка
   static BOOL fPrevLine = FALSE;  //флажок наличия прямой в окне
```

```
   switch (uMsg) {
      case WM_LBUTTONDOWN:  //нажата левая клавиша мышки
            SetCapture(hwndMain);  // захват мышки


         GetClientRect(hwndMain, &rcClient);  //определяем начальные координаты
         ptClientUL.x = rcClient.left;
         ptClientUL.y = rcClient.top;


         ptClientLR.x = rcClient.right + 1;
         ptClientLR.y = rcClient.bottom + 1;

         ClientToScreen(hwndMain, &ptClientUL);
         ClientToScreen(hwndMain, &ptClientLR);


         SetRect(&rcClient, ptClientUL.x, ptClientUL.y,
            ptClientLR.x, ptClientLR.y);

         ClipCursor(&rcClient);


         ptsBegin = MAKEPOINTS(lParam);  //устанавливаем начальные координаты
линии
         return 0;

      case WM_MOUSEMOVE:  //перемещаем курсор мышки
            if (wParam & MK_LBUTTON) {  //если левая клавиша нажата

            hdc = GetDC(hwndMain);  //получает дескриптор контекста устройства

            SetROP2(hdc,  R2_NOTXORPEN); //устанавливаем режим рисования линий
инвертированием

            if (fPrevLine) {  //предыдущая линия есть
               MoveToEx(hdc, ptsBegin.x, ptsBegin.y,  //стираем ее используя режим
инвертирования
                  (LPPOINT) NULL);
               LineTo(hdc, ptsPrevEnd.x, ptsPrevEnd.y);

            }

            ptsEnd = MAKEPOINTS(lParam);  //рисуем новую линию
            MoveToEx(hdc, ptsBegin.x, ptsBegin.y,
               (LPPOINT) NULL);
            LineTo(hdc, ptsEnd.x, ptsEnd.y);
```

```
            fPrevLine = TRUE;  //линия на экране есть
            ptsPrevEnd = ptsEnd;  //запоминаем текущую позицию
            ReleaseDC(hwndMain, hdc);  //закрываем контекст устройства
        }
        break;

    case WM_LBUTTONUP:  //левая клавиша мышки отпущена


        fPrevLine = FALSE;    //линии на экране нет
        ClipCursor(NULL);     //отпускаем мышку
        ReleaseCapture();
        return 0;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    .
    . // обработка других сообщений
    .
}
```

3.14 Ресурсы

3.14.1 Основные понятия

Приложение в Windows часто содержат разнообразные изображения, пиктограммы, тексты сообщений, курсоры, шрифты и пр., которые могут занимать значительный объем ОП. Важным свойством таких данных в том, что они не изменяются. Для хранения и манипулирования такими данными в системе предусмотрен механизм подключения ресурсов. Таким образом, ресурс приложения это блок неизменяемых данных, которые необходимо приложению в процессе работы. В системе ресурсы хранятся в виде отдельного модуля, описание ресурсов производятся на специальном скриптовом языке. Каждый ресурс имеет имя и тип. Имя ресурса представляет собой идентификатор и некоторое целое беззнакове число. Тип ресурса задает способ загрузки и обработки с помощью API-функций. В ОС Windows зарегистрировано некоторое множество типов: иконки, курсоры, растровые изображения, меню, ускорители, диалоговые окна, таблица строк символов и пр. В системе также предусмотрен механизм хранения и использования ресурса, определенного прикладным программистом. Для создания и использования ресурсов в приложении необходимо выполнить два шага:

1. Компоновка ресурсов при создании приложения.
2. Использование ресурсов в процессе работы приложения.

На первом шаге необходимо подготовить данные, которые будут храниться в ресурсе. Например, часть ресурсов может быть представлена в файлах (изображения, пиктограммы, шрифты, другие пользовательские ресурсы). Остальные могут быть представлены в файле описания ресурса. Далее все ресурсы описываются в файле ресурса (текстовый файл с расширением rc). Каждому ресурсу назначается имя и указывается соответствующий тип. Если ресурс не стандартный, то его тип назначает сам программист.

После того как описание ресурсов готово, текстовый файл описаний ресурсов подключается к проекту приложения. Далее при формирования приложения, компилятор ресурсов производит разбор описания ресурсов и формирует объектный код ресурсов. Затем компановщик, при создании приложения, подключает ресурсы.

Использование ресурсов производится с помощью API-функций, которые обеспечивают: поиск, загрузку, фиксацию ресурсов, и выделение и утилизацию оперативной памяти, занимаемой ресурсами.

3.14.2 Язык описания ресурса

Язык описания ресурсов довольно простой язык не имеющий ни циклов не ветвлений. И состоит из последовательности:

<имя ресурса> <тип ресурса> <файл ресурса или непосредственное описание>

В файл описания ресурсов можно использовать директивы препроцессора. Например, #include или #define.

Структура описания ресурса зависит от типа ресурса. Предусмотрены следующие типы: простые, составные и типы, определенные программистом.

Простые типы

Рисунок

BITMAPтип ресурса для хранения рисунков. Рисунок в формате BMP должен храниться в файле. Правила записи:

Имя_ресурса BITMAP имя_файла

Пример

```
disk1   BITMAP disk.bmp
12      BITMAP PRELOAD diskette.bmp
```

Курсор

CURSOR — тип ресурса предназначенный для представления изображения курсора, предварительно должен быть записан в файле с расширением CUR. Правила записи

Имя_ресурса CURSOR имя_файла

Например,

```
cursor1 CURSOR bullseye.cur
2       CURSOR "d:\\cursor\\arrow.cur"
```

Шрифт

FONT — тип ресурса для преставления шрифта, шрифт предварительно должен быть записан в файле. Правила записи:

Имя_ресурса FONT имя_файла

ghbvth

5 FONT CMROMAN.FNT

Иконка

ICON — тип ресурса предназначен для представления иконки приложения. Иконка предварительно должна быть записана в файл. Правила записи следующие:

Имя_ресурса ICON имя_файла

Например,

```
desk1   ICON desk.ico
11      ICON DISCARDABLE custom.ico
```

Составные типы

DIALOG тип определяет управляющие элементы диалогового окна.
MENU тип определяет описание меню.
RCDATA определяет двоичные данные.
STRINGTABLEтип, определяет таблицу строк.

1. Ускорители

Тип ACCELERATORSпредназначен для представления коды клавиш быстрого доступа. Правила записи описаний ускорителей следующие:

Имя_ресурса ACCELERATORS

BEGIN
event, **idvalue**, [**type**] [**options**]
. . .
END

event — описывает клавишу или комбинацию клавиш, иммется следующие варианты записи:

1) «символ»;
2) «виртуальный код»;
3) целое значение.

idvalue — идентификатор, который передается в оконную процедуру при нажатие этой клавиши или комбинации клавиш.

type имеет два значения ASCII или VIRTKEY

Опции **(options)**

ALT — сочетание нажатием клавишей ALT.
SHIFT — сочетание нажатием клавишей SHIFT.
CONTROL — сочетание нажатием клавишей CONTROL.

Например,

```
1 ACCELERATORS
BEGIN
 "^C",  IDDCLEAR        ; control C
 "K",   IDDCLEAR        ; shift K
 "k",   IDDELLIPSE, ALT  ; alt k
 98,    IDDRECT, ASCII   ; b
 66,    IDDSTAR, ASCII   ; B (shift b)
```

```
  "g",   IDDRECT         ; g
  "G",   IDDSTAR         ; G (shift G)
  VK_F1, IDDCLEAR, VIRTKEY              ; F1
  VK_F1, IDDSTAR, CONTROL, VIRTKEY      ; control F1
  VK_F1, IDDELLIPSE, SHIFT, VIRTKEY     ; shift F1
  VK_F1, IDDRECT, ALT, VIRTKEY          ; alt F1
  VK_F2, IDDCLEAR, ALT, SHIFT, VIRTKEY  ; alt shift F2
  VK_F2, IDDSTAR, CONTROL, SHIFT, VIRTKEY ; ctrl shift F2
  VK_F2, IDDRECT, ALT, CONTROL, VIRTKEY   ; alt control F2
END
```

2. Описание ресурса диалогового окна

Тип ресурса DIALOG предназначен для описания диалогового окна и его управляющих элементов. Правила записи этого ресурса следующие.

```
<имя_ресурса> DIALOG [ load-mem] x, y, width, height
[optional-statements]
BEGIN
   control-statement
   . . .
END
```

Параметры:

1. Имя ресурса.
2. load-mem — параметры загрузки и класса памяти.
3. Описание параметров окна.

CAPTION "text" — заголовок.
CHARACTERISTICS dword — флажки.
CLASS class — оконный класс.
STYLE styles — стиль окна.

Описание управляющих элементов (control-statement) задается одним общим правилом:

CONTROL text, id, class, style, x, y, width, height

где CONTROL принимает следующие значения

AUTOCHECKBOX
AUTORADIOBUTTON
CHECKBOX

COMBOBOX
CTEXT
DEFPUSHBUTTON
EDITTEXT
GROUPBOX
LISTBOX
PUSHBUTTON
RADIOBUTTON

text — текст заголовка;
id, — идентификатор объекта управления;
class — класс объекта;
style, — стиль объекта;
x, y, — координаты на окне;
width, height — ширина и высота элемента.

Пример, описание диалогового окна в ресурсном файле:

```
#include <windows.h>

ErrorDialog DIALOG  10, 10, 300, 110
STYLE WS_POPUP|WS_BORDER
CAPTION "Error!"
BEGIN
   CTEXT "Select One:", 1, 10, 10, 280, 12
   PUSHBUTTON "&Retry", 2, 75, 30, 60, 12
   PUSHBUTTON "&Abort", 3, 75, 50, 60, 12
   PUSHBUTTON "&Ignore", 4, 75, 80, 60, 12
END
```

3. Описание ресурса меню

Ресурс меню предназначен для преставления меню приложения. Правила записи ресурса меню следующие:

```
menuID MENU [load-mem]
[optional-statements]
BEGIN
   item-definitions
   . . .
END
```

где item-definitions может принимать следующие значения:

1) MENUITEM
2) MENUITEM SEPARATOR
3) POPUP

Элемент MENUITEM предназначен для описания конечного пункта меню и имеет следующий синтаксис:

```
MENUITEM  text, result, [optionlist]
```

Здесь
text — обозначат, текст содержащий описание элемента меню;
result — целое значение, передающееся в оконную процедуры при выборе данного пункта в меню;
optionlist — указывает на некоторые опции меню:

CHECKED — это пункт меню при выборе будет отмечен.
GRAYED — означает, что будет изменен цвет пункта.
HELP — это помощь.
INACTIVE — пункт виден но не может быть выбран.
SEPARATOR — разделитель

Пример описания ресурса меню.

```
MENUITEM "&Roman", 206, CHECKED, GRAYED
MENUITEM SEPARATOR
MENUITEM "&Blackletter", 301
```

Всплывающее меню

Всплывающее меню имеет синтаксис такой же как и MENU

```
POPUP text, [optionlist]

BEGIN
   item-definitions
   .
   .
   .
END
```

Тогда общий пример описания меню в ресурсе будет следующий:

```
chem MENU
BEGIN
   POPUP "&Elements"
   BEGIN
      MENUITEM "&Oxygen", 200
      MENUITEM "&Carbon", 201, CHECKED
      MENUITEM "&Hydrogen", 202
      MENUITEM SEPARATOR
      MENUITEM "&Sulfur", 203
      MENUITEM "Ch&lorine", 204
   END
   POPUP "&Compounds"
   BEGIN
      POPUP "&Sugars"
      BEGIN
        MENUITEM "&Glucose", 301
        MENUITEM "&Sucrose", 302, CHECKED
        MENUITEM "&Lactose", 303, MENUBREAK
        MENUITEM "&Fructose", 304
      END
      POPUP "&Acids"
      BEGIN
          "&Hydrochloric", 401
          "&Sulfuric", 402
      END
   END
END
```

Основное меню будет содержать два всплывающих окна «&Elements» и «&Compounds».

Первое всплывающее меню содержит 6 элементов, один из которых разделитель. Второе всплывающее меню содержит еще два всплывающих окна.

Ресурсы, определяемые пользователем

Программист в ресурсном файле может организовать описание собственных ресурсов, двумя способами:

1) используя тип RCDATA;
2) используя свой собственный тип.

Для использования RCDATA необходимо использовать следующие правила:

```
nameID RCDATA [load-mem]
[optional-statements]
BEGIN
```

```
   raw-data
   . . .
END
```

где raw-data описание строк символов и целых чисел.

Например,

```
resname RCDATA
BEGIN
  L"Here is a Unicode string\0",
  1024,              /* int       */
  0x029a,            /* hex int    */
  0o733,             /* octal int  */
  "\07"              /* octal byte */
END
```

Программист может не использовать RCDATA, а написать свой тип русурса. Правила записи в этом случае:

nameID typeID [load-mem] filename

Например,

Massiv MPOINTS точки.bin

Ресурс таблица строк символов STRINGTABLE

Таблица строк символов **STRINGTABLE** может быть только одна, поэтому синтаксис этой таблицы следующий:

```
STRINGTABLE [load-mem]
[optional-statements]
BEGIN
   stringID string
   . . .
END
```

где stringID — идентификатор строки, целое число.

string — строка символов, длинной не более 255.

Пример,

```
#define IDS_HELLO    1
#define IDS_GOODBYE  2

STRINGTABLE
BEGIN
```

```
   IDS_HELLO,   "Hello"
   IDS_GOODBYE, "Goodbye"
END
```

ПРИМЕЧАНИЕ.

В тех случаях, когда вместо имени ресурса ставится номер, то в API-функциях, вместо номера ресурса используется следующий макрос

```
LPTSTR MAKEINTRESOURCE(WORD wInteger);
```

Определение которого следующее

```
#define MAKEINTRESOURCE(i)  (LPTSTR) ((DWORD) ((WORD) (i)))
```

Например,

```
LoadCursor(hInsatnce, MAKEINTRESOURCE(10) );
```

Здесь 10 — константа, обозначающая имя ресурса курсора

3.14.3 Использование ресурсов

Для использования ресурсов в программе имеется набор API-функций. Ниже перечислены основные функции для загрузки стандартных ресурсов:

LoadAccelerators	загрузка таблицы акселераторов
LoadBitmap	загрузка изображения
LoadCursor	загрузка курсора
LoadIcon	загрузка иконки
LoadMenu	загрузка меню
LoadString	загрузка строки

После использования ресурсов производится утилизация памяти. Ниже перечислены API-функции для утилизации памяти соответствующих типов ресурсов.

Ресурс	Функция удаления ресурса
Таблица акселераторов	DestroyAcceleratorTable
Изображение	DeleteObject
Курсор	DestroyCursor
Иконка	DestroyIcon
Меню	DestroyMenu

Наиболее часто ресурсы загружаются в оперативную память при обработке сообщения WM_CTREATE. Соответственно, утилизируются при обработке сообщения WM_DESTROY.

Рассмотрим подробнее описание функции загрузки, все функции принимают в качестве входного параметра дескриптор приложения (HINSTANCE) или дескриптор модуля (HMODULE).

1. Загрузить строку из ресурса

int LoadString(HINSTANCE hInstance, UINT uID, LPTSTR lpBuffer, int nBufferMax);

производится загрузка строки из ресурса таблица строк, где uID- номер строки: lpBuffe — буфер куда строка будет помещена: nBufferMax — размер буфера.

2. Загрузить курсор из ресурса

HCURSOR LoadCursor(HINSTANCE hInstance, LPCTSTR lpCursorName);

lpCursorName — имя ресурса, содержащего курсор.

При выполнении этой функции, ОС по имени ищет ресурс, если находит, загружает его, создает объект «Cursor» и возвращает дескриптор.

3. Загрузить изображение

HBITMAP LoadBitmap(HINSTANCE hInstance, LPCTSTR lpBitmapName);

При выполнении этой функции, ОС по имени ищет ресурс, если находит, загружает его, создает объект «Bitmap» и возвращает дескриптор.

4. Загрузить таблицу акселераторов

HACCEL LoadAccelerators(HINSTANCE hInstance, LPCTSTR lpTableName);

При выполнении этой функции, ОС по имени ищет ресурс, если находит, загружает его, создает объект «Accel» и возвращает дескриптор.

5. Загрузить меню

HMENU LoadMenu(HINSTANCE hInstance, LPCTSTR lpMenuName);

При выполнении этой функции, ОС по имени ищет ресурс, если находит, загружает его, создает объект «Menu» и возвращает дескриптор.

6. Загрузить иконку

HICON LoadIcon(HINSTANCE hInstance, LPCTSTR lpIconName);

При выполнении этой функции, ОС по имени ищет ресурс, если находит, загружает его, создает объект «Icon» и возвращает дескриптор.

7. Загрузка пользовательских ресурсов

Ресурсы, созданные прикладным программистом, грузятся с помощью трех функций:

1) FindResource — найти ресурс с заданным именем и типом:
2) LoadResource — загрузить найденный ресурс;
3) LockResource — получить указатель на загруженный ресурс.

Первая функция имеет следующее описание:

HRSRC FindResource(HMODULE hModule, LPCTSTR lpName, LPCTSTR lpType);

hModule — дескриптор модуля или приложения, содержащего требуемый ресурс;

lpNam — имя ресурса;

lpType — тип ресурса.

Если функция находит ресурс, то она возвращает актуальный дескриптор ресурса (HRSRC).

Функция загрузки нестандартного ресурса имеет описание:

HGLOBAL LoadResource(HMODULE hModule, HRSRC hResInfo);

где hResInfo — дескриптор найденного ресурса.

Функция загружает указанный ресурс в виртуальную память и возвращает указатель типа HGLOBAL.

Функция LockResource по указателю на виртуальную память закрепляет найденный ресурс в оперативной памяти и возвращает адрес блока паяти в котором находится требуемый ресурс. Описание функции следующее:

LPVOID LockResource(HGLOBAL hResData);

где hResData — указатель на виртуальную память. Функция возвращает указатель типа VOID. Для использования этого указателя необходимо явно преобразовать к указателю с заданным описанием структуры.

Рассмотрим пример,

```
struct  Exmp { int x; int y; char s[100]; };
...
HGLOBAL   hExmp=LoadResourse(hInstance, FindResource(hInstance,"srct1",
"EXMP" ));
...
struct  Exmp  *p_exmp=( struct  Exmp  *) LockResource(hExmp);
...
OutText(p_exmp->x, p_exmp->y, p_exmp->s,100);
```

Описание ресурса в ресурсном файле следующее:

```
srct1   EXMP       src1.bin
```

где src1.bin — файл в котором хранится два целых числа и строка длинной 100 байт.

Фрагмент, описанный выше показывает, как надо создавать и загружать пользовательский ресурс.

3.15 Процесс обработки WM_COMMAND

Сообщение WM_COMMAND служит для указания, что пользователь приложения выбрал один из пунктов меню, щелкнул мышкой на некотором стандартном элементе управления, например, на кнопке, или нажал комбинацию клавиш, которая описана в таблице акселераторов. Основные параметры, которые передаются с эти сообщением следующие:

```
wNotifyCode = HIWORD(wParam); // код дополнительного уведомления
wID = LOWORD(wParam);    // идентификатор элемента управления
hwndCtl = (HWND) lParam;    // дескриптор элемента управления
```

При создании каждого элемента управления необходимо указать дескриптор родительского окна и идентификационный номер этого элемента управления (См. описание CreateWindow). Когда пользователь выбирает один из пунктов меню, щелкаем мышкой на элементе управления или нажимает набор акселераторов, в родительское окно посылается сообщение WM_COMMAND с индетификационным номером и дополни-

тельной информацией. Обычно код для обработки сообщения WM_COMMAND следующий:

```
case WM_COMMAND:
{
   switch(LOWORD(wParam)) {
case CM_ABOUT:  //код выполнения команды ABOUT
      break;
case CM_COMMAND1: // код выполнения команды COMMAND1
      break;
case ID_LISTBOX: //код выполнения команды при выборе элемента из перечня
      break;
…
      }
}
```

где CM_ABOUT, CM_COMMAND1, ID_LISTBOX определены с помощью define в заголовочном файле. А затем связаны с элементами меню (см. описание меню в ресурсном файле) или указаны при создании элементов управления (см. CreateWindow).

Ели оконная процедура обрабатывает это сообщение, то она должна вернуть 0.

3.16 Стандартные элементы управления (Control)

Для создания механизма управления и ввода информации в Windows зарегистрировано несколько стандартных оконных классов. Это классы для создания различных кнопок, окон ввода, перечней, линеек прокрутки, статических полей.

Все стандартные элементы управления в окне создаются с помощью функции CreateWindow, как дочерние окна. Если управляющий элемент служит для указания некоторого действия или команды, то необходимо указать идентификационный номер, который будет передан вместе с WM_COMMAND.

Для диалоговых окон описание стандартных элементов может быть дано в ресурсном файле (см. описание ресурса DIALOG).

Процесс взаимодействия с элементом управления производится с помощью функций PostMessage и SendMessage. Описание PostMessage следующее:

```
BOOL PostMessage(
   HWND hWnd,//дескриптор окна или элемента управления
```

```
  UINT Msg,// сообщение, которое надо послать
  WPARAM wParam,// wParam, если необходимо, если нет то NULL
  LPARAM lParam // lParam, если необходимо, если нет то NULL
 );
```

Функция PostMessage обеспечивает передачу сообщения в очередь сообщений приложения.

Описание функции SendMessage следующее:

```
LRESULT SendMessage(
HWND hWnd,//дескриптор окна или элемента управления
  UINT Msg,// сообщение, которое надо послать
  WPARAM wParam,// wParam, если необходимо, если нет то NULL
  LPARAM lParam // lParam, если необходимо, если нет то NULL
 );
```

Функция SendMessage вызывает оконную процедуру для дескриптора hWnd и ждет завершения обработки этого сообщения вызванной оконной процедурой.

Функция SendMessag позволяет организовать передачу и извлечение данных для оконных процедур. Например, если оконная процедура возвращает число или адрес, то SendMessage возвращает это значение в точке вызова:

```
addr= SendMessage(hwnd,msg,wParam,lParam);
```

Кроме того, wParam и/или lParam может содержать адреса структур, которые могут быть заполнены в оконной процедуре.

3.16.1 Кнопки (Buttons)

Клавиши в Windows могут быть следующих типов: кнопки нажатия (Push Buttonы), Отмечаемые блоки — флажки (Check Boxes), переключатели (Radio Buttons). Для создания клавиши необходимо в CreateWindow указать имя оконного класса «BUTTON» и установить стиль клавиши:

BS_CHECKBOX	Создается отмечаемый блок, маленькое окно, в котором отметка производится в виде «птицы»
BS_DEFPUSHBUTTON	Стандартная клавиша по умолчанию. Если нажата клавиша Enter, то будет выбрана эта клавиша

BS_GROUPBOX	Объединение группы элементов управления
BS_LEFTTEXT	Places text on the left side of the radio button or check box when combined with a radio button or check box style. Same as the BS_RIGHTBUTTON style
BS_OWNERDRAW	Creates an owner-drawn button. The owner window receives a WM_MEASUREITEM message when the button is created and a WM_DRAWITEM message when a visual aspect of the button has changed. Do not combine the BS_OWNERDRAW style with any other button styles
BS_PUSHBUTTON	Стандартная кнопка
BS_RADIOBUTTON	Создается маленький круг с текстов слева или справа
BS_BITMAP	Указывает, что на клавише будет изображение

3.16.2 Статические поля

Статические поля предназначены для представления некоторой информации в окне, обычно это некоторый текст перед или между другими стандартными элементами управления. Статические поля не мог быть выбраны или отмечены и имеют зарегистрированный оконный класс «STATIC». При создании статического поля в функции CreateWindow необходимо указать класс «static», и стиль, значения которого записаны в следующей таблице.

SS_BITMAP	Будет содержать изображение
SS_BLACKFRAME	Будет иметь черную рамку
SS_CENTER	Текст будет отформатирован по центу
SS_CENTERIMAGE	Изображение будет в центре прямоугольника клавиши
SS_LEFT	Текст будет отформатирован относительно левой границы прямоугольника
SS_NOTIFY	Посылка родительскому окну STN_CLICKED, STN_DBLCLK, STN_DISABLE и STN_ENABLE уведомлений
SS_OWNERDRAW	Указывает, что на статическом поле

	будет выведена информации по запросу. Запрос выдается в форме подачи родительскому окну сообщения WM_DRAWITEM
SS_RIGHT	Форматирование текста относительно правой границы прямоугольника
SS_SIMPLE	Простое статическое поле, с выводом текста

3.16.3 Редактируемые поля (Edit)

Редактируемые поля предназначены для ввода текстовой информации с использование клавиатуры. Редактируемое поле представляет собой простейший редактор строки символов. Для создания редактируемого поля необходимо в CreateWindow записать имя оконного класса «edit», и указать стиль:

ES_MULTILINE	В поле может быть несколько строк
ES_LEFT	Вывод текста относительно левой границы поля
ES_CENTER	Вывод текста по центу поля
ES_RIGHT	Вывод текста относительно правой границы
ES_AUTOHSCROLL	Автоматически прокручивает в горизонтальном направлении
ES_AUTOVSCROLL	Автоматически прокручивает в вертикальном направлении
ES_PASSWORD	Вводит символы в режиме «пароль»

Для установки и извлечения информации из редактируемого поля существует множество сообщений:

WM_SETFONT	
EM_CANUNDO	Возвращает TRUE если редактируемое поле может отменить последнее действие
EM_GETLINE	Копирует строку редактирования в заданный буфер
EM_GETLINECOUNT	Возвращает число строк
EM_GETMODIFY	Проверяет изменилось ли содержание строки

EM_GETRECT	Возвращает координаты редактируемого поля
EM_GETSEL	Получает позицию текущего выбранного фрагмента
EM_LINELENGTH	Возвращает длину строки
EM_REPLACESEL	Заменить текущий выбранный фрагмент строки
EM_SETMODIFY	Устанавливает или сбрасывает флаг модификации
WM_COPY	Копировать текст из буфера обмена (clipboard)
WM_GETFONT	Вернуть дескриптор шрифта
WM_GETTEXT	Копировать текст в указанный буфер
WM_GETTEXTLENGTH	Вернуть длину текста
WM_SETTEXT	Копировать строку символов из указанного буфера в буфер редактируемого поля

3.16.4 Перечни

Перечень — стандартный элемент управления, представленный в виде специального окна, в котором вертикально перечислены строки и изображения. Перечень предназначен для просмотра и выбора одного или нескольких элементов. Для организации перечня в окне существует оконный класс, зарегистрированный по имени «listbox». Перечень создается с помощью функции CreateWindow, дополнительный свойства перечня устанавливаются с помощью флажков в параметре style. Возможные флажки и их свойства перечислены в следующей таблице:

LBS_EXTENDEDSEL	Разрешает многоэлементный выбор с помощью нажатия клавиши SHIFT
LBS_MULTIPLESEL	Позволяет выбирать более одного элемента в перечне одновременно
LBS_NOSEL	Запрет выбора элементов
LBS_SORT	Отсортировать элементы в алвавитном порядке
LBS_STANDARD	Стандартный перечень

Взаимодействие с перечнем производится с помощью функции SendMessage, при этом сообщения могут быть следующие:

LB_ADDSTRING	Вставить стоку в перчень
LB_DELETESTRING	Удалить строку из перечня
LB_FINDSTRING	Найти строку в перечне
LB_GETCOUNT	Вернуть число элементов в перечне
LB_GETCURSEL	Вернуть номер выбранного элемента
LB_GETSEL	Вернуть состояние (выбран/невыбран) элемента
LB_GETSELCOUNT	Вернуть число выбранных элементов
LB_GETTEXT	Копирует элемент перечня в буфер
LB_GETTEXTLEN	Получает длину элемента перечня
LB_INSERTSTRING	Вставляет строку в перечень
LB_RESETCONTENT	Удаляет все элементы из перечня
LB_SELECTSTRING	Выбрать первый найденный элемент перечня
LB_SETCOUNT	Установить число элементов в перечне
LB_SETCURSEL	Выбирает элемент перечня по номеру
LB_SETSEL	Выбирает элемент перечня с множественным выбором

3.16.5 Линейки прокрутки

Линейка прокрутки — стандартный элемент управления Windows, предназначенный для просмотра информации, которая не помещается в текущие размеры окна. Линейки прокрутки существуют горизонтальные и вертикальные.

Для того чтобы встроить линейки прокрутки в окно необходимо выполнить следующие действия:

1. Создать окно, используя стили WS_VSCROLL и WS_HSCROLL.
2. Установить диапазон изменения с помощью функции SetScroll-Range.
3. Написать код для обработки сообщений WM_VSCROLL и WM_HSCROLL.

Рассмотрим пример для прокрутки текста в окне

```
HDC hdc;
PAINTSTRUCT ps;
TEXTMETRIC tm;
SCROLLINFO si;
static int xClient;     // ширина клиентской части окна
static int yClient;     // высота клиентской части окна
```

```
static int xClientMax;  //  максимальная ширина

static int xChar;       // ширина символа
static int yChar;       // высота символа
static int xUpper;      // Ширина заглавной буквы

static int xPos;        // текущая позиция горизонтального скроллинга
static int yPos;        // текщая позиция вертикального скроллинга
static int xMax;        // максимальная позиция горизонтального скроллинга
static int yMax         // максимальная позиция вертикального скроллинга

int xInc;               // приращение горизонтального скроллинга
int yInc;               // приращение вертикального скроллинга

int i;
int x, y;              // текущие координаты вывода

int FirstLine;         // первая строка в окне
int LastLine;          // последняя строка в окне

//массив строк для организации демонстрации прокрутки
#define LINES 27
static char *abc[] = { "яблоко", "апельсин", "мандарин", "киви",
  "хурма", "слива", "груша", "гранат", "лимон", "грейпфрут",
  "ранетка", "банан", "кокос", "манго", "маранг", "папайя",
  "ананас", "физалис", "тамаринд", "лайм", "кизил", "инжир",
  "виноград", "олива", "маракуйя", "айва",
  "рябина" };

switch (uMsg) {
     case WM_CREATE :
        hdc = GetDC (hwnd);       //открыть контекст устройства
        GetTextMetrics (hdc, &tm);  //получить метрику шрифта

       xChar = tm.tmAveCharWidth; //установить ширину символа
       xUpper = (tm.tmPitchAndFamily & 1 ? 3 : 2) * xChar/2;  //установить ширину про-
писных букв
       yChar = tm.tmHeight + tm.tmExternalLeading;  //установить высоту букв
      ReleaseDC (hwnd, hdc);  //закрыть контекст устройства

       xClientMax = 48 * xChar + 12 * xUpper;  //установить максимальную ширину
       return 0;

   case WM_SIZE: //установить размеры клиентской части окна
       yClient = HIWORD (lParam);
       xClient = LOWORD (lParam);

  //установить максимальное значение вертикальной позиции
       yMax = max (0, LINES + 2 - yClient/yChar);
 //установить текущее занчение вертикальной позиции
```

```
        yPos = min (yPos, yMax);
 //установить параметры вертикальной прокрутки
        si.cbSize = sizeof(si);
        si.fMask  = SIF_RANGE | SIF_PAGE | SIF_POS;
        si.nMin   = 0;
        si.nMax   = yMax;
        si.nPage  = yClient / yChar;
        si.nPos   = yPos;
        SetScrollInfo(hwnd, SB_VERT, &si, TRUE);


//аналогично установить параметры горизонтальной прокрутки
        xMax = max (0, 2 + (xClientMax - xClient)/xChar);
        xPos = min (xPos, xMax);
         si.cbSize = sizeof(si);
        si.fMask  = SIF_RANGE | SIF_PAGE | SIF_POS;
        si.nMin   = 0;
        si.nMax   = xMax;
        si.nPage  = xClient / xChar;
        si.nPos   = xPos;
        SetScrollInfo(hwnd, SB_HORZ, &si, TRUE);
         return 0;

    case WM_PAINT:

        hdc = BeginPaint(hwnd, &ps);

        //определяем номера начально и конечной строк для вывода
        FirstLine = max (0, yPos + ps.rcPaint.top/yChar - 1);
        LastLine = min (LINES, yPos + ps.rcPaint.bottom/yChar);

      //вывод линий

        for (i = FirstLine;i < LastLine;i++) {
           x = xChar * (1 - xPos);
           y = yChar * (1 - yPos + i);
           TextOut (hdc, x, y, abc[i], lstrlen(abc[i]));
           }

        EndPaint(hwnd, &ps);
        break;

    case WM_HSCROLL:    //обработка сообщения горизонтальной прокрутки
        switch(LOWORD (wParam)) {

            case SB_PAGEUP: //обработка щелчка слева от бегунка

               xInc = -8;
               break;
```

```
        case SB_PAGEDOWN::  //обработка щелчка справа от бегунка
            xInc = 8;
            break;

        case SB_LINEUP: :  //обработка щелчка на  клавише «стрелка влево» горизонтальной прокрутки
            xInc = -1;
            break;


        case SB_LINEDOWN: //обработка щелчка на  клавише «стрелка вправо» горизонтальной прокрутки

            xInc = 1;
            break;

        case SB_THUMBTRACK:  //обработка перемещения бегунка
            xInc = HIWORD(wParam) - xPos;

            break;
         default:
            xInc = 0;

      }

//реализация новой позиции бегунка
      if (xInc = max (-xPos, min (xInc, xMax - xPos))) {
         xPos += xInc;

         ScrollWindowEx (hwnd, -xChar * xInc, 0,
            (CONST RECT *) NULL, (CONST RECT *) NULL,
            (HRGN) NULL, (LPRECT) NULL, SW_INVALIDATE);
         si.cbSize = sizeof(si);
         si.fMask  = SIF_POS;
         si.nPos   = xPos;
         SetScrollInfo(hwnd, SB_HORZ, &si, TRUE);
         UpdateWindow (hwnd); //сгенерировать WM_PAINT
      }
       return 0;

   case WM_VSCROLL:  //обработка сообщения от вертикальной прокрутки
         switch(LOWORD (wParam)) {
           case SB_PAGEUP:
            yInc = min(-1, -yClient / yChar);
            break;

          case SB_PAGEDOWN:
            yInc = max(1, yClient / yChar);
            break;
```

```
        case SB_LINEUP:
            yInc = -1;
            break;

        case SB_LINEDOWN:
            yInc = 1;
            break;

       case SB_THUMBTRACK:
            yInc = HIWORD(wParam) - yPos;
            break;
        default:
            yInc = 0;
    }

//реализация новой позиции бегунка вертикальной прокрутки
    if (yInc = max(-yPos, min(yInc, yMax - yPos))) {
       yPos += yInc;
       ScrollWindow(hwnd, 0, -yChar * yInc,
          (CONST RECT *) NULL, (CONST RECT *) NULL,
          (HRGN) NULL, (LPRECT) NULL, SW_INVALIDATE);
       si.cbSize = sizeof(si);
       si.fMask  = SIF_POS;
       si.nPos   = YPos;

       SetScrollInfo(hwnd, SB_VERT, &si, TRUE);
       UpdateWindow (hwnd);
    }

    return 0;
```

3.17 Диалоговые панели

Часто в приложениях необходимо создавать вспомогательные окна для ввода выбора и указания некоторой информации. Для таких целей служат диалоговые панели. Для создания диалоговой панели:

1) создать описание диалога в ресурсном файле;
2) создать код, выводящий диалоговую панель на экран;
3) создать диалоговую процедуру, которая будет обрабатывать сообщения.

Диалоговые панели создаются с помощью функций DialogBox и CreateDialog.

Функция DialogBox создает диалоговую панель, работающую в модальном режиме, при котором приложение не будет активизировано, пока не будет закрыта диалоговая панель. Описание функции следующее:

```
int DialogBox(
   HINSTANCE hInstance,// дескриптор приложения
   LPCTSTR lpTemplate,//имя диалога в ресурсном файле
   HWND hWndParent,// дескриптор окна
   DLGPROC lpDialogFunc //адрес диалоговой процедуры
  );
```

Описание диалоговой процедуры должно иметь следующий вид:

```
BOOL APIENTRY DlgProc(hwndDlg, message, wParam, lParam)
HWND hwndDlg;
UINT message;
WPARAM wParam;
LPARAM lParam;
{
   switch (message) {

      // Обработка сообщений
case  WM_INITDIALOG: //
        return TRUE;
        case WM_COMMAND://
        break;
      default:
         return FALSE;
   }
}
```

Глава 4. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Введение

Практика программирования на процедурно-ориентированных языках, таких как Си или Паскаль показала, что хотя эти языки являются эффективным инструментом создания программ, однако существуют проблемы адаптации программного обеспечения. Часто незначительные изменения в базовых структурах или функциях приводят к ситуации переделки достаточно большой части программы, а то и всей программы. Ниже рассматриваются проблемы адаптации программ и основные предпосылки перехода на идеи объектно-ориентированного программирования.

Для освоения идей объектно-ориентированного программирования требуется знание процедурно-ориентированного программирования и языка программирования Си. Рассмотрим несколько примеров для организации программ реализации стека написанных на языке программирования Си. Для рассмотрения проблем и предпосылок взята хорошо известная в программирования структура — стек. Предлагается последовательно ряд примеров с описаниями проблем и недостатков, решение которых приводит к пониманию объектно-ориентированного программирования.

```
#include <iostream.h>
//#define VAR1 //Для прогона первого    варианта
//#define VAR2 //Для прогона второго    варианта
//#define VAR3 //Для прогона третьего   варианта
//#define VAR4 //Для прогона четвертого варианта
//#define VAR5 //Для прогона пятого     варианта
//#define VAR6 //Для прогона шестого    варианта
//#define VAR7 //Для прогона седьмого   варианта
//#define VAR8 //Для прогона восьмого   варианта
/*************************************************************************
            Вариант первый. Простейший
*************************************************************************/
#ifdef VAR1
#define SIZE 100  //размер стека
int Stack[SIZE];  //статический массив под стек
int Top;          //указатель стека
/*************************************************************************
/
void Init() { Top=0; } //установка указателя стека
/*************************************************************************
/
void Push(int Item)    //поместить значение в стек
{
 if(Top<SIZE) Stack[Top++]=Item;
}
```

```
/*************************************************************************
/
int Pull()            //удалить значение из стека
{
 if(Top>0) return Stack[--Top];
 return 0;
}
/*************************************************************************
/
int main()            //примеры использования стека
{
 int e;
 Init();
 Push(10); Push(20); Push(30);
 while(e=Pull()) cout<<e<<"\n";
}
/*************************************************************************
*
 Основные достоинства: простота программного кода и
 эффективное его выполнение.
 Недостатки: Стек один, память статическая, размер стека фиксирован заранее,
 все переменные стека разрознены и доступны для всех. Стек хранит сами значения,
 а не указатели на объекты. Все имена внешние и могут конфликтовать с именами
 других разработчиков.
*************************************************************************
*/
#endif


#ifdef VAR2
/*************************************************************************
/
/*Вариант второй. Объединение всех данных стека в единую структуру.        */
/*************************************************************************
/
#define SIZE 100
struct Stack  //объединение данных стека в структуру
{
 int Stack[SIZE]; //массив стека
 int Top;        //указатель стека
};
struct Stack Var2; //выделение статической памяти под стек
/*************************************************************************
/
void Init() { Var2.Top=0; } //инициализация указателя
/*************************************************************************
/
void Push(int Item)        //поместить значение в стек
{
 if(Var2.Top<SIZE) Var2.Stack[Var2.Top++]=Item;
```

```
}
/*************************************************************************
/
int Pull()                //удалить значение из стека
{
 if(Var2.Top>0) return Var2.Stack[--Var2.Top];
 return 0;
}
/*************************************************************************
/
int main()                //примеры использования
{
 int e;
 Init();
 Push(101); Push(202); Push(303);
 while(e=Pull()) cout<<e<<"\n";
}
/*************************************************************************
/
/*
Из примера видно, что основные недостатки остались, несколько усложнилось
описание функций, т.к. стали более длинные имена данных стека. Но в этом случае,
имена данных стека объеденены и спрятаны в структуре, хотя доступ к ним попрежне-
му
открыт для всех.
*/
/*************************************************************************
/
#endif


/*************************************************************************
/
/*        Вариант третий. Использование динамической памяти.          */
/*************************************************************************
/
#ifdef VAR3
#include <alloc.h>
typedef struct //задается новый тип

{
 int *Stack; //память под стек выделяется динамически, но при создании
 int Size;   //размер стека
 int Top;    //указатель стека
 } STACK;

STACK Var3; //выделяем память под структуру var3
/*************************************************************************
/
void Create(int Size) //Функция выделения памяти под массив stack.
```

```
{                  //Size - максимальное количество целых, которое может
                   //хранить стек.
 Var3.Stack=(int *)malloc(sizeof(int)*Size);
 if(Var3.Stack==NULL) Var3.Size=0;
 else Var3.Size=Size;
 Var3.Top=0;
}
/*************************************************************************
/
void Init() { Create(100); } //функция инициализации, если не надо явно
                        // указывать размер
/*************************************************************************
/
void Push(int Item)          //Поместить элемент в стек
{
 if(Var3.Top<Var3.Size) Var3.Stack[Var3.Top++]=Item;
}
/*************************************************************************
/
int Pull()                   //Вытащить элемент из стека
{
 if(Var3.Top>0) return Var3.Stack[--Var3.Top];
 return 0;
}
/*************************************************************************
/
void Destroy()               //Уничтожить стек
{
 if(Var3.Size>0) free(Var3.Stack);
}
/*************************************************************************
/
int main()                   //Примеры использования
{
 int e;

 Create(50);
 Push(111); Push(222); Push(333);
 while(e=Pull()) cout<<e<<"\n";
 Destroy();
}
#endif

/*************************************************************************
/
/*Вариант четвертый. Использование параметров для работы с разными стеками */
/*************************************************************************
/
#if defined(VAR4)||defined(VAR5)
#include <alloc.h>
```

```
typedef struct  //описание данных стека
{
 int *Stack; //память под стек выделяется динамически, но при создании
 int Size;   //размер стека
 int Top;    //указатель стека
 } STACK;
/*************************************************************************
/
void Create(STACK *st,int Size) //создать массив для параметра st
{
 st->Stack=(int *)malloc(sizeof(int)*Size);
 if(st->Stack==NULL) st->Size=0;
 else st->Size=Size;
 st->Top=0;
}
/*************************************************************************
/
void Init(STACK *st) { Create(st,100); } //инициализация по умолчанию для st
/*************************************************************************
/
void Push(STACK *st,int Item) //поместить элемент Item в стек st
{
 if(st->Top<st->Size) st->Stack[st->Top++]=Item;
}
/*************************************************************************
/
int Pull(STACK *st)          //удалить элемент из стека st
{
 if(st->Top>0) return st->Stack[--st->Top];
 return 0;
}
/*************************************************************************
/
void Destroy(STACK *st)      //освободить память выделенную под массив стека
{
 if(st->Size>0) free(st->Stack);
}
#ifndef VAR5
/*************************************************************************
/
int main()                   //пример использования
{
 STACK v4_1, v4_2;
 int e;
 Create(&v4_1,50);
 Init(&v4_2);
 Push(&v4_1,111); Push(&v4_1,222); Push(&v4_1,333);
 Push(&v4_2,Pull(&v4_1)); Push(&v4_2,Pull(&v4_1)); Push(&v4_2,Pull(&v4_1));
 while(e=Pull(&v4_2)) cout<<e<<"\n";
 Destroy(&v4_1);
```

```
 Destroy(&v4_2);
}
#endif
/*************************************************************************
/
/*
```

Внимательно присмотревшись к данному варианту реализации стека, можно увидеть, что структура STACK и функции Create, Init, Push, Pull, Destroy тесно связаны друг с другом. Эта связь проявляется в том, что первым параметром передается адрес объекта (в нашем примере это переменные v4_1 и v4_2). Кроме того функции Create, Init, Destroy выполняют особую роль. Они вызываются только один раз и Create, Init — сразу после создания объекта, а Destroy перед уничтожением объекта. Итак напрашиваются следующие предложения:

1) связать явно структуру STACK и соотвествующие функции, при этом имя объекта удалить из списка параметров и в текстах самих функций. При этом неявно предполагается что адрес объекта передается первым в списке параметров. Описание функций должно быть в структуре STACK;

2) функции Create, Init вызывать автоматически при создании объектов, при этом в описании явно указать что это функции создания объекта;

3) функцию Destroy вызывать автоматически в момент удаления объекта, при этом в описании явно указать что это функции удаления объекта.

Тогда наш пример будет выглядеть приблизительно таким образом:

```
class STACK
{
 int *Stack; //память под стек выделяется динамически, но при создании
 int Size;   //размер стека
 int Top;    //указатель стека
 void Push(int it);
 int Pull();
 constructior Create(int sz);
 destructor Destroy();
 };

 STACK v1(50), v2(100);
 v1.Push(111); v1.Push(222); v1.Push(333);
 for(int i=0; i<3; i++) v2.Push(v1.Pull());
 while(e=Pull(&v4_2)) cout<<e<<"\n";
*/
/*************************************************************************
/
#endif
```

```
/*************************************************************************
/
/* Предположим, что в структуру STACK необходимо внести дополнения, например
вести подсчет количества отрицательных и положителельных чисел хранящихся в
стеке. В этом случае приходится переписывать заново практически все, и структуру и
функции.
*/
#ifdef VAR5
/*************************************************************************
/
typedef struct {
 STACK stack; //струтура старого( базового) стека
 int positiv; //новые вводимые элементы
 int negativ;
} NSTACK;
/*************************************************************************
/
void NCreate(NSTACK *st,int Size) //создать массив для параметра st
{
  Create(&st->stack,Size);
  st->positiv=0;
  st->negativ=0;
}
/*************************************************************************
/
void NPush(NSTACK *st,int Item) //поместить элемент Item в стек st
{
 if(Item<0) st->negativ++;
  else
 if(Item>0)
  st->positiv++;
 Push(&st->stack,Item);
}
/*************************************************************************
/
int NPull(NSTACK *st)          //удалить элемент из стека st
{
 int Item=Pull(&st->stack);
 if(Item<0) st->negativ--;
  else
 if(Item>0)
  st->positiv--;
 return Item;
}
/*************************************************************************
/
void NDestroy(NSTACK *st)       //освободить память выделенную под массив стека

{
 Destroy(&st->stack);
```

```
}
/*************************************************************************
/
int GetPositive(NSTACK *st) { return st->positiv;}
/*************************************************************************
/
int GetNegative(NSTACK *st) { return st->negativ;}
/*************************************************************************
/
int main()                  //пример использования
{
 NSTACK v5_1, v5_2;
 int e;
 NCreate(&v5_1,50);
 NCreate(&v5_2,100);
 NPush(&v5_1,111); NPush(&v5_1,-222); NPush(&v5_1,333);
 NPush(&v5_2,NPull(&v5_1)); NPush(&v5_2,NPull(&v5_1));
 NPush(&v5_2,NPull(&v5_1));
 cout<<"Положительные "<<GetPositive(&v5_2)<<"\n";
 cout<<"Отрицательные "<<GetNegative(&v5_2)<<"\n";
 while(e=NPull(&v5_2)) cout<<e<<"\n";
 NDestroy(&v5_1);
 NDestroy(&v5_2);
}
/*************************************************************************
/
/*
 Возможность создания нового программного обеспечения путем изменения уже
 имеющегося получило название "наследования". Наследование может быть
 организовано:
 1) введением новых данных в структуру;
 2) изменением уже имеющихся функций
 3) добавление новых функций.
*/
#endif
/*************************************************************************
/
/*
 Рассмотрим пример еще одну практическую задачу, которая часто встречается при со-
провождении программного обеспечения. Предположим, что у нас имеется функция
 void AnimalSay(Animal *a);
*/
#ifdef VAR6
#include <string.h>
#include <alloc.h>
typedef struct {
 int x;
 char *say;
} ANIMAL;
```

```
/*************************************************************************
/

void Create(ANIMAL *a,char *s) //создание объекта
{
 a->say=(char *)malloc(strlen(s)+1);
 if(a->say!=NULL) strcpy(a->say,s);
 a->x=0;
}
/*************************************************************************
/
void Destroy(ANIMAL *a)      //удаление памяти
{
 if(a->say!=NULL) free(a->say);
}
/*************************************************************************
/
void Say(ANIMAL *a)
{
 cout<<a->say<<"\n";
 a->x=100/2;          //код, котрый требуется изменить
}
/*************************************************************************
/
void AnimalSay(ANIMAL *dog ) //Функция где используется Say
{
 //.... операторы
 Say(dog);
 //.... операторы
}
/*************************************************************************/
void main()
{
  ANIMAL dog;
  Create(&dog,"гав гав");
  AnimalSay(&dog);
  Destroy(&dog);
}
#endif
/*************************************************************************
/
/* Предположим, что необходимо изменить функцию Say, при этом нет исходного
текста AnimalSay. Ясно что при таком подходе изменить вызов функции в FnimalSay
Say нельзя. Однако выход есть, если в структуру ANIMAL ввести указатель на
функцию, а в функции AnimalSay
использовать вызов функции через этот указатель. Тогда
*/
/*************************************************************************
/
#ifdef VAR7
```

```
#include <string.h>
#include <alloc.h>

typedef struct Animal {
 int x;
 char *say;
 void (*Say)(Animal *a);   //указатель на функцию
} ANIMAL;
/*************************************************************************
/
void Create(ANIMAL *a,char *s) //создание объекта
{
 a->say=(char *)malloc(strlen(s)+1);
 if(a->say!=NULL) strcpy(a->say,s);
 a->x=0;
}
/*************************************************************************
/
void Say_Dog(ANIMAL *a)
{
 cout<<a->say<<"\n";
 a->x=100/2;          //код, который требуется изменить
}
/*************************************************************************
/
void Say_Cow(ANIMAL *a)
{
 cout<<a->say<<"\n";
 a->x=100*2;          //код, который требуется изменить
}
/*************************************************************************
/
void Create_Dog(ANIMAL *dog) //конструктор для Dog
{
  Create(dog,"Собака лает: Гав-Гав");
  dog->Say=Say_Dog;
}
/*************************************************************************
/
void Create_Cow(ANIMAL *cow) //конструктор для Cow
{
  Create(cow,"Корова мычит: My-My");
  cow->Say=Say_Cow;
}
/*************************************************************************
***/

void Destroy(ANIMAL *a)      //удаление памяти
{
 if(a->say!=NULL) free(a->say);
```

```
}
/*************************************************************************
/
void AnimalSay(ANIMAL *a ) //Функция где используется Say
{
 //.... операторы
 a->Say(a);
 //.... операторы
}
/*************************************************************************
/
void main()
{
  ANIMAL dog;          //создание объектов dog и cow
  ANIMAL cow;
  Create_Dog(&dog);
  Create_Cow(&cow);

  AnimalSay(&dog);    //использование созданных объектов
  AnimalSay(&cow);

  Destroy(&cow);      //удаление объектов dog и cow
  Destroy(&dog);
}
/*
```

В этом примере показан механизм, который обеспечивает изменение некоторой части программы, причем вызов функции Say через указатель может быть в любом месте функции AnimalSay, при этом саму функцию AnimalSay ни коим образом не изменяем. Предположим, что AnimalSay должна выдать «Кошка мяукает: Мяу-Мяу» Для этого создаем функцию Cat_Say(), конструктор Create_Cat() и деструктор Destroy_Cat(), если этот деструктор необходимо.

В этом примере показана идея организации полиморфного поведения. Что здесь важно отметить:

1. Описание AnimalSay не изменяется, на входе у нее параметр — указатель на структуру типа ANIMAL.

2. Структура ANIMAL содержит указатель на функцию.

3. Используя идеи наследования, мы в переопределяем функции Create(), Say() и Destroy()

4. Используя указатель типа Animal, который содержит адреса на разные объекты, мы получаем разный результат (полиморфное поведение).

Этот пример учебный, в реальной практике все может оказаться сложнее. Например, разработана большая бухгалтерская программа, в которой запрограммирована работа с десятками различных бланков, и вот формат одного из бланков меняется. Используя идеи ООП возможно:

1) используя свойство наследования определить структуру и функции обеспечивающие работу с новым форматом бланка;

2) используя свойство полиморфизма встроить новую кусок программы в уже имеющийся таким образом, что другая часть осталась без изменения.

```
*/
#endif
#ifdef VAR8
//Построение таблицы функций
enum {Func_X,Func_Y,Func_Z};
typedef struct A {
int x;
void *func[3];
} AF;
int X() { cout<<"Привет\n"; return 1;}
void Y(int i) { cout<<"Здорово "<<i<<"\n";}
void Z(int i,char *s ) { cout<<"Добрый день "<<i<<" "<<s<<"\n";}
/**************************************************************************
/
void main()
{
 AF v;
 //заполнение таблицы
 v.func[Func_X]=(void *)X;
 v.func[Func_Y]=(void *)Y;
 v.func[Func_Z]=(void *)Z;

 cout<<"**********************\n";
  //вызовы функций через соотвествующие элементы таблицы
 ((int (*)())(v.func[Func_X]))();
 ((void (*)(int))(v.func[Func_Y]))(100);
 ((void (*)(int,char *))(v.func[Func_Z]))(100,"Козлик");
}
#endif
```

4.1 Объектно-ориентированное программирование на C++

4.1.1 Простейший ввод/вывод

В C++ имеется достаточно простой с точки зрения пользователя метод ввода/вывода, основанный на потоках. Поток это удобный инструмент ввода/вывода информации. Поток организуются от источника информации (устройство ввода) в оперативную память, и из оперативной памяти в приемник информации. Хотя понятие источник и приемник может быть расширено (это может быть ОП, устройство ввода/ вывода, файл). В C++ имеется два стандартных потока:

1) cin — поток для ввода информации с клавиатуры;
2) cout — поток для вывода информации на терминал.

Как организовать ввод/вывод с помощью стандартных потоков:
1) подключить заголовочный файл iosteam.h;
2) записать поток для ввода, используя операцию «>>»;
3) записать поток для вывода информации, используя операцию «<<».

Например,

```
#include <iostream.h>
void main()
{
  int i, char s[80];
  cin>>i; //Ввод целого и присвоение в качестве значения для i
  cin>>s; //Ввод строки символов и присвоение переменной s
  //Можно записать вместе
  cin>>i,s;
  //Для вывода информации необходимо использовать cout
cout<<i; //Вывод целого значения, которое хранится в i
cout<<s; //Вывод строки символов
//можно вместе
cout<<i<<s;
//можно и так
cout<<"значение i ="<<i<<"  значение s ="<<s<<"\n";
}
```

Приведенные выше примеры вполне достаточны для освоения материала по организации ООП в С++. Более подробно о потоках будет сказано в разделе ...

4.1.2 Понятие ссылки

Как известно в Си параметры в функции передаются по значению. В С++ параметры можно передать как по значению, так и по ссылке. Ссылка скрытно связана с указателем. Если указатель это переменная, содержащая адрес объекта, то ссылка это просто сам адрес объекта. Иными словами, ссылка это еще одно имя уже имеющегося объекта. В некоторых случаях вместо передачи в качестве аргументов указателей, лучше воспользоваться ссылкой.

Простые ссылки

Ссылку можно объявить вне описания функции следующим образом:

```
int i; //создаем объект int с именем i
int &iref=i; //создаем ссылку на объект с именем i, это просто еще одно имя объекта
iref=3;  //это тоже что и i=3;
```

Очевидно, что &i и &iref имеют одно и тоже значение, поскольку это адрес одного и того же объекта.

Аргументы ссылки

Ссылки могут быть объявлены как параметры функции:

```
void Func1(int i)
void Func2(int& iref) // объявление параметра как ссылки
  ...
int sum=5;
Func1(sum); //передача параметра по значению
Func2(sum); // передача параметра по ссылке.
```

В первом случае значение переменной sum переписывается (копируется) в локальную память функции и все изменения, которые происходят в функции с параметром i, никак не влияют на переменную sum. Во втором случае, передается не значение sum, а его адрес поэтому все изменения, которые происходят с параметром i в функции непосредственно влияют на объект sum. Передача параметров через ссылки альтернативный метод передачи параметров через указатели. Рассмотрим пример

```
int Trip1(int n)  // передача и возврат значений
{
 return 3*n;
}
   ...
int x,i=5;
x=Trip1(i);     //  теперь x=15 i=5 без изменения
    ...
void Trip2(int *nptr) //передача адреса объекта через указатель
{
  *nptr=(*nptr)*3;
}
  ...
Trip2(&i); //передаем адрес объекта i, после выполнения i=15;

void Trip3(int &nref)//передача по ссылке
{
  nref=nref*3;
}
```

```
  ...
Trip3(i); //После выполнения i=45;
```

Передача параметров копированием является предпочтительным вариантом (см описание функции Trip1). Однако в случае достаточно больших по размеру объектов такой способ передачи становится неэффективным. Для этого использовали передачу адреса объекта через параметр типа указатель, однако, этот метод очень громоздкий с точки зрения написания и читабельности программы (см. описание функции Trip2). Передача параметров через ссылки для таких случаев наиболее проста и понятна.

Преобразование типа при создании ссылки

Если при объявлении ссылки инициализируется константа или объект с другим типом, то автоматически создается временный объект. Рассмотрим примеры

```
int& iref=5; //создается временный объект типа int, которому будет
присвоено значение 5

double x;
int& ix=x; //создается временный объект типа int, которому будет
присвоено значение 5

double x=10.5;
 int& ix=x; //создается новый объект типа int, значение x преобразуется в
целое (10)
                 //и присваивается этому временному объекту
 cout<<ix<<"\n";
 ix=5.1;      //происходит преобразование в целое и присвоение объекту
типа int, a
               //значение x не изменится
 cout<<ix<<" "<<x<<"\n";
```

Аналогично это касается и передачи параметров. Например,

```
void Doub(int &n) { n=n*2; }
...
float x=8;
Doub(x);
cout<<x;
```

В этом случае при передаче параметра будет создан временный объект типа int, ему будет присвоено значение 8, потом адрес его передан в функцию Doub и там удвоен, после выполнения функции значение x не изменится.

Ссылки на структуры

```
struct S { int i; char s[10]; };
void Show(S& z) { cout<<z.i<<" "<<z.s<<"\n";
...
S y={10, "десять" };
Show(y);
```

В этом случае внутри функции ссылка воспринимается как обычная структура.

Ссылки на указатели

Указатель можно рассматривать как объект, поскольку по него выделяется память, поэтому на сам указатель можно организовать ссылку. Например

```
int *p;
int* &pref=p; //организация ссылки на указатель
                //далее pref можно использовать как указатель
int j=9;
p=&j;
cout<<*pref<<"\n"; //будет выведено значение 9
```

Ссылки на указатели можно организовать и как параметры функций. Например,

```
struct Zs { int x; char *ptrs;};
void func(Zs* &pref) //параметр pref ссылка на указатель типа Zs
{
  cout<<pref->x<<" "<<pref->ptrs<<"\n";
}
...
Zs w, *pw;
... //присваиваются значения w и pw
func(&w);
func(pw);
```

Надо помнить следующие отличия.

В первом вызове при передаче параметра создается временный объект типа указателя на Zs и если происходит изменения в указателе в теле функции, то изменяется значение вновь созданного объекта. Во втором случае изменения будут содержаться в указателе pw.

Возврат ссылок из функций

Можно организовать возврат ссылок из функций, например

```
struct B { int x; double dx; };
B a; //создание статического объекта
B& func() { return a; } //функция возвращает ссылку на объект a
....
func().x=10;    //присвоение значения a.x
func().dx=0.55;//присвоение значения a.dx
....
cout<<func().x+2<<"  "<<2*func().dx<<"\n";
....
```

Результатом работы этого фрагмента программы будет:

12 1.10

Возврат ссылок можно организовать на объекты, которые сохраняются после завершения работы функции, а именно:

1) на статические объекты;
2) на динамические объекты;
3) на параметры, которые переданы в функцию как ссылки или указатели.

Рассмотрим возврат ссылок на динамические объекты (этот раздел требует знаний операторов new и delete).

```
struct Pair { int x; int y; };
 Pair& Get()
 {
   Pair *ptr;
   ptr=new Pair; //выделяем память под объект
   return *ptr;    //возвращаем ссылку на новый объект типа Pair
 }
....
Pair& pair=Get();
Pair* ptr=&Get();
Get().x=20; //так делать нельзя, теряется память выделенная под объект
```

```
Get().y=30; //здесь тоже
....
delete &pair;
delete ptr;
```

Описанная функция Get() создает новый объект при каждом обращении и возвращает его адрес как ссылку. Поэтому в вызывающей функции необходимо адрес нового объекта как ссылку или как указатель (см. первые две строчки примера) Следующие две строки будут выполнены, но адреса объектов будут потеряны, т.к. они нигде не запоминаются, таким образом, образуется мусор в памяти. Если эти действия многократны, то может произойти ситуация когда память не будет выделена. Последние две строчки предназначены для утилизации памяти под объекты созданные функцией Get().

В некоторых случаях в функции полезно принимать объект по ссылке и возвращать его по ссылке. Например,

```
stream& Show(stream& out, ...)
{
  ...
  return out;
}
```

Здесь stream некоторый тип, например описание класса предназначенного для вывода.

4.1.3 Операторы new и delete

Динамическое распределение памяти является эффективным инструментом программирования, позволяющее конструировать объекты во время выполнения программы. Причем время когда создавать и когда удалять эти объекты решает сам программист. В Си с этой задачей функции, описанные в заголовочном файле alloc.h, например, для выделения динамической памяти функция malloc(size_t n), для освобождения free(). В C++ для этих целей ввели специальные операторы:

1) new — выделяет память по объект;
2) delete — освобождает память занимаемую объектом.

Рассмотрим примеры:

```
//пример №1
int *ptri;
```

```
ptri=new int; //выделить память под целое и адрес блока памяти присвоить
указателю ptri;
.... //операторы, где используется указатель ptri
delete ptri; //освобождение памяти.
```

Если помять не может быть выделена, то оператор new возвращает NULL. Поэтому часто возникает необходимость проверки возвращаемого значения. Кроме того, существуют механизмы генерации исключительных ситуаций, о которых будет сказано позже.

```
//пример №2
struct  Book { double price; char *name; char *author;};

Book *best;
best=new Book;
...// Использование best
delete best;
```

Здесь показан пример распределения памяти под объект типа структура. Что нужно отметить, что оператор new автоматически определяет размер необходимой памяти под объект. В данном примере память будет выделена под переменную типа double и два указателя типа char.

Память динамически можно выделять и для ссылок. Например,

```
//Пример №3
long &lref=*new long;
lref=20L;
...//операторы где используется lref
cout<<lref<<"\n";
delete &lref; //удаление объекта
```

Можно также рассмотреть пример из предыдущего раздела (см. функцию Get())

Рассмотрим теперь выделение и освобождение памяти под массивы. Тут есть небольшие отличия. Необходимо явно записывать размер выделяемой памяти, кратный количеству элементов.

Например,

```
int ivec[]=new int[20]; //выделение памяти под массив из 20 целых чисел.
for(int i=0; i<20; i++) ivec[i]=i;
....
delete []ivec;
```

При удалении памяти, выделенной с явным указанием количества элементов в операторе new, в операторе delete необходимо записать квадратные скобки. См. оператор delete для вектора ivec. Все это и касается типа char. Например,

```
char *ptrs=new char[20];
strcpy(ptrs,"Привет козлик");
cout<<ptrs<<"\n";
...
delete []ptrs;
```

Типичная ошибка для начинающих, забывают поставить скобки [] в операторе delete для удаления массива.

В C++ имеется возможность переопределять операторы new и delete для отдельных классов, а также и для всех типов данных разрабатываемой программы.

С помощью оператора new можно распределять память по многомерные массивы

Например,

```
int n=10;
double (*matrix)[6]; //указатель на вектор размеров 6 элементов
matrix=new double[n][6]; //выделение памяти под матрицу
matrix[0][0]=1.; //операторы для работы с матрицей
delete []matrix; //удаление памяти под матрицей

int k=2;
double (*tensor)[5][10]; //указатель на матрицу размером [5][10] элементов
matrix=new double[k][5][10]; //выделение памяти под тензор
tensor[0][0][0]=1.; //операторы для работы с тензором
delete []tensor; //удаление памяти  занимаемый тензором
```

Необходимо заметить, что для использования оператора new для всего массива требуется указать заранее значения всех размерностей, кроме первой. В тех случаях, когда размерности неизвестны, то используют механизм распределения основанные на векторизации массива (см. раздел ...).

Новые возможности функций в С++

В Си++ функции делятся на простые функции и функции члены класса. Причем с одним и тем же именем может быть много функций. Например,

```
void Show(int i) { cout<<i; }
void Show(int i,char *s) { cout<<i<<s; }
void Show(double f) { cout<<f; }
void Show() { cout<<"Привет"; }
```

В некоторых источниках дается термин переопределения функций. На самом деле компилятор для каждой функции создает уникальное имя. Например,

```
Show(int i)   внутренне имя будет Show_int, для остальных соответственно
Show_int_char
Show_double
Show
```

Важно знать, чтобы у совпадающих функций по именам, типы параметров в списке параметров не совпадали.

Для функций-членов классов правила несколько сложнее и будут рассмотрены ниже

Значение параметров по умолчанию.

4.1.4 Операция разрешения видимости ::

В С++ имена функций членов классов могут совпадать поэтому для указания принадлежности функции можно явно указывать при ссылке на член класса имя класса.

Например,

```
void X::Show() { ... } //функция принадлежит классу X
void V::Show() { ... } //функция принадлежит классу V
```

В любом месте функций можно обращаться к членам класса с помощью имени класса разделенного двойным двоеточием и именем члена класса. Например,

```
j=X::k;   //это выражение должно в функции члене класса X или в
наследуемом классе
z=X::Show(); // аналогично
```

4.1.5 Понятие класса

Класс в языке программирования C++ задает описание нового типа. Можно также говорить о классе как об описании множества объектов, поведение которых одинаково или очень похоже. Что нужно чтобы задать класс, нужно сделать описания данных и соответствующих функций. Например, для примера из раздела .... о стеке:

```
struct Stack //заголовок класса
{
  //описание данных
  int stack[100];
  int top;
  //описание  функций
  Init() { top=0; } //инициализировать указатель стека
  void Push(int Item)  //вставить целое значение в стек
   { if(top<100) stack[top++]=Item; }
 int Pull() { if(top>0) return  stack[--top];  //удалить значение из стека
                  return 0;
              }
};
```

Внимательно приглядевшись к описанию структуры Stack, мы увидим, что в этой структуре есть как описание данных (top, stack) , так и описание функций. Причем в телах функций есть обращение к данным, записанным в структуре (top и stack). Запись функций стала значительно проще, т.к. функции непосредственно связаны с данными структуры.

Элементы класса stack, top, Init, Push, Pull называются членами класса. Функции, принадлежащие классу, в некоторых источниках называют методами класса.

Класс в C++ это описание типа и ничего больше, в процессе исполнения программы класс нигде не хранится, и даже нельзя узнать имя класса объекта (если не предусмотрен специальный механизм идентификации класса во время выполнения RTTI, который будет рассмотрен позже).

Объект в C++ это фрагмент оперативной памяти, имеющий имя и тип, в некоторых случаях объект может не иметь имени, но он имеем всегда адрес и тип. Свойства объектов в C++ перечислены в разделе. Здесь мы покажем несколько примеров объявлений.

```
Stack s1; //выделяется память под объект  s1, тип памяти(локальный или
статический)
            //зависит от места где объявлен объект
Stack *ptr; //объявить указатель типа Stack
```

```
Stack &ref=s1; //объявить ссылку на объект s1
Stack ms[10];   //объявить массив из 10 элементов типа Stack
```

Правила записи обращения к членам класса такие же как и для структур в С. Например, для объектов объявленных выше,

```
s1.top=5;        //к членам данным обращение записывается как для структур в С.
s1.Push(20);   //вызов Push метода класса Stack для объекта s1
ptr=&s1; //инициализация указателя
int e=ptr->Pull(); //вызов метода Pull для объекта s1, через указатель ptr.
ref..Push(10);      //вызов метода Push по ссылке
ms[5].Init();        //вызов метода Init через элемент массива
ms[5].Push(1000);
...
```

Рассмотрим теперь использование класса Stack.

```
void main()
{
 Stack MStack; //построение объекта типа Stack в локальной памяти по имени MStack
                  //соответственно этот объект имеет адрес.
 MStack.Push(10);   //положить в MStack 10
 MStack.Push(20);   //положить в MStack 20
 MStack Push(500); //положить в MStack 500

 int e;
 while(e=MStack.Pull()) cout<<e<<"\n"; //вытащить из стека и отпечатать
}
```

Классы в С++ могут быть организованы с помощью ключевых слов: class, struct, union.

Последние два ключевых слова взяты из С. Отличие struct и class заключается в различном толковании прав доступа по умолчанию (см описание прав доступа).

Общие правила записи класса следующие:

```
class X { ... };
struct Y { ... };
union Z { ... };
```

4.1.6 Указатель this

Каким образом функция — член класса знает, что она работает именно с этим объектом? На этот вопрос можно легко ответить, адрес объекта передается в функцию член класса неявно, первым в списке параметров. Запись MStack.Push(10); можно интерпретировать как Push(&MStack,10); (см. пример из ....). Этот неявный параметр имеет имя this и является ключевым словом C++. Таким образом, ключевое слово this является адресом данного объекта для нестатических членов функций любого класса (про статические члены будет сказано позже). Очевидно, что this имеет локальную природу, т.к. фактически это параметр функции. Просто компилятор прячет его передачу.

Однако не для всех членов функций класса определен этот указатель, более подробно будет осуждено ниже.

4.1.7 Объект

Объект — это блок оперативной памяти, принимающий множество различных значений.

Перечислим основные свойства и атрибуты:

1. Объект должен иметь тип, типы могут быть стандартные (int, char, float и т.д.) и типы определяемые классами.

2. Объект обязан иметь адрес, поскольку он находится в оперативной памяти.

3. Объект может иметь несколько имен, в некоторых случаях объект имени может не иметь, например, при динамическом распределении памяти. Мы можем знать только его адрес

4. Объект может быть локальным, динамическим, статическим. Локальный объект — это объект явно объявленный в блоке { }. Под динамический объект память выделяется с помощью оператора new. Под статический объект память выделяется в момент загрузки программу, перед вызовом функции main и удаляет объект поле завершения функции main.

5. Видимость объекта это свойство, при котором есть доступ к объекту по имени.

6. Область действия объекта свойство, которое характеризует протяженность фрагмента программы в котором существует объект.

Простые объекты

```
int x, char y, float f; //зарезервированные типы
```

Указатель как объект

Поскольку под указатель распределяется память, то указатель можно рассматривать как специальный объект, содержащий адрес другого объекта.

Объекты массивы

```
int x[20];
double z[100][100];
```

Сложные объекты

```
MStack  stack(20);
```

4.1.8 Конструктор

Конструктор класса специальная функция-член класса, которая вызывается при создании объекта, тип которого является имя данного класса. Конструктор для статического объекта вызывается в момент загрузки программы в оперативную память. Для локального объекта в момент входа в соответствующий блок. Для динамического блока во время выполнения оператора new.

Конструктор для объекта вызывается только один раз, в момент создания объекта, поэтому конструктор вызывается автоматически. По этой же причине конструктор не возвращает никакого значения. Для объявления конструктора в классе необходимо записать функцию-член класса, у которой нет описания возвращаемого значения и имя конструктора совпадает с именем класса. Правила записи конструктора следующие:

```
class X
{
  ...
X() {...} //конструктор класса
...
}
```

Конструкторов может быть несколько (см. особенности записи функций) Например,

```
class Hello
{
  int n;
  double x;
  char str[80];
 public:
   Hello() { n=0; x=0; strcpy(str,"здрасте"); }
   Hello(int i) { n=i; x=0; strcpy(str,"Привет"); }
   Hello(double z) { n=0; x=z; strcpy(str,"Доброк утро"); }
```

```
  Hello(int i,double x) { n=0; this->x=x; strcpy(str,"Добрый день"); }
  Hello(int n,double t, char *s) { Hello::n=n; x=t; strcpy(str,s); }
};
```

Рассмотрим примеры объявления объектов, и вызова соответствующих конструкторов

```
Hello x; //вызывается конструктор  Hello()
Hello y(10) //вызывается конструктор Hello(int)
Hello z(10.5) // ... Hello(double)
Hello *ptr; //никакой конструктор не вызывается т.к объект не создается
ptr=new Hello(5,4.19); //создается динамический объект, вызывается
конструктор
Hello(int,double), адрес объекта присваивается указателю ptr;
Hello &t=x; //Конструктор не вызывается т.к. создается ссылка на
существующий объект
Hello Hi(10,5.0,"Hi"); //вызывается конструктор Hello(int,double,char*)
```

Конструктор копирования

Более сложные механизмы конструкторов будут описаны ниже.

4.1.9 Деструктор

Деструктор класса специальная функция, которая вызывается при удалении объекта. Он вызывается автоматически и не должен иметь параметров. Поэтому в классе может быть только один деструктор, который для объекта вызывается только один раз непосредственно при удалении объекта. Деструктор для статических объектов вызывается после завершения выполнения функции main. Деструктор для локальных объектов вызывается при выходе из соответствующего блока. Деструктор для динамических объектов вызывается во время выполнения оператора delete.

Правила записи деструктора следующие

```
class Example
{
  ....
  Example() {...}  //конструктор
  ~Example() { ...} //деструктор
 ....
};
```

Рассмотри примеры записи классов с конструкторами и деструкторами и их использование.

Пример 1

```
#include <iostream.h>
class Exp1
{
  int i;
 public: //это ключевое слово будет рассмотрено в разделе ...
 Exp1() { i=0; cout<<”Конструктор Exp1()\n”; }
 Exp1(int k) { i=k; cout<<”Конструктор Exp1(int)\n”; }
~Exp1() { cout<<”-Деструктор ”<<i<<”\n”; }
void Show() { cout<<i<<”\n”;
};

Exp1 x_static(55); //объявление статического объекта
void main()
{
  cout<<”Начало\n”;
  Exp1 y_local; //локальный объект с конструктором  Exp1()
  Exp1 z_local(777); //локальный объект с конструктором  Exp1()
  Exp1 *ptr =new Exp1; //динамический объект с конструктором Exp1()
  x_static.Show();
  y_local.Show();
  z_local.Show();
  ptr->Show();
  delete ptr;      //удаление динамического объекта
} //удаление локальных объектов
//удаление статических объектов
```

Листинг

```
Конструктор Exp1(int)
Конструктор Exp1()
Конструктор Exp1(int)
Конструктор Exp1()
55
0
777
0
Деструктор 0
Деструктор 777
Деструктор 0
Деструктор 55
```

Пример №2

```
class Vector
```

```
{
  int *vec; //указатель
  int size;  //размер вектора
  Vector() { vec=NULL; size=0; } //конструктор для пустого вектора
  Vector(int n) //конструктор для заданного размера
    {
      vec=new int[n]; //распределение памяти под вектор
       size=n;
     }
~Vector() {  if(size) delete []vec; } //деструктор
...// Другие члены
};
```

В этом примере показан механизм создания объекта, когда размер объекта заранее не определен. Если размеры некоторых элементов объекта заранее не определены, то поступают следующим образом: в описании класса элементы объектов с заранее неизвестными размерами заменяют соответствующими указателями, в конструкторах класса записывают параметры, через которые будут передаваться размеры элементов, и в теле конструкторов используется оператор new, соответственно, деструктор класса должен содержать операторы delete для всех динамически созданных элементов класса.

Очевидно, что объект Vector может быть статическим, локальным, динамическим, но память под элементы вектора (указатель vec) выделяется динамически.

Сложность конструктора зависит от сложности объектов класса. Объекты класса могут содержать отдельные элементы операционной системы, такие как дескрипторы окон, процессов и т.д.

4.1.10 Конструктор копирования

Во многих случаях происходит неявное копирование объектов. Например, при передаче параметров, или при возвращении значений из функций. Стандартное копирование объектов производится побитным копированием из блока памяти выделенное под источник в блок памяти, выделенный под приемник, можно это пояснить на примере

```
Type Obj1, Obj2;
Obj1=Obj2; //эта запись аналогична следующему
memcpy(&Obj2,&Obj1,sizeof(Type));
```

Однако, мы знаем что сам объект может состоять не только из сплошной памяти, но из множества различных блоков динамической памя-

ти (см Пример №2). В таких случаях копирование объекта будет некорректно, т.к. при копировании не будет учтен динамический характер элементов объекта. Например,

```
Vector v1(20),v2;
v1=v2;
```

В этом случае указатели v2.vec и v1.vec будут ссылаться на один и тоже блок динамической памяти. Что, очевидно, может привести к ошибке.

Другой пример

```
Vector x(20);
Vector FindVector(); //функция которая возвращает значение типа Vector
```

Тогда

```
x=FindVector(); //здесь скрыта ошибка
```

Здесь ошибка заключается в том, что объект x уже имеет динамическую память (указатель vec уже ссылается на динамическую память размера) и при стандартном копировании указатель x.vec теряет свое значение. Отсюда происходи утечка памяти.

Для избежания подобных ошибочных ситуаций необходимо предусматривать конструктор копирования. Правила записи конструктора копирования:

```
class X
{
  ....
 X(X &x) { ... } //конструктор копирования
 ....
};
```

Запишем конструктор копирования для класса Vector (См пример №2).

```
Vector::Vector(Vector &x)
{
 if(size) delete []vec;  //удаление динамической памяти у объекта приемника
vec=new int[x.size]; //выделение памяти под vec
size=x.size;
for(int i=0; i<size; i++) vec[i]=x.vec[i]; //копирование динамической памяти
}
```

Имея такой конструктор можно корректно производить следующие действия:

```
X x;
X y,z(x); //инициализация
x=y;       //присваивание
...
X func(X a, X b) { .... } //описание функции
...
z=func(x,y); //передачу и возврат значений.
```

4.1.11 Определение прав доступа к членам объектов класса

Определение прав доступа к членам объектов класса в C++ производится с помощью ключевых слов public, private, protected. Ключевое слово public обеспечивает открытость данного члена для использования в любом месте программы. Ключевое слово private означает, что к данному члену можно обратиться только из функций-членов данного класса. Ключевое слово protected означает, что к данному члену может иметь доступ все члены наследуемых классов (наследование см. ниже) По умолчанию в классе объявленном как class все члены имеют атрибут private. В классе struct по умолчанию все члены public. В классе union все члены по определению public.

```
class X {
  int i;   // X::i  является закрытым членом по умолчанию
  char ch;  // также и X::ch
public:
  int j;   // j и к открыты
  int k;
protected:
  int l;   // X::l  защищен
};
struct Y {
  int i;   // Y::i открытый по определению
private:
  int j;   // Y::j  защищенный
public:
  int k;    // Y::k открытый
};
union Z {
  int i;   // все члены открыты без вариантов
  double d;
};
```

Все открытые члены-функции класса задают, так называемый, интерфейс класса.

4.1.12 Статические члены класса

Статическими членами класса могут быть как данные, так и члены функции.

Статические члены класса объявляются с помощью ключевого слова static.

Статические члены-данные

Особенностью статических членов данных является тот факт, что память под них выделяется статически, один раз и таким образом для всех объектов этого класса объявленных в программе, независимо от их класса памяти, имеется всего один блок статической памяти выделенный под данный член класса. Очевидно, что инициализация таких членов не может быть осуществлена в конструкторах, то такие члены должны инициализироваться в вне описания класса. В С++. И доступ к таким переменным может быть осуществлен по имени класса. Рассмотрим пример,

```
class Point
{
 public:
 static int NumberPointObj;  //объявление статического члена
private:
 int x,y;
public:
  Point() { x=y=0; NumberPointObj++; }
  Point(int v, int t) { x=v; y=t; NumberPointObj++;}
  void Show() { cout<<x<"-"<<y<<"\n";
~Point() { NumberPointObj--};
//функция член-класса для вывода количества объектов
void ShowNumberObj() { cout<<"num="<< NumberPointObj<<"\n"; }
};
//обычная функция для вывода количества объектов Point
void ShowNumberObj() { cout<<"num="<< Point::NumberPointObj<<"\n"; }

int Point:: NumberPointObj=0; //инициализация статического члена класса Point

//Использование класса
//инициализация статических переменных класса  NumberPointObj равно 0
```

```
...
Point x; //создан один объект до входа в основную программу
            //значение NumberPointObj равно 1
void func()
{
  Point a(2,2),b(3,3),c(4,4), *ptr; //создано еще три объекта
  ShowNumberObj() ; //будет выведено значение 5
  a.ShowNumberObj() ; // здесь тоже будет выведено значение 5
  x.ShowNumberObj() ; // здесь тоже будет выведено значение 5
  ptr=new Point(10,20);
  ShowNumberObj() ; //будет выведено значение 6
  c.ShowNumberObj() ; // здесь тоже будет выведено значение 6
  ptr->ShowNumberObj() ; // здесь тоже будет выведено значение 6
  delete ptr; //удалить динамический объект
  ShowNumberObj() ; //будет выведено значение 5
  b.ShowNumberObj() ; // здесь тоже будет выведено значение 5
  //при выходе из функции все локальные объекты уничтожаются
   //значение NumberPointObj станет равно 2
}
////
void main()
{
  Point y(1,1); //создан еще один объект
            //значение NumberPointObj равно 2
  func();
  ShowNumberObj() ; //будет выведено значение 2
  y.ShowNumberObj() ; // здесь тоже будет выведено значение 2
  //удаляется локальный объект y
}
//удаляется статический объект x
```

Статические члены-функции

Статические члены-функции объявляются с помощью ключевого слова static.

Особенностью этих функций является, то обстоятельство, что они не имеют указателя this, они не имеют явного доступа членам класса. Например,

```
class Obj
{
 int type;
public:
```

```
 Obj(int t) { type=t; }
void Show() { cout<<"type "<<type<<"\n";
static void Print(Obj o)
{
  cout<<type; //здесь будет ошибка нет доступа к членам объекта
  Show();//здесь тоже будет ошибка нет доступа к членам объекта
  cout<<o.type; //здесь все будет нормально
 }
};

//использование статических функций
void main()
{
 Obj x(10);
 x.Show();
 x.Print(x);  //вызов статической функции для объекта x
 Obj::Print(x); //вызов статической функции через имя класса для объекта x
}
```

Инициализация статических членов для вложенных классов

```
class Example {
   static int x;
   static const int size =  5;
   class  Elem{  //объявление вложенных классов
      static float f;  //статический член вложенного класса
      void func(void);
      };
public :
   char array[size];
};
int Example::x = 1;
float Example::Elem::f = 3.14;  //инициализация статического вложенного
члена
void Example::Elem::func(void) {     /* определение функции-члена
вложенного класса */  }
```

4.1.13 Наследование

Наследование можно определить как формальный механизм расширения класса путем включения в него классов разработанных ранее. Предположим, что у нас имеется класс B

```
class B { ....};
```

и мы хотим получить другой класс D добавив туда несколько новых членов. Это расширение можно записать так:

```
class D: public B { ... };
```

Эта запись означает, что класс D включает в себя все члены класса B и новые члены, записанные в фигурных скобках. В таком случае говорят о том, что класс D наследует свойства класса B. Класс D может наследовать несколько классов, например,

```
class D: public B1, private B2 { ... };
```

Ключевые слова public и private определяет права доступа к членам класса B1 и B2 в классе D.

Рассмотрим примеры наследования:

```
class Point  //базовый класс
{
  protected:  //эти члены будут доступны для функций из наследуемых классов
    int x, int y;
  public: //интерфейс класса
 int& X() { return x;}
 int& Y() { return y; }
 void Show() { cout<<x<<" "<<y<<"\n";
};

class Pixel: public Point  //класс Pixel наследует класс Point
{
 int color;   //по умолчанию private
 public:     //интерфейс класса
  void Set(Point z, int col) { x=z.x; y=z.y; color=col; }
  void Show() { putpixel(x,y,color); }
};

void main()
{
  Pixel t;
  Point v;
   t.x=100; //здесь будет сгенерирована ошибка , т.к. x защищенный член.
   t.X()=100; //присвоить значение 100 t.x
```

```
  t.Y()=200; //присвоить значение 200 t.y
  v.Set(t,255); //установить значения для объекта v
  cout<<v.X()<<"  "<<v.Y()<<"\n"; //вывод координат  x и y объекта v.
  v.Show(); //вывести точку
  v.Point::Show() //вывести значения координат точки
}
```

Как видим из примера, что члены класса Point становятся членами класса Pixel.

Однако видимость некоторых членов класса Point исчезает, например функция Show() класса Point перекрывается функцией Show() класса Pixel. Используя имя класса и операцию разрешения видимости можно организовать доступ к функции Point::Show()

В тех случаях, когда имена членов данных в классах совпадают, то поступают аналогичным образом.

Правила определения прав доступа при наследовании

При записи класса, который наследует свойства другого класса, явно указывается ключевое слово, обозначающее права доступа. Ключевое слово может иметь значение:

public, protected и private.

1. Если при наследовании класс объявлен как public, то:

1) все public-члены базового класса останутся public в наследующем классе;

2) все protected-члены базового класса останутся protected в наследующем классе;

3) все private-члены базового класса останутся private в наследующем классе.

2. Если при наследовании класс объявлен как protected, то:

1) все public-члены базового класса останутся protected в наследующем классе;

2) все protected-члены базового класса останутся protected в наследующем классе;

3) все private-члены базового класса останутся private в наследующем классе.

3. Если при наследовании класс объявлен как private, то все public и protected члены базового класса в наследующем классе будут private.

Иерархия классов

Иерархия классов это совокупность классов порожденных от одного. Например

```
class Base { ... };
//первый уровень наследования
class D1: public Base { ... };
class D2: protected Base { ... };

//второй уровень наследования
class D11: public D1 { ...};
class D12: public D1 { ... };
class D21: public D2 { ...};
class D22: public D2 { ... };
```

//Третий уровень наследования это классы, порожденные от классов второго уровня.

Количество уровней неограниченно

Виртуальный класс

Выше дан пример простейшей иерархии классов. В реальной ситуации наследование может быть сложнее. Например

```
class Base { ... };
class D: public Base { ... };
class C: public Base { ... };
class Next: public D, public C { ... };
```

В этом случае класс Base входит и в D и в C. Получается две копии класса B. Для того чтобы избежать такой ситуации класс, который в механизме наследования может входить не один раз можно объявить виртуальным (ключевое слово virtual) тогда в производных классах будет храниться одна копия. Тогда можно записать:

```
class Base { ... };
class D: virtual public Base { ... };
class C: virtual public Base { ... };
```

Преобразования типов объектов для иерархии классов

Для иерархии классов возможно преобразование типов. Так объект объявленный для любого класса иерархии может быть преобразован к базовому. Например,

для описанной выше иерархии можно записать:

```
Base *pb; D1 *pd;
D1 o1; D2 o2; D11 o11; D22 o22;
pb=&o1; //через указатель pb  типа Base имеем доступ к членам объекта o1,
которые принадлежат классу Base (механизм виртуальных функций будет
описан ниже)
pb=&o11; // через указатель pb имеем доступ к членам объекта o11,
которые принадлежат классу Base.
pd=&o11; // через указатель pd типа D1 имеем доступ к членам объекта
o11, которые принадлежат классу D1
```

Возможно и обратное преобразование. Например,

```
D11 *pd11=(*D11) pd;
```

Однако узнать точно, что указатель pd ссылается объект данного производного класса в общем случае нельзя. Поэтому программист должен железно знать, что указатель ссылается на объект производного класса или использовать механизмы проверки типов во время выполнения программы (Например, RTTI — описание этого механизма смотри ниже.)

Все вышесказанное касается и ссылок.

Механизм доступа к функциям объекта производного класса через указатель на базовый класс иерархии описан в разделе Полиморфизм.

Конструкторы при наследовании

Конструкторы при наследовании записываются следующим образом:

```
class B {
 ...
B(..) { ...} //конструктор B
 ....
};
class D: public B {
...
D(..) :B(..) { .... }  //конструктор класса D
...
};
```

Перед описанием тела конструктора наследуемого класса, записывается список вызовов конструкторов базовых классов. Такая запись предполагает, что первыми будут выполнены конструкторы классов, которые лежат в основании иерархии.

Запишем пример

```
class OneD //базовый класс
{
  double x;
public:
  OneD() { x=0; cout<<"OneD\n"; }
  OneD(double z) { x=z; cout<<"OneD\n"; }
  Show() { cout<<"OneD::Show()\n"; }
};
class TwoD: public OneD //наследуем базовый
{
 double y;
 public:
  TwoD(): OneD() { y=0; cout<<"TwoD\n";} //вызов конструктра базового класса
  TwoD(double t, double v) :OneD(t) { y=v; cout<<"TwoD\n"; }
  Show() { cout<<"TwoD::Show()\n");
};
class ThreeD: public TwoD
{
 double z;
 public:
  ThreeD(): TwoD() { z=0; cout<<"ThreeD\n"}
  ThreeD(double t, double v, double w) :ThreeD(t) { z=w;  cout<<"ThreeD\n";}
  Show() { cout<<"ThreeD::Show()\n");
};
void main()
{
  TreeD xyz(1,2,3);
  xyz.Show();
}
```

Листинг

```
OneD
TwoD
ThreeD
ThreeD::Show()
```

Конструкторы для виртуальных классов

Последовательность вызова конструкторов для иерархии с виртуальными классами будет следующая:

1) первыми всегда вызываются конструкторы для виртуальных классов;
2) затем вызываются конструкторы для базовых классов;
3) последними вызываются конструкторы производных классов.

Например,

```
class  Alpha: public Beta, virtual public Gamma { ... };
Alpha x;
```

последовательность вызова конструкторов следующая:

```
Gamma()  //всегда первым
Beta()
Alpha()
```

Деструкторы при наследовании

Деструкторы при наследовании вызываются автоматически строго в обратном порядке вызовов конструкторов (см. Порядок вызова конструкторов).

Вызов конструкторов для объектов членов класса:

Например

```
class Point  //базовый класс
{
  protected:  //эти члены будут доступны для функций из наследуемых классов
    int x, int y;
  public: //интерфейс класса
Point() { x=0; y=0;}
Point(int x, int y) {  Point::x=x; Point::y=y; }
Point(Point& t) { x=t.x; y=t.y; }
};

class Line
{
 Point t1, t2;
```

```
public:
 Line(int x1, int y1, int x2, int y2):t1(x1,y1),t2(x2,y2) { cout<<"Line::Line(1)\n"; }
 Line(Point v,Point z):t1(v),t2(z) { cout<<"Line::Line(2)\n"; }
 Line(Line& a):t1(a.t1),t2(a.t2) { cout<<"Line::Line(3)\n"; }
...
};
```

4.1.14 Дружественные функции и классы

В некоторых случаях становится необходимым разрешить доступ функциям или другим классам к закрытым и защищенным членам данного класса. Это делается следующем образом: в классе, к членам которого требуется обеспечить доступ, записывается имя внешней функции или класса с ключевым словом friend.

4.1.15 Полиморфизм

Начнем рассмотрение понятия полиморфизма с примера. Предположим, у нас имеется некоторый класс X и некоторая функция Func

```
class X
{
   int x;
public:
 X() { x=0;}
 X(int i) {x=i;}
 void Show() { cout<<"X::Show()\n"; }
};
void Func(X *x) { ... x->Show(); ...}

void main()
{
 X x1;
Func(&x1); //здесь будет выведено X::Show()
}
Теперь используя свойство наследования построим класс Y
class Y: public X
{
   public:
  Y():X() {}
  Y(int c):X(c){}
  void Show() { cout<<Y::Show()\n"; }
};
void main()
{
 Y y1;
```

```
Func(&y1); //здесь будет выведено X::Show()
}
```

Нам бы хотелось чтобы в функции Func была вызвана член класса Y::Show(), не делая никаких изменений ни в классе X, ни в функции Func. В языке имеется специальный механизм виртуальных функций. Этот механизм позволяется через указатель или ссылку на объект базового класса подставлять объект наследного класса и при этом вызывить функции члены наследного класса. Для этого необходимо, что бы функции базового класса были объявлены виртуальными (ключевое слово virtual), а прототипы соответствующих членов наследного класса полностью совпадали. Например,

В базовом классе X необходимо функцию Show объявить следующим образом:

```
virtual void Show() { ... }
```

Рассмотрим еще пример

```
#include <iostream.h>
class Enimal
{
  public:
    virtual void Name() { cout<<"животное\n"; }
    virtual void Speak() { cout<<"неизвестно\n"; }
};
class Cat: public Enimal
{
   //...//члены данные
   public:
    virtual void Name() { cout<<"Кошка\n"; }
    virtual void Speak() { cout<<"Мяу-мяу\n"; }
   //...//другие члены функции
};
class Dog: public Enimal
{
   //...//члены данные
    int paw;
   public:
    virtual void Name() { cout<<"Собака\n"; }
    virtual void Speak() { cout<<"Гав-Гав\n"; }
    virtual int Paw() { cout<<"Лапы\n"; } //новая виртуальная функция
 //...//другие члены функции
};
class Bobik: public Dog
{
 //...//члены данные
   public:
```

```
    virtual void Name() { cout<<"Бобик\n"; }
    virtual void Speak() { cout<<"Гав-Гав\n"; }
//...//другие члены функции
};
class Barbos: public Dog
{
  //...//члены данные
   public:
    virtual void Name() { cout<<"Барбос\n"; }
    virtual void Speak() { cout<<"Тяф-Тяф\n"; }
    virtual int Paw() { cout<<"Лапы грязные\n"; }
//...//другие члены функции
};

void main()
{
 Enimal Unknown;
 Cat Little;
 Dog Mad;
 Bobik Bob;
 Barbos Derty;
 Enimal *Enimals[5]= { &Unknown, &Little, &Mad, &Bob, &Derty };
 for(int i=0; i<5; i++)
  {
   Enimals[i]->Name();
   Enimals[i]->Speak();
  }
  //массив ссылок не разрешен
 Dog *Dogs[3]={ &Mad, &Bob, &Derty };
 for(int i=0; i<3; i++)
  {
   Dogs[i]->Name();
  // Dogs[i]->Speak();
   Dogs[i]->Paw();
  }
}
```

Листинг
животное
неизвестно
Кошка
Мяу-мяу
Собака
Гав-Гав
Бобик
Гав-Гав
Барбос
Тяф-Тяф

Собака
Лапы
Бобик
Лапы
Барбос
Лапы грязные

Описанный выше пример показывает иерархию классов. Базовый класс иерархии является Enimal. В нем описаны две виртуальные функции Name и Speak. Производные классы Cat и Dog наследуют класс Enimal. В классе Dog определяется еще одна виртуальная функция Paw. Класс Dog является базовым еще для двух классов Bobik и Barbos. В классе Barbos переопределяется функция Paw, а в классе Bobik она отсутствует.

В функции main показаны примеры использования этой иерархии. Вначале программы объявлено пять объектов разных типов (принадлежащие различным классам) Далее объявлен массив указателей Enimals типа Enimal (базового класса всей иерархии) и элементам этого массива присвоены адреса объектов. Далее в цикле для каждого элемента массива вызывается функции Name и Speak. Эффект этого вызова показан в листинге. Если бы попытались в этом цикле вызвать функцию Paw, то компилятор выдал бы сообщение об ошибке, т.к. должны использоваться функции, описанные в базовом классе.

Во втором цикле уже используется массив указателей типа Dog и его элементам присвоены адреса объектов типа Dog, Bobik и Barbos. Здесь уже можно вызывать функцию Paw. Как видно функция Paw в классе Bobik не определена, то для объекта данного класса будет вызвана функция Paw базового класса (см. листинг)

Абстрактные классы

Описанный выше пример показывает, что функции в базовом классе Enimal не имеют смысла (поскольку имя у животного конкретное и выражается оно конкретно!) Поэтому в C++ предусмотрена возможность объявлять чистые виртуальные функции. В нашем случае:

```
virtual void Name()=0;
virtual void Speak()=0;
```

Присвоение виртуальной функции значения 0, делает эту функцию чисто виртуальной, а соответствующий класс становится абстрактным.

Для абстрактного класса характерно:

1) он является базовым для некоторой иерархии классов;
2) он не может описывать объекты;

3) он не может иметь конструкторов.

С помощью абстрактного класса могут быть объявлены указатели, ссылки и параметры функций.

Виртуальные деструкторы

В отличие от конструкторов, которые вызываются при создании объектов, деструкторы могут быть виртуальными. В иерархии классов предпочтительно деструкторы делать виртуальными. Виртуальные деструкторы записываются с помощью ключевого слова virtual Рассмотрим пример

```
#include <iostream>
class color {
public:
  virtual ~color() {  // это виртуальный деструктор
     cout << "color dtor\n";
     }
};
class red : public color {
public:
  ~red() {  // этот деструктор тоже виртуальный
     cout << "red dtor\n";
     }
};
class brightred : public red {
public:
  ~brightred() {  // этот деструктор также виртуальный
      cout << "brightred dtor\n";
      }
};
int main() {
  color *palette[3]; //массив указателей
  palette[0] = new red;
  palette[1] = new brightred;
  palette[2] = new color;

    delete palette[0]; //вызывается деструктор для класса color
    delete palette[1]; //вызывается деструктор для класса red
    delete palette[2]; //вызывается деструктор для класса brightred
  return 0;
}
```

Листинг программы

```
red dtor
color dtor
brightred dtor
red dtor
color dtor
color dtor
```

В том случае если бы мы не объявили деструкторы виртуальными, то вызвались бы деструкторы только для класса color и листинг бы содержал три строчки «color dtor».

4.1.16 Перегрузка операций

В C++ реализована перегрузка операций. Перегрузка операций возможна только для классов. Перегрузка операций естественный путь создания удобного интерфейса для класса. Например, если определяется класс для представления матриц, то естественно ввести операции сложения и умножения матриц.

Например,

```
class Matrix { ... };
Matrix x,y,z;
z=x+y; //сложение матриц удобнее представлять так
z=Add(x,y); //чем такое функциональное представление.
```

Предположим, имеется некоторая операция # тогда можно записать

1) x#y

2) x.operator#(y)

3) operator#(x,y)

Представление (2) и (3) есть две функциональных альтернативы перегрузки операции.

Компилятор как встречает выражение, включающее операцию # (например, x#y), то подставляет соответствующие функции x.operator#(y) или operator#(x,y). В первом случае функция operator#(C y) является членом класса C, во втором случае — просто функция.

Перегружать можно бинарные и унарные операции. Перегружать можно практически все операции, кроме следующих операций

. .* :: ?:

Операции =, [ ], ( ), и -> можно перегружать как нестатические члены классов.

Перегрузка бинарных операций

Рассмотрим переопределение операций + и – у на примере класса квадратных матриц размером 2.

```
class Matrix2
{
 double x[2][2];
 public:
  Matrix2() { x[0][0]=x[0][1]=x[1][0]=x[1][1]=0; }
  Matrix2(int x1,int x2,int x3,int x4) { x[0][0]=x1; x[0][1]=x2; x[1][0]=x3; x[1][1]=x4; }
  void operator+(Matrix2 y)  //переопределение операции сложения
   {
     for(int i=0; i<2; i++)
     for(int j=0; j<2; j++)
       x[i][j]+=y.x[i][j];
   }
  void operator-(Matrix2 y)  //переопределение операции вычитания
   {
     for(int i=0; i<2; i++)
     for(int j=0; j<2; j++)
       x[i][j]-=y.x[i][j];
   }
 void Show() //вывод элементов матрицы в одну строку
   {
     for(int i=0; i<2; i++)
     for(int j=0; j<2; j++)
       cout<<x[i][j]<<" ";
     cout<<"\n";
   }
};
void main()
{
 Matrix2 x; // нулевая матрица
 Matrix2 y(1,2,3,4); //присвоение значений (1,2,3,4) элементам матрицы
 Matrix2 e(1,1,1,1); //единичная матрица
 x.Show();  //показать значения   матрицы x
 x+y;         //вызвать член функцию x.operator+(y)
 x.Show();  //показать
 x-e;         //вызвать член функцию x.operator-(e)
 x.Show(); //показать
 //x+y-e;    //такая запись будет ошибочна
}
```

Листинг программы

```
0 0 0 0
1 2 3 4
0 1 2 3
```

Как видно из примера для унарных операций перегрузка операции осуществляется с помощью ключевого слова operator, затем идет символ операции, а затем в круглых скобках один параметр.

Как видим из приведенного выше описания класса члены-функции operator+() и operator-() не возвращают ничего, поэтому запись выражений типа x+y-e приводит к ошибке, поскольку результатом выражения x+y есть void. Для того чтобы это выражение имело смысл необходимо, чтобы члены функции operator+() и operator-() возвращали значения или ссылки типа Matrix2. Перепишем заново эти функции для класса Matrix2

```
Matrix2& operator+(Matrix2 y)  //переопределение операции сложения
    {
      for(int i=0; i<2; i++)
      for(int j=0; j<2; j++)
        x[i][j]+=y.x[i][j];
      return *this
     }
Matrix2& operator-(Matrix2 y)  //переопределение операции вычитания
    {
      for(int i=0; i<2; i++)
      for(int j=0; j<2; j++)
        x[i][j]-=y.x[i][j];
      return *this;
    }
```

Тогда для объектов a,b,c,d,e,f типа Matrix2 можно записать любые выражения с операциями + и -. Например

```
a+b-c+d-e;
//это выражение приведет к последовательному вызову функций
a.operator+(b) //a=a+b
a.operator-(c)  //a=a-c
a.operator+(d) //a=a+d
a.operator-(e)  //a=a-e
```

Выражение *this означает, что функция возвращает ссылку на данный объект.

Для данного класса можно записать следующее выражение:

```
a=c-d+b;
```

В этом случае

```
c.operator-(d) //c=c-d
c.operator+(b)  //c=c+b
a=c;
```

При выполнении операции присваивания будет использован механизм побитового копирования объектов. В данном случае выполнение этого механизма не приведет к побочным эффектам, т.к. класс не имеет динамических составляющих (см.)

Как видно из примера результат операции накапливается в первом объекте выражения.

А второй операнд передается в члены функции operator+() и operator-() копированием.

Если мы хотим чтобы при выполнении операций + и – значения объектов не менялись, то необходимо возвращать результат операции копированием. Тогда необходимо переписать члены функции следующим образом:

```
Matrix2 operator+(Matrix2 y)  //переопределение операции сложения
    {
      Matrix2 tmp; //временный объект для хранения результата сложения
      for(int i=0; i<2; i++)
      for(int j=0; j<2; j++)
        tmp.x[i][j]= x[i][j]+y.x[i][j];
      return tmp; //вернуть результат
     }
Matrix2 operator-(Matrix2 y)  //переопределение операции вычитания
    {
     Matrix2 tmp; //временный объект для хранения результата вычитания
      for(int i=0; i<2; i++)
      for(int j=0; j<2; j++)
        tmp.x[i][j]= x[i][j]+y.x[i][j];
        return tmp;
    }
тогда
Matrix2 v(1,2,3,4),s(2,2,2,2),f(4,3,2,1),x;
x=v+s-f;
```

Для вычисления результата будут произведены следующие действия:

1. tmp1=v.operator+(s)

2. Вызов конструктора копирования и создания временного объекта для хранения промежуточного значения. Если не определен конструктор копирования явно, то вызывается неявно конструктор копирования побитовый. Обозначим этот временный объект tmp1.

3. tmp2=tmp1.operator-(f)

4. x=tmp2;

5. Удаление временных объектов tmp1 и tmp2.

В случае с наличием динамических элементов в объектах класса необходимо переопределять конструктор копирования и операцию присваивания. Например

```
#include <iostream.h>
#include <string.h>
class Mystring
{
  int size; //размер строки
  char *str; //указатель содержащий строку
  char nameobj[10]; //имя объекта для учебных целей
  public:
    Mystring() { strcpy(nameobj,"tmp"); Out("Конструктор(empty)",""); size=0; str=NULL; }
    Mystring(char *n) { strcpy(nameobj,n); Out("Конструктор(name)",""); size=0; str=NULL;
}
    Mystring(char *s,char *n) { strcpy(nameobj,n); Out("Конструктор(str)","");size=strlen(s);
str=new char[size+1];  strcpy(str,s); }
    Mystring(Mystring &s) { strcpy(nameobj,"tmp"); Out("Конструктор(copy)",s.nameobj);
size=s.size; str=new char[size+1];  strcpy(str,s.str); }
    Mystring operator+(Mystring &s)
     {
       Mystring res("res");
       cout<<">>Сложение "<<nameobj<<".operator+("<<s.nameobj<<")\n";
       res.size=size+s.size;
       res.str=new char[res.size+1]; //выделить память под конкатенацию строк
       if(str!=NULL) strcpy(res.str,str); //копируем первую
       else *res.str='\0'; //записать ноль -пустая строка
       if(s.str!=NULL) strcat(res.str,s.str);//присоединяем вторую
       return res;
    }
    Mystring operator=(Mystring s)
     {
       cout<<">>Присвоение "<<nameobj<<".operator=("<<s.nameobj<<")\n";
       if(str!=NULL) delete str;
       size=s.size;
       str=new char[size+1]; //выделить память
       if(s.str!=NULL) strcpy(str,s.str); //присваиваем
       else *str='\0'; //записать ноль -пустая строка
       return *this;
    }

void Show() { if(size>0) cout<<str<<"\n"; }
~Mystring() { Out("Деструктор",""); if(size>0) delete str; }
private: //эта функция предназначена для внутренних целей
void Out(char *s,char *n) { cout<<">>"<<s<<" объекта "<<nameobj<<"  "<<n<<"\n"; }
}; //конец описания класса

//использование класса
void main()
{
```

```
 Mystring Hello("Привет","Hello");
 Mystring All(" Вам","All");
 Mystring x("x");
  x=Hello+All;
  x.Show();
 }
```

Листинг программы

```
>>Конструктор(str) объекта Hello
>>Конструктор(str) объекта All
>>Конструктор(name) объекта x
>>Конструктор(name) объекта res
>>Сложение Hello.operator+(All)
>>Конструктор(copy) объекта tmp  res
>>Деструктор объекта res
>>Присвоение x.operator=(tmp)
>>Конструктор(copy) объекта tmp  x
>>Деструктор объекта tmp
>>Деструктор объекта tmp
Привет Вам
>>Деструктор объекта x
>>Деструктор объекта All
>>Деструктор объекта Hello
```

Это пример учебный и показывает внутренние механизмы вычисления выражений с помощью переопределенных операций. Важно отметить, что компилятор генерирует код, при котором автоматически создаются объекты для хранения промежуточных значений в вычислениях. В листинге это явно видно создано два временных объекта для хранения промежуточных значений. В обоих случаях вызывается конструктор копирования и затем соответственно, деструктор (эти объекты носят имя tmp).

Поскольку функция член operator+() возвращает объект, то можно вызвать другие члены функции определенные в этом классе. Например,

```
Mystring V("Вася","V"),M("+Маша","M"),R("=Любовь","R");
(V+M+R).Show();
```

При выполнении этого кода будет выдана следующая строка:

```
Вася+Маша=Любовь
```

Что будет если мы запишем:

```
X=Y+"Привет";
```

где X и Y принадлежат классу Mystring. В этом случае компилятор сгенерирует ошибку, поскольку строковая константа не является объектом класса Mystring. Для такого случая в класс Mystring необходимо добавить новую функцию, перегружающую операцию +:

```
Mystring operator+(char *s)
    {
     Mystring res("res"),
     Mystring  scon(s,"scon"); //создаем объект Mystring с заданной строкой
     cout<<">>Сложение с константой
"<<nameobj<<".operator+("<<s<<")\n";
     res=*this+scon; //сложение для двух объектов уже известно для класса
     return res;
  }
```

Тогда можно записать

```
Mystring x("x"), y("Начало","y");
x=y+" раз"+" два"+" три";
x.Show();
....
```

При выполнении этого фрагмента x примет значение "начало раз два три" и выдано на печать. При этом три раза будет вызвана функция operator+(char *):

```
y.operator+("раз")
tmp.operator+("два")
tmp.operator+("три")
x.operator=(tmp) //операция присвоения.
```

Что будет если мы запишем:

```
x="раз улыбка"+"два улыбка";
```

В этом случае будет сгенерирована ошибка, и в C++ нет механизма, который бы определил функцию operator+(char *, char *). В таких случаях можно поступить следующим образом:

```
x=x+"раз улыбка"+"два улыбка";
```

В этом случае все будет ok!

Не забыть про добавление новых функций с Matrix2 operator+(int)

Перегрузка операторов new и delete

Глобальные операторы ::operator new() and ::operator new[]() могут быть перегружены. Каждый перегруженный глобальный оператор должен быть уникальным. Тем не менее можно использовать различные глобальные операторы new в разных файлах исходного текста.

Для классов также можно переопределить операторы распределения памяти. Перегрузка операторов new и delete нужна для организации альтернативного механизма распределения памяти.

Перегрузка оператора new

Оператор new может по определению принимает только один аргумент — размер необходимого блока памяти, однако при переопределении можно в операторе new передавать несколько аргументов. Определенный пользователем оператор new должен возвращать адрес блока памяти описанный как void* первым параметром должен быть размер требуемого блока в байтах (size_t – unsigned). Для того чтобы перегрузить оператор new необходимо использовать прототипы, объявленные в заголовочном файле new.h (каталог INCLUDE) . Запись перегружаемых операторов new следующая:

```
void * operator new(size_t Type_size)    // для переменных
{
  ....
}
void * operator new[](size_t Type_size)  //для массивов
{
  ....
}
```

Важно запомнить также, что операторы new перегружаются как статические функции, это означает что нельзя использовать обращение к членам класса и к указателю this. Это очевидно, т.к. обращение к new делается в момент, когда память под объект только будет выделяется.

Задание: можно ли явно не задавая класс памяти, определить его в момент создания объекта.

Перегрузка оператора delete

Глобальные операторы ::operator delete() и ::operator delete[]() не могут быть перегружены. Однако можно их перегрузить для конкретного класса. Определенный пользователь оператор delete должен быть объявлен как void и может иметь один или два аргумента, где первый — это указатель типа void, второй, необязательный, size_t — размер блока освобождаемой памяти. Для класса T может быть определено более одной версии операторов T::operator delete[]() и T::operator delete(). Для того чтобы перегрузить операторы delete, необходимо использовать следующие прототипы:

```
void operator delete(void *Type_ptr, [size_t Type_size]);     // для переменных
void operator delete[](size_t Type_ptr, [size_t Type_size]);  // для массивов
```

Примеры перегрузки операторов

В этом примере рассмотрено простейшее применение перегрузки операторов new и delete для класса целых векторов размерностью 3. Причем вместо динамической в операторе new используется статическая память.

```
#define SIZEMEM 10000
int memory[SIZEMEM];  //блок статической памяти для выделения памяти под объекты
int top=0; //указатель на свободные ячейки
void  Out(char *s) { cout<<">>"<<s<<"\n"; } //функция вывод строки
class X {  //класс целых векторов размера 3
 int vec[3];
   public:
   void* operator new(size_t size)  //перегрузка оператора new
   {  int *ptr=&memory[top];
     top+=3;
    Out("operator new");
    return ptr;
}
   void operator delete(void* p) { Out("operator delete"); } //перегрузка delete
   X() {Out("Конструктор()"); vec[0]=vec[1]=vec[2]=0; }
   X(int x1, int x2, int x3) { Out("Конструктор(*,*,*)"); vec[0]=x1; vec[1]=x2; vec[2]=x3; }
   ~X() { Out("Деструктор для X");  }
   void Show() { cout<<vec[0]<<"-"<<vec[1]<<"-"<<vec[2]<<"\n"; }
};

void main()
{
 X *ptrA, *ptrB;
 ptrA=new X(1,2,3);
 ptrB=new X(5,6,7);
```

```
 ptrA->Show();
 ptrB->Show();

 delete ptrA;
 delete ptrB;
}
```

Листинг программы

```
>>operator new              //распределение памяти для ptrA
>>Конструктор(*,*,*)       //вызов конструктора для объекта ptrA
>>operator new              //распределение памяти для ptrB
>>Конструктор(*,*,*)       //вызов конструктора для объекта ptrB
1-2-3                      //функция Show для ptrA
4-5-6                      //функция Show для ptrB
>>Деструктор для X         //вызов деструктора для объекта ptrA
>>operator delete          //вызов оператора delete для ptrA
>>Деструктор для X         //вызов деструктора для объекта ptrB
>>operator delete          // вызов оператора delete для ptrB
```

Пример №2

Рассмотрим более сложный пример. Здесь предлагается создать следующий механизм управления памятью:

1. Первоначально выделяется достаточно большой блок динамической памяти (здесь можно использовать и блок статической памяти) И из него формируется список свободных блоков памяти. В каждом элементе списка хранится размер блока в байтах и адрес следующего блока. Вначале в этом списке всего один этот блок.

2. Алгоритм выделения памяти следующий. Ищется первый блок подходящего размера, если размер больше, то он разделяется на два, адрес блока заданного размера возвращается, в оставшейся части изменяется размер. Если размер блока совпадает с запрашиваемым размером, то этот блок удаляется из списка. Если в результате поиска не удается найти подходящий блок, то возвращается значение NULL.

3. Алгоритм удаления (утилизации) работает следующим образом: определяется адрес и размер удаляемого блока памяти, далее, этот блок вставляется в список свободных блоков для повторного использования.

Рассмотрим программную реализацию этого механизма

```
union MyBlock //класс для описания элемента списка свободных блоков
{
  char buf[1]; //блок памяти
  struct
  {
```

```
  int size;   //размер блока
  MyBlock *next; //адрес следующего блока
 } e;
public:
  MyBlock*& Next() { return e.next; }  //взять/присвоить следующий
  int& Size() { return e.size; }     //взять/присвоить размер
  void* Addres() { return buf; }     //взять адрес блока
  void* operator[](int i) { return &buf[i]; }  //взять адрес при дроблении
};
```

Класс MyAlloc обеспечивает механизм управления памятью, описанный выше

```
class MyAlloc
{
 char *mem; //Описание кучи
 MyBlock *list; //список блоков
 int sizelist;  //количество блоков
 public:
  MyAlloc() { //создание кучи
   cout<<"Create mem\n";
   mem=list=(MyBlock *)new char[10000];
   list->size()=10000;
   list->Next()=NULL;
   sizelist=1;
 }
  MyAlloc(int sizemem) {   //первоначальное выделение памяти один раз
обращение к ОС
   cout<<"Create mem\n";
   mem=list=(MyBlock *)new char[sizemem];
   list->Size()=sizemem;
   list->Next()=NULL;
   sizemem=1;
  }
 ~MyAlloc() { delete []mem;  } //деструктор
 void* Alloc(int size) //выделение памяти размером size
  {
   //поиск подходящего блока
   MyBlock *prev=NULL; //адрес предыдущего блока
   for(MyBlock *li=list; li!=NULL; li=(MyBlock*)li->Next())
    {
     if(li->Size()==size) //размер точно совпадает
      {
```

```
       //удалить из списка
      if(prev==NULL) //если это первый элемент списка, то корректируем
указатель списка
       {
        list=li->Next();
       }
      else //иначе, корректируем указатель Next у предыдущего элемента
       prev->Next()=li->Next();
      return li->Addres(); //нашли точно по размеру
     }
    else
    if(li->Size()>size+sizeof(MyBlock)) //нашли больший по размеру
     {
      int all=li->Size(); //взять размер свободного блока
      li->Size()-=size; //корректируем размер
      return (void*)(*li)[all-size]; //вернуть адрес блока требуемого размера
     }
    prev=li; //если не нашли, корректируем значение prev и цикл
продолжаем
   }
  //если выходим из цикла здесь, то требуемого размера в списке не нашли
   cout<<"Ошибка при распределении\n";
   return NULL;
  }
  void Free(void *obj,int size) //утилизация используемой памяти
  {                                    //obj- адрес блока памяти
                                       //size- размер блока
   cout<<"Free\n";
   if(obj==NULL||size<=0) return;
   MyBlock *p=(MyBlock*)obj; //преобразование к типу MyBlock
   p->Next()=list; //организовать Next
   list=p;              //корректировать указатель
   p->Size()=size; //установить размер блока
  }
 void Show() { //вывод списка свободных элементов
   cout<<"Show()\n";
   for(MyBlock *li=list; li!=NULL; li=(MyBlock*)li->Next())
    {
     cout<<"Size "<<li->Size()<<" Next "<<li->Next()<<"  Addres  "<<li-
>Addres()<<"\n";
    }
   }
};
```

```
//Тестовый пример использования этого механизма
void main()
{
  int j;
 void *obj[10];   //массив из 10 указателей
//массив размеров
 int osize[10]= { 100, 2000, 30, 150, 90, 10000, 50 , 30, 120, 500 };
 cout<<"Начало\n";
 MyAlloc x(10000); //выделить память
 //x.Show();
 for(int i=0; i<10; i++){ obj[i]=x.Alloc(osize[i]); }
 x.Show();
 //cin>>j;
 for(int i=0; i<10; i++)    x.Free(obj[i],osize[i]);
 x.Show();
 obj[0]=x.Alloc(1000);
 x.Show();
}

//использование предложенного механизма для класса векторов
MyAlloc mem; //память должна быть выделена до использования
соответствующего класса Y.

class Y {
 int vec[3];
   public:
   //перегрузка операторов new и delete
   void* operator new(size_t size) {   return mem.Alloc(size);}
   void operator delete(void* p) {  mem.Free(p,sizeof(Y)); }
   Y() {Out("Конструктор()"); vec[0]=vec[1]=vec[2]=0; }
   Y(int x1, int x2, int x3) { vec[0]=x1; vec[1]=x2; vec[2]=x3; }
   Set(int x1, int x2, int x3) { vec[0]=x1; vec[1]=x2; vec[2]=x3; }
   ~Y() { Out("Деструктор");  }
   void Show() { cout<<vec[0]<<"-"<<vec[1]<<"-"<<vec[2]<<"\n"; }
};
//Использование класса Y
void main()
{
 Y *ptr[2];    //обявить массив указателей
 cout<<"начало";
 ptr[0]=new Y(); //создать два объекта типа Y
 ptr[1]=new Y();
 mem.Show(); //посмотреть список свободных блоков
```

```
ptr[0]->Set(1,2,3); //присвоить значения векторам
ptr[1]->Set(3,2,1);
ptr[1]->Show(); //показать значения второго вектора
delete ptr[0]; //удалить первый вектор
delete ptr[1]; //удалить второй вектор
mem.Show(); //посмотреть список свободных блоков
}
```

Рассмотренный выше пример показывает возможности C++ для создания и использования гибких (своих) механизмов распределения памяти. Этот пример является одним из простейших вариантов. Перечислим его основные недостатки:

1. Выбирается в списке блок первый подходящий, а не самый подходящий, это приводит к дроблению памяти.

2. Поиск линейный, сродни полному перебору. Можно отсортировать этот список по длине блоков.

3. Не используется механизм сборки мусора — объединение блоков в один блок, где это возможно.

4. Не хранится информация об используемых блоках памяти, это дает возможность хранить адреса и длину выделенной памяти.

Доводы в пользу своего механизма распределения памяти:

1. Обеспечить более эффективное использование ОП, отказаться от свопингов и прочих стандартных механизмов, которые иногда минутами тасуют страницы.

2. Обеспечить контроль за множеством объектов, которые созданы динамически. На стадии отладки это может выявить множество ошибок с использованием объектов через указатели.

3. Обеспечить правильную утилизацию всех динамических объектов.

Пример №3

Этот пример показывает использование операторов new с несколькими параметрами (взят из файла помощи Cbuilder). В данном примере описан класс Alpha, в котором перегружены new и delete. Но в данном варианте не происходит распределения памяти, а в момент создания объекта подсовывается уже существующая память с другой структурой.

```
#include <iostream>
using std::cout; //используется стандартный поток
using std::endl; //сивол перевод строки "\n";
class Alpha {
  union { //рассматривается объект как символ или строка символов
```

```
   char  ch;
   char  buf[10];
   };
public:
  Alpha(char c = '\0') : ch(c) { //конструктор для символа
    cout << "character constructor" << endl;
    }
  Alpha(char *s) { //конструктор для переменной
    cout << "string constructor" << endl;
    strcpy(buf,s);
    }
  ~Alpha( ) { cout << "Alpha::~Alpha() " << endl; } //деструктор
  void * operator new(size_t, void * buf) { //опреатор new
    return buf;
    }
};
```

В данном классе new переопределен так, что он не выделяет новой памяти, а объект накладывается на выделенную ранее память, адрес которой передается через параметр.

```
void main() {

  char *str = new char[sizeof(Alpha)];    //выделить память с помощью стандартного new

  // Place 'X' at start of str.
  Alpha* ptr = new(str) Alpha('X'); //создать объект Alpha и наложить его на str
  cout << "str[0] = " << str[0] << endl;

  // Вызвать деструктор
  ptr -> Alpha::~Alpha();

  // создать другой объект
  ptr = new(str) Alpha("my string");
  cout << "\n str = " << str << endl;

  // вызвать деструктор класса
  ptr -> Alpha::~Alpha();
  delete[] str; //удаление символьного массива
  }
```

Листинг программы
character constructor
str[0] = X
Alpha::~Alpha()
string constructor
 str = my string
Alpha::~Alpha()

Перегрузка унарных операций

Можно перегружать как префиксные, так и постфиксные унарные операции путем объявления нестатических членов класса без параметров, или записывая обычную функцию с одним аргументом. Если символ @ представляет унарную операцию, то записывая @x или x@ можно интерпретировать как или x.operator@() или operator@(x). здесь все зависит от того как объявлены эти функции. Правила здесь следующие:

если operator++ или operator—объявлены как члены класса без параметров, или они объявлены как простые функции с одним параметром, то это префиксные операции. Если эти operator++ или operator—объявлены как члены класса с одним параметром типа int или объявлены как простые функции с двумя параметрами, первый из которых принадлежит классу, а второй int, то считаются что эти операции постфиксные.

Рассмотрим примеры:

```
#include <iostream.h>
class A
{
 int a;
 public:
  A() { a=0; }
  A(int i){a=i;}
 //префиксная операция ++
  A& operator++() { cout<<"Префиксная ++ \n"; a++; return *this; }
//постфиксная операция ++
  A& operator++(int) { cout<<"Постфиксная ++\n"; a++; return *this; }
//префиксная операция --
A& operator--() { cout<<"Префиксная -- \n"; a--; return *this; }
//постфиксная операция --
  A& operator--(int) { cout<<"Постфиксная --\n"; a--; return *this; }
//перегрузка унарного минуса
  A& operator+() { cout<<"+OP\n"; a++; return *this; }
//перегрузка унарного минуса
  A& operator-() { cout<<"-OP\n"; a++; return *this; }
```

```
//перегрузка унарного &(операции взятия адреса
  A& operator&() { cout<<"&OP\n"; a++; return *this; }
//перегрузка взятия значения *
  A& operator*() { cout<<"*OP\n"; a++; return *this; }
  };
//примеры использования

void main()
{
 A x;
 *x; //вызов  функции x.operator*()
&x; //вызов  функции x.operator&()
++x;// вызов  функции x.operator++()
x++;// вызов  функции x.operator++(int)
--x;//вызов функции x.operator--()
x--;// вызов  функции x.operator--(int)
-x; //вызов  функции x.operator-()
+x; //вызов  функции x.operator+()
+x++; //вызов  функции x.operator++(int), а затем x.operator+()
++x++; //вызов  функции x.operator++(int), а затем x.operator++()
}
```

В тех случаях, когда постфиксные операции явно не перегружены, компилятор перегружает префиксные как постфиксные и выдает соответствующее сообщение.

Перегрузка операции //

Перегрузка операции [] довольно проста. Важно помнить, что при перегрузке этой операции необходимо указывать только один параметр. Может быть несколько вариантов перегрузки операции [] в одном классе. Рассмотрим примеры.

Пример №1

```
#include <iostream.h>

class Vector //класс целых векторов
{
 int *vec; //вектор
 int size;  //размерность вектора
 public:
   Vector(int size) { vec=new int[size]; Vector::size=size; }
```

```
  int& operator[](int i) { if(i<0||i>=size) i=0; return vec[i]; }
 ~Vector() { delete []vec; }
 void Show() { cout<<"vec["<<size<<"] "; for(int i=0; i<size; i++)
cout<<vec[i]<<" "; cout<<"\n"; }
};
```

Внимательно посмотрите объявление перегрузки

```
int& operator[](int i) { ... return vec[i]; }
```

Функция возвращает ссылку на заданный элемент вектора, это означает, что эта операция может быть как справа, так и слева от знака присваивания. Например,

```
Vector x(20);
x[5]=20; //пятому элементу вектора x присвоить значение 20
j=x[10]; //переменной j присвоить значение 10 элемента вектора x
```

Пример №2

В этом примере показано возможность определения классов с множеством границ, таких как матрицы и тензоры и пр., и соответственно необходимо использование нескольких квадратных скобок для указания элемента. Рассмотрим класс целых матриц.

```
class Matrix //класс целых матриц определен как массив векторов
{
 Vector **matrix; //здесь будет храниться матрица
 int m,n; //размеры матрицы
 public:
  Matrix(int m,int n) //конструктор
   {
    Matrix::m=m; Matrix::n=n;
   matrix=new Vector*[m]; //создать вектор указателей типа Vector
   for(int i=0; i<m; i++) matrix[i]=new Vector(n); //создать вектор для каждого элемента
   }
 int M() { return m; } //определить размер строки
 int N() { return n; }  //определить размер столбца
 Vector& operator[](int i) {  return *matrix[i]; } //вернуть i-й вектор
 ~Matrix() {
    for(int i=0; i<m; i++) delete matrix[i];  //удалить вектора
    delete []matrix; //удалить вектор указателей
```

```
  }
  void Show() //вывести значения матрицы
  {
   cout<<"---------Matrix["<<m<<"]["<<n<<"]------------\n";
   for(int i=0; i<m; i++) matrix[i]->Show();
   cout<<"-----end matrix -----\n";
  }
};
//Здесь определен тензор, как вектор матриц
class Tensor
{
  Matrix **tensor; //указатель на массив указателей
  int l,m,n; //размеры тензора
  public:
   Tensor(int l,int m,int n) //конструктор
    {
     Tensor::m=m; Tensor::n=n; Tensor::l=l;
    tensor=new Matrix*[l]; //создать вектор указателей на матрицы
    for(int i=0; i<l; i++) tensor[i]=new Matrix(m,n); //для каждого элемента
массива указателей создать матрицу
    }
  int L() { return l; } //вернуть количество матриц
  int M() { return m; } //вернуть размер строки
  int N() { return n; } //вернуть размер столбца
  Matrix& operator[](int i) {  return *tensor[i]; } //вернуть ссылку на i-ю мат-
рицу
  ~Tensor() { //деструктор
     for(int i=0; i<l; i++) delete tensor[i];  //удалить матрицы
    delete []tensor;  //удалить массив указателей
  }
  void Show() //вывести тензор
  {
   cout<<"---------Tensor["<<l<<"]["<<m<<"]["<<n<<"]-----------\n";
   for(int i=0; i<l; i++)
    {
     tensor[i]->Show();
    }
   cout<<"-----end tensor ------\n";
  }
};
//Примеры использования матриц и тензоров
void main()
{
```

```
  Matrix x(2,2); //создать матрицу (2,2)
  int k=0;
  for(int i=0; i<x.M(); i++)
  for(int j=0; j<x.N(); j++)
    {
     x[i][j]=k++; //присвоить значения элементам матрицы
    }
  x.Show(); //показать матрицу

  Tensor z(2,3,4); //создать тензор 2 матрицы размером (3,4)
  int h=0;
  for(int k=0; k<z.L(); k++)
  for(int i=0; i<z.M(); i++)
  for(int j=0; j<z.N(); j++)
    {
     z[k][i][j]=h++; //присвоить значения элементам тензора
    }
  z.Show();  //вывести значения тензора
}
```

Листинг

Рассмотрим внимательнее выражения x[i][j] и z[k][i][j]. В первом случае, для объекта x будет вызвана функция x.operator[](i), которая вернет ссылку на объект типа Vector, далее будет вызвана функция operator[](j) для этого объекта, тогда сделав, подстановки для первого выражения, получим:

```
(x.Matrix::operator[](i)).Vector::operator[](j)
```

Аналогично для выражения z[k][i][j] получим

```
((z.Tensor::operator[](k)).Matrix::operator[](i)).Vector::operator[](j)
```

Последовательность вызова:

Tensor::operator[](k) — возврат ссылки на объект типа Matrix
Matrix::operator[](i) — возврат ссылки на объект типа Vector
Vector::operator[](j) — возврат ссылки на объект типа int

В общем случае может быть в классе может быть определено несколько функций перегружающих операцию []. Например, можно переопределить функцию поиска некоторого имени в таблице (Tabl[Name]) При

этом семантику каждой перегрузки определяет сам программист. В общем случае можно записать:

```
Type1 operator[](Index1 i) { ... }
Type2 operator[](Index2 i) { ... }
Type3 operator[](Index3 i) { ... }
...
```

где Type1, Type2, Type3, Index1, Index2, Index3 — стандартные типы или типы определенные самим программистом.

Перегрузка операции ()

Перегрузка операции вызова функции в некоторых случаях является более эффектной и компактной записи программы. Правила записи здесь следующие:

```
operator()(<список параметров>) { /*тело функции */ }
```

Рассмотрим пример

```
#include <iostream.h>
class Z
{
 int z;
 public:
  Z() { z=0; }
  Z(int i){z=i;}
  void operator()(){ cout<<"x("<<x<<")\n"; }
  Z&  operator()(int x,int y){ cout<<"x("<<x<<","<<y<<")\n"; return *this; }
  Z&  operator()(int n,...) { cout<<"x("<<n<<",..."<<")\n"; return *this; }
  int operator()(char *s) { return strlen(s); }
};
//Использование
void main()
{
 Z x;
 x(); //вызов функции x.operator()()
 x(6,10); //вызов функции x.operator()(int,int)
 x(10,40,200,300,400); //вызов функции x.operator()(int,...)
 x("Привет"); //вызов функции x.operaor()(char*)
 }
```

Перегрузка операций [], () удобна, когда класс имеет несколько членов объектов типа вектор или список.

4.1.17 Импорт объектов

При описании членов классов можно использовать ссылки. Однако, в отличии от указателей, ссылки необходимо проинициализировать в конструкторах. Инициализацию можно осуществить двумя путями. Создать объект с помощью оператора new или передать адрес объекта через параметр конструктора. Например,

```
class V
{
 int key; //ключ определяющий объект создан или импортирован
 int &c; //ссылка на целое
  public:
   V(int &ref):c(ref){ key=0;}  //импорт объекта через ссылку
   V(int *ptr):c(*ptr){key=0;}  //импорт объекта через указатель
   V():c(*new int) {key=1;}     //создание объекта
   ~V() { if(key) delete &c; }  //удаление созданного объекта через ссылку
};
```

Тогда можно записать:

```
static int x;
automated int y;
int *ptr=new int(20);
V x1(y), y1(x), z1(ptr), w();
```

Рассмотрим более сложный пример. В этом примере приведены следующие классы:

X, T — базовые классы с виртуальной функцией Show(); Z — класс импортирующий объект типа X; Y — наследующий классы X и T.

```
#include <iostream.h>
class X //базовый класс
{
 int x;
 public:
  X() { x=0; }
  X(int v) { x=v; }
  virtual void Show() { cout<<"X::Show()\n"; }
};
```

```
class T //базовый класс
{
 double t;
 public:
  T() { t=0.; }
  T(double v) { t=v; }
  virtual void Show() { cout<<"T::Show()\n"; }
};

class Z
{
  X &x; //ссылка на объект типа X
  int Stat; //статус обекта
  enum {CreateObject  //объект создан в конструкторе
          ,ImportObject //объект импортирован
  };
  public:
   Z():x(*new X()){ Stat=CreateObject; } //конструктор создания объекта
   Z(X &v):x(v){ Stat=ImportObject; }    //импорт объекта через ссылку
   Z(X *p):x(*p){ Stat=ImportObject;}    //импорт объекта через указатель
   Z(Z &o):x(o.x){ Stat=ImportObject; } //импорт через конструктор копиро-
вания
   ~Z() { if(Stat==CreateObject) delete &x; } //деструктор в
   void Show() { x.Show(); }
   virtual void New() { cout<<"Новая виртуальная функция\n"; }
};
//создает класс с множественным наследованием
class Y: public X, public T
{
 public:
 Y(int i):X(i),T((double)i){}
 void Show() { cout<<"Y::Show()\n"; }
};

 void Func(T &t) { t.Show(); } //Функция принимающая ссылку типа T
//Использование
void main()
{
  X x(20); //создаем объект типа X
  Y y(30); //создаем объект типа Y
 Func(y); //передаем объект типа Y, функция Show() виртуальна (см. описа-
ние класса T)
             //будет выведено Y::Show()
```

```
 x.Show(); //здесь будет выведено X::Show()
 y.Show(); //здесь будет выведено Y::Show()
 Z z1(x),   //импортируем в z1 объект x;
    z2(y);   //импортируем в z2 объект y
 z1.Show(); // здесь будет выведено X::Show()
 z2.Show(); // здесь будет выведено Y::Show()
 z1.New();
}
```

4.1.18 Идентификация объектов во время выполнения (RTTI)

Для определения типа объекта во время выполнения программы в C++ предусмотрен специальный механизм, обеспечивающий определение типа RTTI (Run Time Type Identification). Кроме этого тип можно определить и для указателей ссылок.

Это делает возможным преобразовать указатель виртуального базового класса в указатель действительного наследуемого класса , который имеет сам объект. Это делается с помощью оператора dynamic_cast.

RTTI механизм также позволяет проверять тип объекта или сравнивать типы двух объектов. Это можно сделать, используя оператор typeid, который проверяет тип объекта записанного в качестве аргумента, а возвращает ссылку на объект типа const type_info, который описывает тип объекта-аргумента.

Можно также определить имя типа, отношение предшествования . В классе type_info имеется два члена функции name() и before(), операции:

==(проверка на равно),

!= (не равно).

Для включения механизма RTTI необходимо включить специальную опцию компилятора, или для описания класса записать ключевое слово __rtti. В исходный текст программы необходимо включить заголовочный файл typeinfo.h. Рассмотрим примеры

```
#include <iostream.h>
#include <typeinfo.h>  //вставляем описание класса type_info
class __rtti X  //включение механизма RTTI
{
 protected:
 int a;
 public:
  X(int b):a(b){}
  virtual void Show() { cout<<a<<"\n"; }
};
class __rtti Y: public X //наследуем класс X
```

```
{
 public:
 Y(int b):X(b){}
 void Show() { cout<<"++"<<a<<"\n"; }
 int Power( ) { return a*a; ) //не виртуальная функция
};
//использование механизма RTTI
void main()
{
 X x(10);
 Y y(20);
 X *ptr=&y;
 Y *yptr;
 cout<<typeid(x).name()<<"\n"; //отпечатать тип объекта x
 cout<<typeid(ptr).name()<<"\n";//отпечатать тип объекта ptr
 if(typeid(x).before(typeid(y))) cout<<"x < y\n"; //класс X описан раньше чем
класс Y
 yptr=dynamic_cast<Y*>(ptr); //проверить и преобразовать указатель к типу
Y
 if(yptr!=NULL) { cout<<yptr->Power()<<"\n"; } //использование указателя
 x.Show();
}
```

Внимание, механизм RTTI нельзя применять для указателей типа void. Например,

```
Y y(10);
void *ptr=&y;
Y *yptr= dynamic_cast<Y*>(ptr); //здесь преобразования не будет
```

Использование механизма RTTI будет показано при описании контейнеров.

Механизм подобный RTTI можно создать самим, это можно следующим образом:

```
#include <iostream.h>
enum typeob { tO,tX,tY,tZ}; //записываем типы всех классов
class TypeObject //создаем  базовый класс
{
  public:
    void* Type(typeob t) { if(t==type) return this; else return NULL; }
    TypeObject() { type=tO; } //установка типа tO объекта
   protected:
```

```
  typeob type; //переменная содержащая тип объекта
};

class X: public TypeObject //первый производный класс
{
  public:
   X() { type=tX; } //установка типа в конструкторе
 //... другие члены
};
class Y: public TypeObject //второй производный класс от TypeObject
{
  public:
      Y() { type=tY; } //установка типа объекта в конструкторе
      //... другие члены
};
class Z: public X //производный класс от класса X
{
  public:
      Z() { type=tZ; }
      //... другие члены
};
void main()
{
 TypeObject *ptr[3];
 X *xptr;
 Z *zptr;
 ptr[0]=new X(); //заполняем массив указателей, адресами конкретных
объектов
 ptr[1]=new Y();
 ptr[2]=new Z();
 if(xptr=ptr[0]->Type(tX)) cout<<"Это X\n"; //здесь будет напечатано "Это
X"
 zptr=(Z *)ptr[0]->Type(tZ); //здесь zptr  будет присвоено значение NULL
 zptr=(Z *)ptr[2]->Type(tZ); //здесь zptr  будет присвоено значение адреса
объекта
 }
```

Для реализации отношения предшествования (before) строится немного более сложный механизм. В каждом классе определяется функция before(int type), в которой вызываются before() базовых классов. Например,

```
class Obj  {
   protected:
```

```
   typeob type;
  public:
    int before(typeob t) { return 0; }
    int AreYou(typeob t) { return (t==tO)?1:0; } //возвращает истина если t это
"Obj"
    //..другие члены
 };
class X: public Obj  {
 public:
 int before(typeob t) { return Obj:: AreYou (t); }
 int AreYou (typeob t) { return (t==tX)?1:0; } //возвращает истина если t это
"X"
 //другие члены
 };
class Z: public X  {
 public:
 int before(typeob t) { return X:: AreYou (t); }
 int AreYou (typeob t) { return (t==tZ)?1:0; } //возвращает истина если t это
"Z"
 //другие члены
 };
class V: public X, public Y  {
 public:
 int before(typeob t) { if(X:: AreYou (t)||Y: AreYou (t) return 1; else return 0; }
 int AreYou (typeob t) { return (t==tV)?1:0; } //возвращает истина если t это
"V"
 //другие члены1
 };
```

4.1.19 Контейнеры

Под контейнером понимается класс, в котором в качестве членов данных используются объекты других классов. Например,

```
class X { ..... };
class Y { X x; ....} //это контейнер для хранения объектов класса X
Конечно, в таком случае лучше использовать механизм наследования
class Y: public X { .... };
```

Однако в случае с вектором, лучше использовать контейнер

```
class VecX
{
```

```
  X vec[10];
  const int size=10;
  public:
  X& operator[](int i) { if(i<size) return vec[i]; return x[0]; }
  int Size() { return size);
};
```

Более интересный вариант контейнера, хранить не сам вектор объектов, а вектор указателей на объект типа базового виртуального класса. Например

```
class MyObject {
virtual void Show()=0;
};
class VecObject
{
  MyObject **vec; //указатель на вектор указателей
  int size;               //размерность вектора
  public:
    VecObject(int sz):size(sz){ vec=new MyObject*[sz]; for(int i=0; i<sz; i++)
vec[i]=NULL; }
    MyObject*& operator[](int i) { if(i<size) return vec[i]; } //ссылка на
указатель
    int Size() { return size; }   //выдать размер вектора
    ~VecObject() { for(int i=0; i<sz; i++) if(vec[i]!=NULL) delete vec[i];  delete
[]vec; }
};
```

Использование данного контейнера VecObject
Класс X наследует класс MyObject

```
class X: public MyObject
{
 int x;
 public:
  X(int t) { x=t ;}
  void Show() { ...}
  int & Val() { return x; }
  int & operator*() { return y; }
};
```

Класс Y также наследует класс MyObject

```
class Y: public MyObject
```

```
{
  double y;
  public:
   Y(double t) { y=t ;}
   void Show() { ...}
   double & Val() { return y; }
   double & operator*() { return y; }

};

//пример использования
void main()
{
VecObject vec(2);
vec[0]=new X(20);
vec[1]=new Y(5.678);
for(int i=0; i<vec.Size(); i++) vek[i]->Show();
}
```

Замечания:

1) деструктор удаляет не только вектор указателей, но и сами объекты, это означает что объекты должны быть созданы с помощью оператора new;

2) объекты можно хранить все, кто наследует класс MyObject, но использовать можно только виртуальные функции. Однако, используя механизм RTTI можно преобразовать указатель базового класса к актуальному (тому, что имеет объект).

Пример использование механизма RTTI в контейнере.

Рассмотрим класс VecObject с включенным механизмом RTTI. Для включения этого механизма нужно явно установить опцию транслятора RTTI или использовать ключевое слово __rtti (см. описание RTTI-механизма).

```
VecObject vec(5);
vec[0]=new X(20);
vec[1]=new Y(5.678);
vec[2]=new Y(25.01);
vec[3]=new X(5);
vec[4]=new Y(5.678);
//....
//Нахождение суммы всех объектов типа Y
```

```
Y *yptr;
double Sum=0.;
for(int i=0; i<vec.Size(); i++)
{
 yptr=dynamic_cast<Y*>(vec[i]);  //преобразование типа  если это возможно
 if(yptr!=NULL) Sum+=yptr->val();  //можно также написать *(*yptr)
}
```

Рассмотрим реализацию контейнера в виде стека, немного видоизменив класс VectorObject.

```
class Vstack
{
 MyObject **stack; //память под стек выделяется как вектор указателей
 int size; //размер памяти выделенный под стек
 int sp; //указатель стека
 public: //интерфес
  Vstack(int sz) :size(sz){ stack=new MyObject*[sz]; sp=0;}
  void Push(MyObject *o) { if(sp<size) stak[sp++]=o; } //здесь кроется
опасность потери объекта
  MyObject* Pull() { if(sp=0) return NULL; else return stack[—sp]; }
 ~Vstack() { delete []stack; } //ответственность по удалению объектов
ложится на программиста
};
 //Примеры использования
Vstack Stack(100) //стек размером 100
Stack.Push(new X(20);
Stack.Push(new Y(10.01);
X *ptr=Stack.Pull();
while((ptr=Stack.Pull())!=NULL) delete ptr; //удаление всех объектов из
стека
```

Недостаток данной организации стека — размер стека фиксирован, вследствие чего стек может переполниться и мы можем потерять объекты. В этом случае можно предложить механизм увеличения размера стека, выделяется память по новый вектор указателей, большего размера, адреса объектов из старого массива переписываются в новый, старый удаляется и в новый добавляется не вместившийся объект.

4.1.20 Списки

Списки играют важную роль в программировании. Под списком понимается множество блоков памяти, связанные между собой. Под связью

здесь понимается адрес некоторого блока, который хранится в другом блоке. В каждом блоке может организована не одна связь, а некоторое их подмножество.

Списки используют тогда, когда размеры сложных объектов заранее не определены, а выделять максимальный фиксированный размер нецелесообразно. Кроме того, списками эффективно можно представлять различные классы графов — деревья и т.д.

Начнем рассмотрение организации списка с рассмотрения простейших

```
class Link //описывает элемент списка
{
 public:
  MyObject *x;
  Link *next;
  Link(MyObject *o,Link *beg):x(o){ next=beg; }
  ~Link() { delete x; }
 };

class List //описывает линейный список
{
 public:
 Link *beg; //указатель списка
  List() { beg=NULL; } //список пуст
 void Add(MyObject *x) {  beg=new Link(x,beg); }
 ~List() { Link*t; while(beg!=NULL) { t=beg; beg=beg->next; delete t; }
};

class ListIterator //класс описывающий навигацию в списке
{
 Link *iter;
 public:
  ListIterator(List *list) { iter=list->beg; }
  void Next() { iter=iter->beg; }
  void Beg() { iter=list->beg; }
  int Null() { (iter==NULL)?1:0; }
  MyObject& Value() ( return *iter; }
  };
//Использование списка
class Name: public MyObject //создаем класс объектов для хранения имен
{
 MyString name;
 Name(char *s):MyString(s){}
```

```
 void Show() { name.Show(); }
};
//
void main()
{
  List list;
//Создание списка
  list.Add(new Name("Вася"));
  list.Add(new Name("Петя"));
  list.Add(new Name("Егор"));
//Вывод списка
 for(ListIterator i(list); !i.Null(); i.Next()) i.Value().Show();  //печать список
имен
}
```

Варианты реализации списка:

1) если все элементы списка однотипные, то можно в элементе хранить указатель типа void и преобразовывать к заданному типу явно и далее использовать механизм RTTI, если имеется механизм наследования;

2) в элементе списка хранить указатель на базовый класс и использовать механизм RTTI для преобразования к конкретному типу;

3) использовать свой механизм определения типа;

4) использовать механизм шаблонов (см. описание шаблонов).

4.1.21 Шаблоны

В реальной практике программирования часто приходится переписывать функции, изменяя в них только описания данных, а алгоритм программы при этом не изменялся.

Например, для нахождения минимального значения:

```
int min(int a,int b) { return (a<b)?a:b; }
int abs(int a) { return (a<0)?-a:a;}
```

В случае если тип данных int заменить на double, то эти функции необходимо переписать с новым типом данных и изменить названия функций. Что, собственно, и делалось в С.

Например, в stdlib описаны функции abs и fabs. В С++ предложен новый механизм генерации программ, который основанный на использовании шаблонов. Шаблон — это текстовая заготовка программы или класса, с некоторыми параметрами. Для получения конкретной программы или класса необходимо установить значения параметров шаблона. В отличии от макросов определяемых с помощью #define, шаблоны учитывают син-

таксис C++. Говорят также о шаблонах как о параметризированных типах или о генераторах. С помощью шаблонов можно получить бесчисленное множество похожих друг на друга классов или функций. Рассмотрим шаблоны для функций.

Шаблоны функций

Шаблоны функций объявляются следующим образом:

tamplate <список параметров> <описание функции с параметрами>.

Например,
для функции abs

```
template <class T>
T abs(T a) { return (a<0)?-a:a; };
```

или для функции min

```
template <class Temp>
Temp min(Temp a, Temp b) { return (a<b)?a:b; };
```

Конечно, можно было бы объявить #define

```
#define min( a, b)   ((a<b)?a:b)
#define abs(a)  ((a<0)?-a:a)
```

Однако при таком определении имеется следующие проблемы:

1. Может возникнуть ситуация сравнивать например объекты с разными типами, например char и struct, что может привести к ошибкам.
2. Если мы запишем некоторый класс:

```
class X {
  public;
   int min(int x, int y); //здесь будет произведена замена на макрос min, ошибка
   double abs(double z); //здесь будет произведена замена на макрос abs, ошибка
//...
};
```

Отсюда следует, что использование шаблонов является наиболее подходящим для подобных случаев. Рассмотрим использование шаблонов, на примере

```
class Y
{
  int y;
   public:
    Y(int k){ y=k; }
     int operator<(Y &t) { return (y<t.y) 1:0; }
//...
};
//Использование шаблонов функций.
void main()
{
  long l=100;
  long k=min(l,20) // будет сгенерирована функция long min(long a,long b)
  Y oy(20),oy2(300);
  Y k=min(oy,oy2) // будет сгенерирована функция Y min(Y a,Y b)
}
```

В некоторых случаях использование шаблона не подходит, в таком случае можно переопределить шаблон-функцию. Например, для шаблона char * min(char*, char*)

будет сгенерирована функция:

```
char * min(char *s1,char* s2)
{
  return (s1<s2)?s1:s2;
}
```

Но нам хотелось не сравнивать адреса, а сравнивать две строки. В этом случае, записывают тело новой функции:

```
char * min(char *s1,char* s2)
{
  return (strcmp(s1,s2)<0)?s1:s2;
}
```

Рассмотри еще примеры шаблонов

```
template <class T>
T min(T a, T b)
```

```
{
  return (a<b)?a:b;
};
void func(int i, char c)
{
  min(i,i) ; //будет сгенерирована min(int,int)
  min(c,c) ; //будет сгенерирована min(char,char)
  min(i,c) //здесь будет сгенерирована ошибка, т.к. типы не совпадают
  min(c,i) //здесь будет также сгенерирована ошибка, т.к. типы не
совпадают
}
```

Для избежания подобной ситуации необходимо перед использованием шаблона объявить функцию с подходящими параметрами, например, для случая описанного выше:

```
template <class T>
T min(T a, T b)
{
  return (a<b)?a:b;
};
int min(int,int); //объявить функцию из шаблона
void func(int i, char c)
{
  min(i,i) ; //будет сгенерирована min(int,int)
  min(c,c) ; //будет сгенерирована min(char,char)
  min(i,c) ; //будет сгенерирована min(int,int)
  min(c,i) ; //будет сгенерирована min(int,int)
}
```

Шаблоны классов

Шаблон класса позволяет задать механизм генерации множества подобных конкретных классов. Рассмотрим пример создания шаблона класса для организации стека.

```
template <class T>
class Stack {
 T *stack; //параметризированный указатель объектов
int size;    //размер стека
int sp;      //указатель стека
public:
 Stack() { stack=NULL; size=0; sp=0; }
```

```
 Stack(int sz) { stack=new T[sz]; size=sz; sp=0; }
void Push(T *d);
T* Pull();
};
//Объявление членов функций вне класса
template <class T>
void Stack<T>::Push(T *d)
{
 if(top<size) stack[top++]=d;
}; //не забудьте про точку с запятой
template <class T>
T* Stack<T>::Pull()
{
 if(top>0) return stack[--top]=d;
 return NULL;
}; //не забудьте про точку с запятой

//использование шаблона
class X { ....}; //объявление некоторого класса
void main()
{
  Stack<int> stack_i(100); //стек для хранения целых чисел
  Stack<X>  stackX(20); //объявление стека для хранения объектов класса X
  int i=20;
  stack_i.Push(i); //поместить в стек значение i
  stack_i.Push(10); //поместить в стек 10
  i=stack_i.Pull(); //взять из стека вершину(10) и присвоить это значение
переменной i
 //Аналогичные действия можно производить и для объектов класса X
}
```

Шаблоны классов могут иметь несколько аргументов. Аргументы могут быть разных типов и иметь значения по умолчанию. Например,

```
tamplate <class T, class V, int size=100> class TwoVector { .....};
```

Для аргументов шаблона, которые не описывают типы, такие как size в приведенном выше примере значения должны быть константными выражениями. Например,

```
   const int N = 128;
   int j=20;
   TwoVector<int,double,N+10> x; //здесь все нормально
```

```
TwoVector<int,char,2*i> y; //здесь ошибка, i не является константой
```

Необходимо при объявлении шаблона класса учитывать, что скобками списка аргументов являются символы «<« и «>«, что в некоторых случаях может привести к ошибкам:

```
TwoVector<int,int,(x>y)?10:200)> xy; //здесь кроется ошибка
```

Список аргументов шаблона закрывается первым символом «>«.

Шаблоны и наследование

Предположим у нас имеется некоторый универсальный список для хранения любых объектов:

```
class UniversalList
{
  public:
   void insert(void *ptr);
void *peek();
//... другие члены
};
```

Теперь нам хотелось бы использовать этот класс для определенного типа данных, можно было бы поступить следующим образом:

```
class ObjList: public UniversalList {
  public:
   void insert(Obj *ptr) { UniversalList::insert(ptr); }
   Obj *peek() { return (Obj*) UniversalList::peek(); }
//... другие члены
};
```

Здесь Obj имя класса, объекты которого будут храниться в списке типа ObjList.

Для создания списка для других классов объектов пришлось заново переписывать описания класса. В этом случае можно предложить шаблон следующего типа:

```
template  <class T>
class List: public UniversalList {
  public:
   void insert(T *ptr) { UniversalList::insert(ptr); }
```

```
  T *peek() { return (T*) UniversalList::peek(); }
//... другие члены
};
```

Тогда можно записать:

List<Obj> oList; //из шаблона создается конкретный класс и объект oList, список для хранения для хранения объектов класса Obj
List<Rect> rList; //из шаблона создается конкретный класс и объект rList, список для хранения для хранения объектов класса Rect.

Другая техника применения шаблонов заключается в следующем:

```
tamplate <class T>
class Base {
//...
private:
 T buf;
//..
};
//
class MyObject: public Base<StrBuf>
{
// ...
};
```

Здесь из шаблона Base создается конкретный класс Base<StrBuf> и далее наследуется классом MyObject.

4.1.22 Обработка исключительных ситуаций

Исключительные ситуации неизбежность при создании больших сложных программ. Обычно такие ситуации возникают при ограниченности ресурсов (например, нехватка оперативной памяти) или неисправности некоторого оборудования и т.д. При программировании без обработки исключительных ситуаций программа просто «зависала» или вылетала, при этом никакой информации об ошибке не выдавалось, кроме того, при работе с такой программой могла произойти потеря данных или даже порча файловой системы или отдельных каталогов и файлов. В язык C++ для обработки исключительных ситуаций были введены специальные механизмы, которые обеспечивают:

1) определить область программного кода, где может возникнуть исключительная ситуация;

2) определить тип ситуации (или исключения), отлавить ее и обработать;

3) генерировать исключение.

1. Область программного кода задается блоком try, синтаксис следующий:

```
try
{
  //область программного кода, которая может генерировать исключение.
}
```

2. Определение типа сгенерированного исключения производится с помощью оператора catch. Этот оператор можно записывать последовательно друг за другом, после описания try-блока. синтаксис этого оператора следующий:

```
try{
//область программного кода, которая может генерировать исключение.
}
catch(Type1 [object1]) {
//операторы обработки исключения типа Type1
}
catch(Type2 [object2]) {
//операторы обработки исключения Type2
}
catch(Type3 [object3]) {
//операторы обработки исключения Type3
}
catch(...) {
//операторы обработки исключения любого другого типа (или
неизвестного типа)
}
```

Здесь в квадратных скобках стоят необязательные параметры, которые могут и отсутствовать при записи оператора catch.

3. Генерация исключения производится с помощью оператора throw, при этом происходит следующее:

1) происходит прерывание выполнения программы;

2) ищется подходящий оператор catch;

3) если обработчик исключения найден, то стек программы безболезненно устанавливается на этот обработчик;

4) управление передаются на найденный обработчик исключения.

Синтаксис оператора throw достаточно простой:

throw [выражение];

Пример №1

throw tobject;

В этом примере происходит прерывание выполнения программы и передача объекта tobject в обработчик исключения, заданный в операторе catch.

Пример №2

throw;

В этом примере показано использование оператора throw для передачи последнего сгенерированного исключения во внешний блок try.

Пример №3

```
void my_func1() throw (TypeX, TypeY)
{
   // Тело функции
}
```

В этом примере показано использование ключевого слова throw для указания, какие исключения могут быть сгенерированы внутри данной функции. Внутри функции могут быть сгенерированы только исключения типа TypeX и TypeY. Все остальные исключения будут генерироваться как unexpected (неожиданные).

Пример №4

```
void my_func2() throw ()
{
    // тело функции
}
```

В этом случае, если в теле функции будет сгенерировано какое-либо исключение, оно будет игнорировано.

Пример №5

В этом примере меня значения переменной i, можно проиграть разные ситуации, при i равной 1 будет сгенерировано целое исключение и бу-

дет осуществлен переход на обработчик целого исключения. При i равной 2 будет сгенерировано char исключение, и произойдет переход на обработчик char исключения. При других значениях i программа выполнит cout<<"i не равно ни 1, ни 2"; и завершит выполнение без исключений.

```
#include <iostream>
void main()
{
 int i=2;    //установка варианта действия 1 — int исключение, 2 — char исключение, др. нет исключения
  try{
    if(i==1) throw i;   //генерируем целое исключение
  else
    if(i==2) throw 'x'; //генерируем char исключение
   cout<<"i не равно ни 1 ни 2\n"; // выполнение программы если не будет исключения
  }
  catch(int t) //обработчик int исключения
   {
     cout<<"int исключение t\n";
   }
   catch(char t) //обработчик char исключения
   {
    cout<<"char исключение t\n";
   }
 }//конец main
```

Пример №6

```
class TError {
int code;
};
```

4.1.23 Потоки ввода/вывода

В разделе №X дано определение потока, как программной абстракции, обеспечивающей передачу данных (информации) от источника к приемнику. Источник данных помечен префиксом i(n) приемник — префиксом o(ut).

Рассмотрим потоки для вывода информации (потоки-приемники — stream output). для выводы информации в поток используется перегруженная операция <<. Левый объект в этой операции должен быть типа ostream, а в правой части должен быть объект предназначенный для вывода.

В классах ostream заранее определены операции вывода (<<) для всех основных типов C++ (char, short, int, long, char*, float, double, void*). Сравнивая с функцией printf можно получить один и тот же результат:

```
int k, long L;
cout<<k<<"—"<<L;
```

Для указателей (void *) можно записать:

```
int j;
cout<<&j;
```

Форматирование потоков осуществляется с помощью множества флажков форматирования определенных в классе ios (базовый класс для потоков). Форматирующие флажки следующие:

```
public:
enum {
  skipws,  //игнорирование управляющих символов во водном потоке
  left, //выравнивание по левой границе при выводе
  right, //выравнивание по правой границе при выводе
  internal, //пропуски после знака или индикатора основания
  dec,   //преобразование к десятичному  представлению
  oct,   //преобразование к восьмеричному  представлению
  hex, // преобразование к шестнадцатеричному представлению
  showbase, //показать основание числа в выводном потоке
  showpoint, //показать десятичную точку для вывода с плавающей запятой
  uppercase, // вывод в верхнем регистре для шестнадцатеричного
представления
  showpos, //вывод целого положительного числа со знаком плюс
  scientific, //вывод числа плавающего формата в экспоненциальном виде
(E)
  fixed,
  unitbuf,
 stdio
};
```

Манипуляторы

Простейший путь форматирования в выводном потоке это использовать специальные функции подобно операциям, называемые манипуляторами. Манипуляторы принимают поток в качестве ссылки и возвращают

ссылку на этот же поток. Поэтому манипуляторы можно вставлять в последовательность операций вывода. Рассмотрим пример,

```
#include <iosteram.h>
#include <iomanip.h>  //заголовок необходимый для манипуляторов
void main()
{
 int i=1234, j=6789, k=10;
cout<<setw(6)<<i<<j<<k<<j;
cout<<"\n";
cout<<setw(6)<<i <<setw(6)<<j<<setw(6)<<k;
}
```

Листинг программы

```
__12346789106789
__1234__6789____10
```

Здесь манипулятор setw(int w) задает ширину вывода для следующего элемента вывода, в нашем примере это 6 символов. Причем по умолчанию выравнивание производится по правой границе.

Основные манипуляторы перечислены в следующей таблице:

Манипулятор	**Действие**
dec	установить флаг преобразования в десятичный формат
hex	установить флаг преобразования в шестнадцатеричный формат
oct	установить флаг преобразования в восьмеричный формат
ws	Пропуск управляющих символов при вводе (whitespace)
endl	вставить символ новой строки \n и очистить поток
flush	очистить поток
setbase(int n)	преобразование в заданный параметром формат , n принимает значения (0,8,10,16) При n равное нулю устанавливается формат преобразования по умолчанию
resetiosflags(long f)	очистить флаги преобразования установленные в f
setiosflags(long f)	установить флаги форматирования установленные в f
setfill(int c)	Установить символ заполнения c
setprecision(int n)	Установить точность вывода для плавающего формата, n задает количество цифр, после запятой
setw(int n)	Установить ширину поля вывода

Ввод

Потоковый ввод похож на потоковый вывод, только операция ввода данных перегружает операцию правого сдвига (>>). Левый операнд имеет тип ***istream***, а справа стоят объекты, операция ввода для которых определена. Операция ввода определена для всех основных типов C++ (см. операцию вывода для основных типов).

По умолчанию все управляющие символы (whitespace — пробелы, перевод строки, табуляция и пр.) пропускаются. Рассмотрим пример,

```
int x;
double z;
cin>>x>>z; //ввод значений для переменных x и z
```

Когда будет выполняться оператор ввода, программа пропустит все управляющие символы, затем прочитает значение для i, затем опять пропустит управляющие символы и прочитает значение для переменной z.

Для типа char также будут пропускаться управляющие символы. Для ввода строк символов можно явно указывать размер вводимой строки, например,

```
char str[SIZE];
cin.width(sizeof(str));
cin>>width;
```

Потоковый ввод/вывод для классов определенных пользователем

Потоковый ввод и вывод для классов создается с помощью перегрузки соответствующих операций ввода (>>) и вывода (<<). Рассмотрим пример,

```
#include <iostream.h>

class X
{
  int size;
 double val;
 char name[20];
public:
X() { size=0; val=0.; name=NULL; }
friend istream operator>>(istream& stream, X& x);
friend ostream operator<<(ostream& stream, X& x);
```

```
// другие члены
};
//определение операции вывода для класса x
ostream operator<<(ostream& stream, X& x)
{
 stream<<" size="<<x.size<<" val="<<x.val<<" name="<<x.name<<"\n";
return stream;
}
//определение операции ввода
istream operator>>(istream& stream, X& x)
{
 stream>>x.size>>x.val>>name;
return stream;
}
//пример использования
void main()
{
 X test;
 int i;
 cin>>i>>test; //ввод переменной i, затем  ввод значений объекта test
 cout<<test;  //вывод объекта test
}
```

Простой файловый ввод/вывод

Для файлового вывода используется класс ***ofstream***, который наследует свойства класса ***ostream***. Для файлового ввода используется класс ***ifstream***, который в свою очередь наследуют свойства класса ***istream***. Классы файловых потоков также содержат конструкторы для создания файлов и управлением файлового ввода/вывода. Для работы с файловыми потоками необходимо включить заголовочный файл fstream.h

Рассмотрим простейший пример копирования файлов.

```
#include <fstream.h>
void main()
{
 char ch;
ifstream src("исходый.txt"); //открытие файла для ввода из него производится в /                                //конструкторе
ofstream dst("копия.txt");  //открытие файла для вывода в него производится в /                                //конструкторе
if(src==NULL) { cout<<"Ошибка при открытии исходного файла\n"; exit(0);
}
```

```
if(dst==NULL) { cout<<"Ошибка при открытии копии файла\n"; exit(0); }
while(dst && src.get(ch)) //чтение символа из файла src
  dst.put(ch) //запись символа в dst
//закрытие файлов производится в деструкторах
}
```

Пример №2

```
#include <fstream.h>
void main()
{
 int vect[5]= { 2, 4, 55, 100, 60 };
 int num=20;
double x[3]= { 10.5, 100.567, 0.003 };
ofstream report("отчет.txt");  //открытие файла для вывода в него  в  тексто-
вой моде
if(report==NULL) { cout<<"Ошибка при открытии файла\n";  }
else
 {
   report<<"Отчет №"<<num<<"\n";
   report<<"Вектор\n";
   for(int i=0; i<5; i++) report<<vect[i]<<" ";
   report<<"\n";
   report<<"Числа с плавающей запятой\n";
   for(int i=0; i<3; i++) report<<x[i]<<" ";
   report<<"\n";
  report<<"Все!!!\n";
 }
//закрытие файлов производится в деструкторах
}
```

Пример №3

Чтение числового массива типа double из тестового файла, первое число в этом файле целое и показывает реальный размер массива чисел, записанный в этом файле. Например,

```
5
5.5 6.6  7.7  8.8  9.9
Имя этого файла "числа.txt"
#include <fstream.h>
void main()
{
```

```
 int size;
 double *x;
 ifstream src("числа.txt"); //открытие файла для ввода из него чисел
if(src==NULL) { cout<<"Ошибка при открытии исходного файла\n";  }
else
{
   src>>size; //чтение размера массива
   cout<<size<<"\n"; //вывод на экран
   x=new double[size]; //распределение памяти под массив
   for(int i=0; i<size; i++) src>>x[i]; //чтение элементов массива
   for(int i=0; i<size; i++) cout<<x[i]<<"\n"; //вывод элементы массива на
терминал
   delete []x;
//закрытие файлов производится в деструкторах
 }
}
```

При выполнении программы с учетом файла "числа.txt", на экране будет

```
5
5.5
6.6
7.7
8.8
9.9
```

Пример №4

Чтение и вывод тестового файла на терминал. Используются следующие члены-функции:

istream& getline(char *str, int MaxSize,char cDelim=’\n’) — чтение строки символов из файла до разделителя cDelim. По умолчанию cDelim — перевод строки.

int eof() — функция член, определяющая конец файла; Возвращает 1, если достигнут конец файла. В противном случае 0.

istream& get(char& ch) — ввод символа из потока.

```
#define MAXSIZE 200 //максимальный размер
void ShowTxtFile(char *NameFile)
{
 char ch='\0';
 int key=1;
```

```
 char s[MAXSIZE];
 ifstream src(NameFile); //открытие файла для ввода из него
if(src==NULL) { cout<<"Ошибка при открытии исходного файла\n";  }
 while(key) //выводить пока key не равен нулю
 {
  for(int i=0; i<24; i++) //вывести на экран 24 строки или меньше
  {
   src.getline(s,MAXSIZE); //читать текущую строку из файла
   cout<<s<<"\n"; //вывести строку на терминал
   if(src.eof()) { key=0; break;} //если конец файла, то key=0; выход из про-
смотра файла
  }
  cin.get(ch); //ожидание реакции пользователя, ввод символа
 }

//закрытие файлов производится в деструкторах
}
```

Конструкторы позволяют объявлять файловый поток без указания имени файла. Позже можно связать с объявленным файловым потоком конкретный файл. Например,

```
ostream outfile;
...
outfile.open("Платежи"); //открыть файл
//обработка файла
output.close();  //закрыть файл
//другие операции
output.open("Сводки"); //переиспользование файлового потока
//....
output.close();
//...
```

Режимы работы с файлами

При открытии файла можно явно указать режим работы с файлом . Ниже перечислены эти режимы:

Мода	Действие
ios::app	Разрешает дописывать данные в уже существующий файл
ios::ate	Устанавливает указатель в конец файла
ios::in	Открыть файл для ввода информации из него (ifstream по умолчанию)

Мода	Действие
ios::out	Открыть файл для вывода в него информации (ofstream по умолчанию)
ios::binary	Открыть файл в двоичной моде
ios::nocreate	Открывает, если файл существует, в противном случае ошибка
ios::trunc	Очистить файл, если существует
ios::noreplace	Открыть, если файл не существует

Для работы с файловыми потоками существует две моды: текстовая и двоичная. Текстовая мода устанавливается по умолчанию, если явно не указан флаг ios::binary (двоичная мода). И означает, что в файле хранятся строки символов и соответственно можно использовать операции >> и <<. В двоичной моде данные хранятся в двоичном виде, т.к. они представлены в оперативной памяти. Все приведенные выше примеры демонстрируют работу с файлами в тестовой моде. Рассмотрим работу с файлами в двоичной моде. Для этого рассмотрим некоторые важные члены-функции:

void open(const char *name, int mode,int prot=filebuf::openprot);

Эта функция открывает файл с именем name, с заданной модой, prot-спецификация защиты, можно не устанавливать.

void close() ;

Эта функция закрывает файловый поток.

int good();

Эта функция возвращает не нулевое значение, если нет ошибок в потоковых операциях

int fail();

Эта функция возвращает ненулевое значение, если в потоковых операциях имееся ошибка

int eof();

Эта функция возвращает ненулевое значение, если в поток достиг конца файла

ostream& write(const char*, int n);

ostream& write(const signed char*, int n);
ostream& write(const unsigned char*, int n);

Эти функции вставляют в поток последовательность байт, адрес начала задан в первом параметре, количество байт записи указывается во втором параметре. Запись осуществляется относительно текущего положения указателя файла. При выполнении операции указатель файла перемещается на записанное количество байт.

istream& read(char*, int);
istream& read(signed char*, int);
istream& read(unsigned char*, int);

Эти функции читают из потока последовательность байт, адрес буфера, куда будет записана последовательнось байт в первом параметре, количество байт чтения указывается во втором параметре. Чтение осуществляется относительно текущего положения указателя файла. При чтении указатель файла перемещается на прочитанное количество байт.

long tellg();

Эта функция определяет текущее значение указателя файла.

istream& seekg(streampos pos);
istream& seekg(streamoff offset, seek_dir dir);

Эти функции перемещают указатель файла. В первом случае функция seekg перемещается в абослютную позицию. Тип strreampos похож на тип long.

Во втором случае, offset задает относительное значение иможет иметь отрицательное или положительное значение. A dir задает точку отсчета:

ios::beg — от начала файла
ios::cur — от текущего положения указателя
ios::end — от конца файла.
streamoff задает тип long. (опреден как typedef long streamoff)

Рассмотрим примеры
Пример №1. Создание файла и запись в него матрицы.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
```

```
void main()
{
 double x[3][2] = { 10., 20., 30., 0.1, 0.2, 0.3 }; //объявление матрицы
 fstream file;  //объявить файловый поток
 char name[10]="matrix XX"; //имя матрицы
 int m=3, n=2; //размерности матрицы
 file.open("marix.dat",ios::out|ios::binary); //открыть файловый поток на
запись
 if(file.fail()) //если ошибка то выход
  {
   cout<<"error open\n"; exit(0);
  }
 file.write(name,10); //писать 10 байт имя
 file.write((char*)&m,sizeof(int)); //писать m
 file.write((char*)&n,sizeof(int)); //писать n
 for(int i=0; i<m; i++)
 for(int j=0; j<n; j++)
 {
  file.write((char*)&x[i][j],sizeof(double)); //последовательно писать значение
элементов
 }
 file.close();//закрыть файловый поток . Файл с матрицей создан
}
```

Пример №2. Программа чтения матрицы из файла , созданного программой из предыдущего примера

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void main()
{
 double x;
 fstream file;
 char name[10];
 int m, n;
//открыть файловый поток для чтения  как двоичный
 file.open("marix.dat",ios::in|ios::binary);
 if(file.fail())
  {
   cout<<"error open\n"; exit(0);
  }
 file.read(name,10); //читать 10 байт имя
```

```
 file.read((char*)&m,sizeof(int)); //читать m
 file.read((char*)&n,sizeof(int)); //читать n
 cout<<name<<"["<<m<<"]["<<n<<"]\n";
 for(int i=0; i<n; i++)
 {
 for(int j=0; j<m; j++)
 {
  file.read((char*)&x,sizeof(double)); //читать элемент матрицы из файла
  cout<<x<<" ";
 }
 cout<<"\n";
 }
 file.close();
 cin>>n;
}
```

Пример №3

Пример простейшей программы ведения базы данных.

В этом примере имеется класс MRecord, который является базовым для классов записей.

Класс MBase описывает функции для работы с базой данных. Класс MHeader описывает заголовок базы и основные функции для работы с ним.

Структура базы данных следующая: заголовок и последовательность записей. Заголовок фиксированнной длинны и хранит тип файла, название организации, версию, количество записей в базе и пр. Записи в базе пронумерованы, первая запись имеет номер 0, вторая 1 и т.д. Эти номера можно назвать индексами или адресами записей. Запись представлет собой строку в таблице (смотри организацию реляционных баз данных). Кокретный вид записи создается наследованием базового класса MRecord, например,

```
MPersonalRecord: public MRecord
{
 //описание строки таблицы
  char surname[20];
  char name[20];
  int    sex;
  double weight;
 //...другие члены-данные
public:
 int Size() { return sizeof(MPersonalRecord); }
//..другие члены-функции
};
```

Возно отметить, что описание записи не должно содержать динамических элементов (указателей), что существенно влияет на sizeof.

Рассмтрим основные функции класса MBase:

Create(name,sizerecord) — создать базу данных, указать имя и размер записи, при этом функция создает файл и записывает туда заголовок.

Open(name) — открыть базу данных с именем name, при этом функция читает заголовок из из файла.

Close() — закрыть базу данных, при этом производится перезапись заголовка базы.

Add(record) — добывать запись в базу, запись добавляется в конец базы данных

Read(index,record) — читать но номеру запись из базы

Rewrite(index,record) — переписать содержимое записи с номером index в базе данных

Delete(index) — удалить запись логически, установить значение байта удаления.

Undelete(index) — отменить удаление

Heap(record) — произвести физическое удаление удаленных записей из базы.

В программе используется функция access(filename,flags) описанная в io.h

Эта функция обеспечивает проверку доступа к файлу.

```
#include <iostream.h> //необходимые заголовочные файлы
#include <fstream.h>
#include <io.h>
#include <mem.h>
#include <string.h>
#define SIZETYPEBASE 10
#define SIZEORG 80
#define SIZEVER 30
////////////////////////////////
class MRecord  //базовый класс описания записи(строки таблицы)
{
 public:
 enum { ActualRecord, //запись нормальная
       DeleteRecord  //запись удаленная
      };
 char isDelete; //байт логического удаления записи из базы
 virtual long Size(){ return sizeof(MRecord);}  //размер записи в байтах
 MRecord() { isDelete=ActualRecord; }  //Нормальная запись
 MRecord(char ch) { isDelete=DeleteRecord; } //удаленная запись
 virtual void Show() { cout<<isDelete; }
```

```
};
////////////////////////////////////
class MHeader  //описание заголовка базы данных
{
 public:
 char TypeBase[SIZETYPEBASE];   //кваливикатор базы
 char Organization[SIZEORG];    //организация
 char Version[SIZEVER];         //версия
 long SizeBase;                 //размер всег файла
 long SizeRecord;               //размер записи
 long TotalRecord;              //количествр записей
 long TotalDeleteRecord;        //количество удаленных логически записей
 public:
  MHeader() {  TypeBase[0]=Organization[0]=Version[0]='\0';
               SizeBase=SizeRecord=TotalRecord=TotalDeleteRecord=0;
            }
  void Set(long szRecord)//Установить значения для заголовка
   {
    memset(this,0,sizeof(MHeader)); //такими операциями пользоваться надо
осторожно!
    SizeRecord=szRecord;            //установить длину записи
    SizeBase=sizeof(MHeader);       //определить размер заголовка
    TotalRecord=0;                  //всего записей 0
    TotalDeleteRecord=0;            //удаленных записей 0
    strcpy(TypeBase,"MBase");       //установить тип базы
    strcpy(Organization,"BobaInternational");  //установить имя владельца
    strcpy(Version,"Version v007");  //версия программы
   }
  void Show() {
cout<<Organization<<"\n"<<Version<<"\n"<<SizeRecord<<"\n";}
  void ReadHeader(fstream& BaseHandle) //читать заголовок
  {
   BaseHandle.seekg(0L,ios::beg);      //установить указатель файла в начало
   BaseHandle.read((char*)this,sizeof(MHeader)); //читать заголовок
  }
 void WriteHeader(fstream& BaseHandle) //писать заголовок в файл
  {
   BaseHandle.seekg(0L,ios::beg);      //установить указатель файла в начало
   BaseHandle.write((char*)this,sizeof(MHeader)); //записать заголовок
   }
};
////////////////////////////////////////////////
// класс базы данных
```

```
//////////////////////
class MBase: public MHeader
{
   char BaseName[120]; //имя файла
   long RecordNumber;  //номер прочитанной записи
   fstream Handle;     //дескриптор файла
  public:
  MBase():MHeader(){;}
  MBase(char *name);
  void  Create(char *Name, long size_rec); //создать базу
  void  Open(char *Name);                  //открыть базу
  void  Close();                           //закрыть базу
  void  Read(long num, MRecord& rec);      //читать запись
  void  Add(MRecord &rec);                 //добавить запись в конец файла
  void  Rewrite(long num, MRecord& rec);   //переписать запись
  void  Delete(long num);                  //удалить запись
  void  UnDelete(long num);                //востановить удаленную запись
  void  Heap(MRecord& rec);                //сборка мусора
  int Exits() { return(FileExists(BaseName)); }//база существует?
  private:
   int FileExists(char *filename) //определить существование файла
    {
     return (access(filename, 0) == 0);
    }
};
//////////
MBase::MBase(char *name) //конструктор базы для открытия
{
 if(FileExists(name)) //если файл существует
 {
  //
  Handle.open(name,ios::in|ios::out|ios::binary); //открыть
  ReadHeader(Handle);  //читать заголовок
  strcpy(BaseName,name);  //сохранить имя базы
 }
 else;//Error
}
void MBase::Open(char *name) //операция открыть файл
{
 if(FileExists(name)) //если файл существует
 {
  //
  Handle.open(name,ios::in|ios::out|ios::binary); //открыть
```

```
  if(Handle==NULL) ; //error
  ReadHeader(Handle);    //читать заголовок
  strcpy(BaseName,name);  //сохранить имя базы
 }
 else ; //error
}
//////////////////
void MBase::Create(char *name, //имя базы данных
                   long size_rec  //длинна записи
                   )
{
 if(FileExists(name)) ; //Error
 else
 {
  Handle.open(name,ios::out|ios::in|ios::binary);
  if(Handle==NULL); //Error
  else
   {
    Set(size_rec); //установить значения для заголовка
    WriteHeader(Handle); //записать заголовок
    strcpy(BaseName,name); //сохранить имя базы
   }
 }
}
//////////////////////////
void MBase::Close()  //закрыть базу
{
 if(Handle!=NULL)
  {
   WriteHeader(Handle);  //записать заголовок
   Handle.close();       //закрыть файл
  }
}
//////////////////
void MBase::Add(MRecord &rec)  //добавть запись в конец файла
{
 if(Handle!=NULL)
  {
   TotalRecord++; //увеличить количество записей
   Handle.seekg(0L,ios::end); //укаазтель файла педвинуть в конец файла
   Handle.write((char*)&rec,(int)SizeRecord); //добавить запись
  }
}
```

```
//////////////////
void  MBase::Read(long num, MRecord& rec) //читать запись с номером num
из базы
{
  if(Handle!=NULL)
  {
   //указатель файла передвинуть в начало указанной в num записи
   Handle.seekg((long)sizeof(MHeader)+num*SizeRecord,ios::beg);
   Handle.read((char*)&rec,(int)SizeRecord); //читать запись
  }
}
//////////////////
void MBase::Rewrite(long num,MRecord &rec) //заменить запись в базе с
номером num
{
 if(Handle!=NULL)
  {
   //указатель файла передвинуть в начало указанной в num записи
   Handle.seekg(sizeof(MHeader)+num*SizeRecord,ios::beg);
   //заменить апись на новое значение
   Handle.write((char*)&rec,(int)SizeRecord);
  }
}
//////////////////
void MBase::Delete(long num)
{
 if(Handle!=NULL) //логическое удаление записи в базе
  {
   MRecord del('*'); //байт с отметкой удаления
   MRecord cur;
   Handle.seekg((long)sizeof(MHeader)+num*SizeRecord,ios::beg);
   Handle.read((char*)&cur,sizeof(MRecord)); //прочитать байт логического
удаления
   if(cur.isDelete!=MRecord::DeleteRecord) //если эта запись уже не удалена
   {
    Handle.seekg(-(long)sizeof(MRecord),ios::cur); //вернуть в исходную
позицию
    Handle.write((char*)&del,sizeof(MRecord)); //записать байт удаления
    TotalRecord--;                  //количество записей уменьшить
    TotalDeleteRecord++;            //количество удаленных записей увеличить
   }
  }
}
```

```
///////
void MBase::UnDelete(long num) //функция востановдения логически
удаленной записи
{
 if(Handle!=NULL)
  {
   MRecord undel, cur;
   Handle.seekg((long)sizeof(MHeader)+num*SizeRecord,ios::beg);
   Handle.read((char*)&cur,sizeof(MRecord)); //читать байт
   if(cur.isDelete==MRecord::DeleteRecord)   //если эта зпись удалена
   {
    Handle.seekg(-(long)sizeof(MRecord),ios::cur);
    Handle.write((char*)&undel,sizeof(MRecord)); //писать байт астуальности
    TotalRecord++;              //количество записей увеличить
    TotalDeleteRecord--;        //количество удаленных записей уменьшить
   }
  }
}
//
void MBase::Heap(MRecord& rec) //сборка мусора- физическое удаление
записей
{
 MBase tmp;
 tmp.Create("###Mbase.tmp",SizeRecord); //создать временную базу
 long n=TotalRecord+TotalDeleteRecord;  //определить общее количество в
старой базе
 for(long i=0; i<n; i++) //просмотр всех записей старой базы
  {
   Read(i,rec); //читать запись
   if(rec.isDelete==MRecord::ActualRecord) //Запись актуальна
                        tmp.Add(rec); //добавить в новую
  }
 tmp.Close(); //закрыть новую
 Close();     //закрыит старую
 //unlink(BaseName); //Удаление старой базы
 rename(BaseName,"Old"); //переименовать старую базу
 rename("###Mbase.tmp",BaseName); //переименовать новую
 Open(BaseName); //открыть обновленную со старым именем!
}
//////////////////////////////
class MRecordBook: public MRecord   //создаем описание записи для базы
хранения записей о книгах
{
```

```
 public:
 char Book[140]; //название
 char Author[60]; //автор
 char Year[20];   //год издания
 char SizePage[20]; //количество страниц
 char Total[10];    //имеется в наличии
 virtual void Show() { cout<<"Show\n"; cout<<Book<<" "<<Author<<"
"<<Year<<" "<<SizePage<<" "; }
 long Size() { return sizeof(MRecordBook); } //размер записи
 MRecordBook() {
 memset(Book,0,sizeof(Book));
 memset(Author,0,sizeof(Author));
 memset(Year,0,sizeof(Year));
 memset(SizePage,0,sizeof(SizePage));
 memset(Total,0,sizeof(Total));
 }
 MRecordBook(char* B,char*A,char *Y,char* SP,char* T)
 {
 memset(Book,0,sizeof(Book));
 memset(Author,0,sizeof(Author));
 memset(Year,0,sizeof(Year));
 memset(SizePage,0,sizeof(SizePage));
 memset(Total,0,sizeof(Total));
 strcpy(Book,B);
 strcpy(Author,A);
 strcpy(Year,Y);
 strcpy(SizePage,SP);
 strcpy(Total,T);
 }
};
//пример использования базы
void main()
{
 MRecordBook rec;
 //Описание записей
 MRecordBook rec0("Как растить капусту","Заяц","1950","200","100");
 MRecordBook rec1("Как готовить зайцев","Волк","1960","350","300");
 MRecordBook rec2("Как обманывать волков","Лиса","1970","125","10");
 MRecordBook rec3("Как задолбать всех","Дятел","1980","400","1");
 MRecordBook rec4("Если чешутся лапки...","Ёжик","1985","201","5");
 MRecordBook rec5("Продажа недвижимости","Улитка","1987","125","0");
 MRecordBook rec6("Как
заключатьдоговора","Медведь","1961","112","11");
```

```
 MRecordBook rec7("Сыр. Производство и
потребление","Ворона","1890","125","3");
MRecordBook rec8("Как носить очки","Мартышка","1950","125","8");
MRecordBook rec9("Как написать генератор","Кручинин","1950","125","8");

 MBase Books;
 Books.Create("MyBooks",rec.Size()); //создать базу
 //добавить записи в базу
 Books.Add(rec0);
 Books.Add(rec1);
 Books.Add(rec2);
 Books.Add(rec3);
 Books.Add(rec4);
 Books.Add(rec5);
 Books.Close();  //закрыть базу

 Books.Open("MyBooks");  //открыть существующую базу
 //вывод базы данных
 for(int i=0; i<Books.TotalRecord; i++)
  {
   Books.Read(i,rec);
   rec.Show();
  }
 //добавить еще две записи
 Books.Add(rec6);
 Books.Add(rec7);
 //удалить записи сномером 1 и 2
 Books.Delete(1);
 Books.Delete(2);
 //сборка мусора (физическое удаление записей из базы )
 {
  MRecordBook rec;
  Books.Heap(rec);
 }
 //вывод всех записей из базы
 for(int i=0; i<Books.TotalRecord; i++)
  {
   Books.Read(i,rec);
   rec.Show();
  }
 Books.Close(); //закрыть базу данных
}
```

Описанный выше пример является учебным и показывает возможности работы с файловыми потоками. Хотя основные идеи организации программ для ведения баз данных в этом примере отражены. Для дальнейшего развития этого примера необходимо ввести понятие ключей, индексных файлов и соотвественно построение и редактирование индексных файлов. Дадим кратко основные идеи:

1. Клуч, это поле или выражение записанное из нескольких полей строки таблицы(записи) по которому упорядочивается множество записей базы данных.

2. Индексный файл — файл, содержащий номера записей упорядоченные относительно некоторого ключа.

Обычно индексные файлы представлены в виде B-деревьев, которые обеспечивают эффективные механизмы упорядочения, поиска и операций вставки, удаления и слияния индексов.

4.2 Реализация классов средствами Си

Язык Си является подмножеством языка программирования C++.

Покажем, что понятие класса является надстройкой на средствами Си.

Для этого необходимо ввети понятие указателя (vptr) на таблицу виртуальных функций (VTABLE).

Пример открытой организации класса

```
#include <conio.h>
#include <iostream.h>
//вспомогательный класс, позволяющий видеть работу локальных деструкторов
class Wait{
 public:
 ~Wait() {
  cprintf("Press any key to exit:");
  while(!kbhit());
  }
};
Wait wait_input;
//описываем абстрактный класс
class Base{
public:
 virtual int X()=0;
 virtual void Y(int i)=0;
 virtual void Z(int i,char *s )=0;
 virtual void One()=0;
};
//Наследуем абстрактный класс и
//переопределяем виртуальные функции
```

```
class Dir1: public Base{
public:
 int  X() { cout<<"Dir1::Hello\n"; return 1;}
 void Y(int i) { cout<<"Dir1::Great "<<i<<"\n";}
 void Z(int i,char *s ) { cout<<"Dir1::Good day "<<i<<" "<<s<<"\n";}
 void One(){;}
 Dir1() { cout<<"Constructor Dir1\n"; }
 ~Dir1() { cout<<"Destructor Dir1\n"; }
};
//Создаем аналог класса средствами C
typedef int  (*XFUNC)(void*);
typedef void (*YFUNC)(void*,int x);
typedef void (*ZFUNC)(void*,int x,char *s);
#define SIZEVTABLE  3
enum {Func_X,Func_Y,Func_Z};
typedef struct class_Dir2 { //описание структуры
        void *vptr;      //указатель на таблицу виртуальных функций
        int x;           //данные класса
        } Dir2;
//Методы класса Dir2, первым параметром в методах, должен быть указатель
//на структуру  типа Dir2 (здесь This не ключевое слово а идентификатор)
int  X(Dir2 *This) { cout<<"Dir2::"<<This->x<<"Hello\n"; return 1;}
void Y(Dir2 *This,int i) { cout<<"Dir2::Great "<<i<<"\n";}
void Z(Dir2 *This,int i,char *s ) { cout<<"Dir2::Good day "<<i<<" "<<s<<"\n";}
//
void *AF_VTabl[SIZEVTABLE]={ //таблица виртуальных функций (vtabl)
     (void *)X,
     (void *)Y,
     (void *)Z
    };
//Конструктор класса
void Dir2_Constructor(Dir2 *This,int x){
 This->vptr=&AF_VTabl[0];   //инициализируем vptr
 This->x=x;
 cout<<"Constructor Dir2\n";
}
void Dir2_Destructor(Dir2 *This){
 cout<<"Destructor Dir2\n";
}

////////////////////////////////
// Программа тестирования
////////////////////////////////
void TestFunc(Base *x){

  x->X();
  x->Y(100);
  x->Z(100," Go Go Go !!!");
 }
void TestFunc2(void **x)
```

```
{
  void **vptr=(void**)(*x);
 ((int(*)(void*))(vptr[0]))((void*)x);
 ((YFUNC)(vptr[Func_Y]))((void*)x,200);
 ((ZFUNC)(vptr[Func_Z]))((void*)x,200,"Great !!");
}
////////////////////////////
//Создаем другой базовый класс
class Base2{
 virtual void W()=0;
 virtual void One()=0;
};
////////////////////////////////////
class Dir3: public Base, public Base2{
              //этот класс имеет две vtabl
              // и соотвественно два vptr
 int i;
 public:
 void One() { cout<<"one\n"; }
 void W() { cout<<"Dir3::Base2\n"; }
 int  X() { cout<<"Dir3::Hello\n"; return 1;}
 void Y(int i) { cout<<"Dir3::Great "<<i<<"\n";}
 void Z(int i,char *s ) { cout<<"Dir3::Good day "<<i<<" "<<s<<"\n";}
 Dir3() { cout<<"Constructor Dir3\n"; }
 ~Dir3() { cout<<"Destructor Dir3\n"; }
};
///////////
void TestFunc3(void **x){
  void **vptr=(void**)(*x);         //первый интерфейс
  void **vptr2=(void**)(x[1]);      //второй интерфейс
 ((int(*)(void*))(vptr[0]))((void*)x);
 ((YFUNC)(vptr[Func_Y]))((void*)x,400);
 ((ZFUNC)(vptr[Func_Z]))((void*)x,600,"Great !!");
 ((void(*)(void*))(vptr2[0]))((void*)x);
 ((void(*)(void*))(vptr2[1]))((void*)x); //это функция One
 ((void(*)(void*))(vptr[3]))((void*)x);  //это тоже функция One
}

/*************************************************************************
***/
void main()
{
 int x;
 //создаем объекты
 cout<<"*******  Create object *********\n";
 Dir1 y;  //здесь коструктор запускается автоматически
 Dir2 object;  //для этого объекта необходимо явно запустить конструктор
 Dir3 z;
 Dir2_Constructor(&object,800);
 cout<<"*******  Test1(Dir1)    **********\n";
```

```
//передаем через казатель на базовый класс объект класса Dir1
TestFunc(&y);
cout<<"*******  Test1(Dir2)    ************\n";
//передаем через казатель на базовый класс объект класса Dir2
TestFunc((Base*)&object);
cout<<"*******  Test2(Dir1)    ************\n";
TestFunc2((void**)&y);
cout<<"*******  Test2(Dir2)    ************\n";
TestFunc2((void**)&object);
cout<<"*******  Test3(Dir3)    ************\n";
TestFunc3((void**)&z);
cout<<"*******  End *********\n";
 //вызовы функций через соотвествующие элементы таблицы
//((int (*)(AF*))(v.func[Func_X]))(&v);
//((void (*)(AF*,int))(v.func[Func_Y]))(&v,100);
//((void (*)(AF*,int,char *))(v.func[Func_Z]))(&v,100,"Kozlik");
Dir2_Destructor(&object);
}
```

4.3 Реализация объктно-ориентированного подхода для создания приложения в Windows

В настоящее время в практике создания приложений для Windows используются объктно-ориетированые базы и библиотеки классов. Например, библиотека классов фирмы Майкрософт (MFC), объектная библтотека для Windows (OWL) и т.д.

Рассмотрим на следующем простом примере создание объектно-ориентированной библиотеки для Windows. Ниже описан класс MWindow, который предсназначен для описание объекта «окно». Он записан в следующем файле hMWindow.h:

```
#ifndef hMWindow
#define hMWindow
#define STRIPT
#include <windows.h>
class MWindow
{
 public:
 static char szMyAppName[40];
 static HINSTANCE hInstance;
 static int RegisterClass(char *szAppName,HINSTANCE hInstance);
 static long WINAPI MWindow::WndMainProc(HWND hwnd,UINT Message,WPARAM
wParam, LPARAM lParam);
 HWND hwnd;
 MWindow(HWND par,MWindow *ptr) {  //конструктор окна
 hwnd=CreateWindow(szMyAppName,
            "",
            WS_OVERLAPPEDWINDOW,
```

```
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            par,
            NULL,
            MWindow::hInstance,
            ptr
            );
   ShowWindow(hwnd,SW_SHOWNORMAL);
  }
  virtual void OnCreate();
  virtual void OnPaint(HDC dc);
  virtual void OnCommand(int command);
  virtual void OnDestroy();
};
//#ifndef hMWindowStatic
//#define hMWindowStatic
//extern char MWindow::szMyAppName[40];
//extern HINSTANCE MWindow::hInstance;
//#endif

#endif
```

Ниже представлена реализация виртуальных функция класса MWindow

```
//////////////////////////////////////////
#include "MWindow.h"
char MWindow::szMyAppName[40];
HINSTANCE MWindow::hInstance;
///////////////
#define WM_NEWCREATE_MWINDOW (WM_USER+1)
void MWindow::OnCreate() {

//hListBox=CreateWindow("LISTBOX","Прими",WS_CHILD|WS_VISIBLE|WS_BORDE
R|WS_VSCROLL,10,70,600,400,hwnd,HMENU(-1),hInstance,NULL);
}
//;}//MessageBox(NULL,"","",MB_OK);}
void MWindow::OnCommand(int command) {;}
void MWindow::OnPaint(HDC dc){;}
void MWindow::OnDestroy() {;}

int MWindow::RegisterClass(char *szAppName,HINSTANCE hInstance){
  WNDCLASS wndclass;
  lstrcpy(MWindow::szMyAppName,szAppName);
  MWindow::hInstance=hInstance;
  HWND hwnd;
  wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
  wndclass.lpfnWndProc   = (WNDPROC)MWindow::WndMainProc ;
```

```
 wndclass.cbClsExtra    = 0 ;
 wndclass.cbWndExtra    = 0 ;
 wndclass.hInstance     = hInstance ;
 wndclass.hIcon         = LoadIcon( NULL, IDI_APPLICATION );
 wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
 wndclass.hbrBackground = (HBRUSH)(GetStockObject(WHITE_BRUSH));
 wndclass.lpszMenuName  = NULL;
 wndclass.lpszClassName = szAppName ;
 if(::RegisterClass (&wndclass)==NULL) { //ErrMsg("WNDCLASS");
 exit(0);}
}
```

Оконная процедура представлена в классе MWindow как статическая функция в которой на сообщение WM_CRAETE проиходит связывание между оконным объектом и оконной процедурой, адрес объекта передается через структуру CreateStruct и адрес обекта связывается с окном с помощью функции SetWindowLong.

```
long WINAPI MWindow::WndMainProc(HWND hwnd,UINT Message,WPARAM wParam,
LPARAM lParam)
{
 switch (Message) {
 case WM_CREATE:
  {
   MWindow *x= (MWindow *)((CREATESTRUCT *)lParam)->lpCreateParams;
   SetWindowLong(hwnd,GWL_USERDATA,(LONG)x);
   PostMessage(hwnd,WM_NEWCREATE_MWINDOW,NULL,NULL);
    //MessageBox(NULL,"","",MB_OK);
    return 0;
  }
 case WM_NEWCREATE_MWINDOW:
 {
  MWindow *win=(MWindow *)GetWindowLong(hwnd,GWL_USERDATA);
  win->OnCreate();
  return 0;
 }
 case WM_COMMAND:
  {
   MWindow *win=(MWindow *)GetWindowLong(hwnd,GWL_USERDATA);
   win->OnCommand(wParam);
   return 0;
  }
```

Таким образом, наследуя класс MWindow и переопределяя виртуальные функции, можно строить довольно сложные приложения.

ЛИТЕРАТУРА

1. Словарь иностранных слов. — М.: Рус. яз., 1989. — 624 с.

2. Кнут, Д. Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы / Д. Кнут. — М.: Мир, 1976. — 736 с.

3. Лавров, С.С. Основные понятия и конструкции языков программирования / С.С. Лавров. — М.: Финансы и статистика, 1982. — 80 с.

4. Брукс, Ф. Мифический человеко-месяц, или как создаются программные системы / Ф. Брукс. — М.: Символ-Плюс, 2001. — 304 с.

5. Вельбицкий, И.В. Технология программирования / И.В. Вельбицкий. — Киев: Техніка, 1984. — 250 с.

6. Ван Тассел, Д. Стиль, разработка, эффективность, отладка и испытание программ / Д. Ван Тассел. — М.: Мир, 1985. — 281 с.

7. Григас, Г. Начала программирования / Г. Григас. — М.: Просвещение, 1987. — 112 с.

8. Грис, Д. Наука программирования / Д. Грис. — М.: Мир, 1984. — 416 с.

9. Иванников, В.П. О преподавании программирования // Компьютерные инструменты в образовании. — 2003. — № 4.

10. Richard, H. Thayer: Software System Engineering: A Tutorial. IEEE Computer 35(4), 2002 p. 68–73.

11. Соммервилл, И. Инженерия программного обеспечения / И. Сомервил. — М.: Вильямс, 2002. — 624 с.

12. Философский энциклопедический словарь. — М.: Сов. энциклопедия, 1983. — 840 с.

13. Жоголев, Е.А. Технологические основы модульного программирования // Программирование. — 1980. — № 2. — С. 44–49.

14. Турский, В. Методология программирования / В. Турский. — М.: Мир, 1981. — 264 с.

15. Дал, У. Структурное программирование / У. Дал, Э. Дейкстра, К. Хоор. — М.: Мир, 1975. — 247 с.

16. Йодан, Э. Структурное проектирование и конструирование программ / Э. Йодан. — М.: Мир, 1979.

17. Лингер, Р. Теория и практика структурного программирования / Р. Лингер, Х. Миллс, Б. Уатт. — М.: Мир, 1982.

18. Хоггер, К. Введение в логическое программирование / К. Хоггер. — М.: Мир, 1988.

19. Логическое программирование : сб. статей. — М.: Мир, 1988.

20. Адаменко, А.Н. Логическое программирование и Visual Prolog / А.Н. Адаменко, А.М. Кучуков . — СПб.: БХВ-Петербург, 2003. — 992 с.

21. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ : [пер. с англ.] / Г.Буч. — 2-е изд. — М.: Бином, СПб., 2000. — 560 с.

22. Фридман, А.Л. Основы объектно-ориентированного программирования на языке Си++. — М.: Горячая линия-Телеком: Радио и связь, 1999. — 208 с.

23. Элиенс, А. Принципы объектно-ориентированной разработки программ / А. Элиенс. — М.: Вильямс, 2001. — 496 с.

24. Ларман, К. Применение UML и шаблонов проектирования / К. Ларман. — М.: Вильямс, 2001. — 496 с.

25. Смольянинов, А.В. Визуальное программирование в учебном процессе / А.В. Смольянинов, В.А. Калмычков // Современные технологии обучения: сб. науч.-метод. тр. — СПб.: СПбГЭТУ, 1997. — Вып. 3. — С. 8–19.

26. Фаронов, В.В. Delphi. Программирование на языке высокого уровня : учеб. для вузов / В.В. Фаронов. — СПб.: Питер, 2004. — 640 с.

27. Шеферд, Д. Программирование на Microsoft Visal C++ NET : [пер. с англ.] / Д. Шеферд. — М.: Русская Редакция, 2003. — 928 с. — ISBN 5-7502-0225-9.

28. Филд, А. Функциональное программирование / А. Филд, П.М. Харрисон. — Мир: 1993. — 637 с.

29. Хендерсон, П. Функциональное программирование. Применение и реализация / П. Хендерсон. — М.: Мир, 1983. — 349 с.

30. Городняя, Л.В. Основы функционального программирования / Л.В. Городняя. — М.: Интернет-университет информационных технологий, 2004. — 280 с.

31. Тыугу, Э.Х. Концептуальное программировани / Э.Х. Тыугу. — М.: Наука, 1984. — 255 с.

32. Андерсен, Р. Доказательство правильности программ / Р. Андерсен. — М.: Мир, 1982.

33. Браун, П. Макропроцессоры и мобильность программного обеспечения / П. Браун. — М.: Мир, 1977. — 253 с.

34. Мобильность программного обеспечения / под ред. Брауна. — М.: Мир, 1980. — 336 с.

35. Астелс, Д. Практическое руководство по экстремальному программированию / Д. Астелс, Г. Миллер, М. Новак. — М.: Вильямс, 2002. — 320 с.

36. Бек, К. Экстремальное программирование: планирование / К. Бек, М. Фаулер. — СПб.: Питер, 2003. — 144 с.

37. Бек, Л. Экстремальное программирование / Л. Бек. — СПб.: Питер, 2002. — 224 с.

38. Чеппел, Д. Технологии ActiveX и OLE : [пер. с англ.] / Д. Чеппел. — М.: Русская Редакция, 1997. — 320 с.

39. Бокс, Д. Сущность технологии COM. Библиотека программиста / Д. Бокс. — СПб.: Питер, 2001. — 400 с.

40. Оберг, Р.Д. Технология COM+. Основы и программирование / Р.Д. Оберг. — М.: Вильямс, 2000. — 480 с.

41. RATIONAL Quatrani T. Visual Modeling with Rational Rose 2000 and UML . — USA: Addison-Wesley, 2000. — 256 с.

42. Элиенс, А. Принципы объектно-ориентированной разработки программ / А. Элиенс. — М.: Вильямс, 2001. — 496 с. (CORBA).

43. Кролл, П. Rational Unified Process — это легко: руководство по RUP для практиков / П. Кролл, Ф. Крачтен ; пер. с англ. С.М. Лунина. — М: Кудиц-Образ, 2004. — 432 с. — ISBN: 5-9579-0019-2, 0-321-16609-4.

44. Martin Fowler. The New Methodology, 2003. — [http://www.martinfowler. com/articles/newMethodology.html].

45. Alister Coorbern Characterizing People as Non-Linear, First-Order Components in Software Development// 4th International Multi-Conference on Systems, Cybernetics and Informatics, Orlando, Florida, June, 2000. — [http://alistair.cockburn.us/crystal/articles/cpanfocisd/characterizingpeopleasnon linear.html].

46. Зелковец М. Принципы разработки программного обеспечения / М. Зелковец, А. Шоу, Дж. Геннон. — М.: Мир, 1982.

47. Моисеев, Н.Н. Системный анализ / Н.Н. Моисеев. — М.: Наука, 1981.

48. Перегудов, Ф.И. Введение в системный анализ / Ф.И. Перегудов, Ф.П. Тарасенко. — М.: Высшая школа, 1989. — 367 с.

49. Кручинин В.В. Разработка компьютерных учебных программ. — Томск: Изд-во Томск ун-та, 1998. — 211 с.

50. Керниган Б., Ритчи Д. Фьюер А. Язык программирования Си. Задачи по языку Си. — М.: Финансы и статистика, 1985.

51. Берри Р., Микинз Б. Язык Си: введение для программистов. — М.: Финансы и Статистика, 1988. — 191с.

52. Ожегов С.И. Словарь русского языка. — М.: Русский язык, 1986. — 797с.

53. Антонов А.В. Восприятие внетекстовых форм информации в издании. — М.: Книга, 1972. — 104с.

54. Словарь издательских терминов / Под ред. А.Э. Мильгина. — М.:Книга, 1983. — 207с.

55. Семченко П.А. Основы шрифтовой графики. —Минск: Виш. школа, 1978. — 96с.

56. Калверт Ч. Программирование в Windows: Освой самостоятельно за 21 день. — М.: БИНОМ, 1995. — 496 с.