Министерство образования Республики Беларусь

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра экономической информатики

СОВРЕМЕННЫЕ ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ, ПРОГРАММА
И КОНТРОЛЬНЫЕ ЗАДАНИЯ
по курсу
“Современные информационные технологии”
для студентов экономических специальностей
заочной формы обучения

Минск 1998

УДК ББК 22.1

Современные информационные технологии: Метод. указания, программа и контр. задания по курсу “Современные информационные технологии” для студентов экон. специальностей заочного отделения / А.В. Бахирев, В.Н. Комличенко, А.Г. Синицын. – Мн.: БГУИР, 1998. – 96 с.

В методическом пособии представлены программа курса “Современные информационные технологии”, краткий справочник по материалам соответствующего лекционного курса, методические рекомендации по выполнению контрольных работ, пример программной реализации типового задания, входящего в состав контрольной работы, список используемой литературы и варианты контрольных работ. В справочнике по материалам курса рассматриваются базовые понятия объектно-ориентированных технологий разработки программных систем и основы объектно-ориентированного программирования, технологии современных баз данных и компьютерных сетей, технологии искусственного интеллекта и обеспечения информационной безопасности.

Программа разработана на кафедре экономической информатики.
Обсуждена и одобрена на заседании кафедры экономической информатики
протокол № 10

Рекомендована Советом экономического факультета
“ 1 ” июля 1998 г. , протокол № 10

1. Программа курса

1.1. Значение и задачи курса

Курс “Современные информационные технологии” предназначен для ознакомления студентов специальности Э.01.03.00 с современными технологиями, системами и средствами обработки информации, а также методами объектно-ориентированного проектирования и программной реализации систем обработки данных. Основной задачей курса является приобретение учащимися навыков использования современных технологий для решения профессиональных задач, овладение методами объектно-ориентированного анализа и проектирования систем обработки данных, получение практических навыков работы с инструментальными средствами поддержки технологий и разработки объектно-ориентированных программ на C++.

1.2. Объем и структура курса

В учебном плане специальности на курс отводится 51 часов лекций и 34 часов лабораторных занятий. Вся нагрузка выполняется в V семестре. В табл.1 приводится распределение объема курса по темам и видам занятий. Материал лекций рассчитан на знание основ объектно-ориентированного программирования

Таблица 1

№	Наименование тем и разделов	Дневное обучение		Заочное обучение	
		Лек-ций	Лаб. раб.	Лек-ций	Лаб раб.
1	2	3	4	5	6
1	*Предмет и содержание дисциплины*	*2*		*2*	
2	*Технологии, методы и средства разработки информационных систем*	*27*	*24*	*2*	*4*
2.1	Объектно-ориентированный анализ (ООА)	2			
2.1.1	Анализ и классификация информационных объектов	2	2		
2.1.2	Информационное моделирование	2	2		
2.1.3	Жизненные циклы и модели поведения	4	2		

Окончание табл. 1

2.1.4	Модели процессов	2	2		
2.1.5	Моделирование больших систем	2	4		
2.2.	Объектно-ориентированное проектирование и программирование	2	4		
2.2.1	Сложные системы и становление объектного подхода	3			
2.2.2	Классы и объекты	2	2		
2.2.3	Методология и процессы	2	2		
2.2.4	Основные методы проектирования и программная реализация проектов	4	4		
3	*Сетевые информационные технологии*	*6*		*0.5*	
4	*Технологии поддержки и проектирование баз данных (БД)*	*8*	*10*	*1*	*2*
4.1	Методы, технологии и инструментальные средства проектирования БД	4	6		
4.1	Распределенная обработка данных. Архитектура клиент-сервер	4	4		
5	*Технологии и системы искусственного интеллекта. Системы поддержки принятия решений*	*4*	*2*	*0.5*	
6	*Технологии обеспечения безопасности информационных систем*	*2*	*2*		
7	*Перспективные направления развития информационных технологий*	*2*			

1.3. Содержание учебных занятий

1. *Предмет и содержание дисциплины (2 ч)*

Современные информационные технологии и системы. Техническое и программное обеспечение информационных технологий.

2. *Технологии, методы и средства разработки программных систем (27 ч)*

2.1. Объектно-ориентированный анализ (ООА) (2 ч)

Введение в объектно-ориентированный анализ. Основные методы, модели и процессы.

2.2. Анализ и классификация информационных объектов (2 ч)

Идентификация объектов. Понятие атрибута и табличная интерпретация данных. Модели представления. Отношения и связи. Виды связей и их формализация.

2.3. Информационное моделирование (2 ч)

Представление связей. Безусловная и условная связь. Формализация и композиция связей. Понятие подтипа и супертипа. Представление информационных моделей.

2.4. Жизненные циклы и модели поведения (4 ч)

Жизненные циклы, состояния, события и действия. Формирование жизненных циклов для супертипов и подтипов. Модели состояний. Динамика связей и систем.

2.5. Модели процессов (2 ч)

Потоки данных и их представление. Идентификация процессов и обмен данными между процессами. Порядок выполнения и многократное использование процессов.

2.6. Моделирование больших систем (2 ч)

Понятие домена и их типы. Связи между доменами, понятие моста. Подсистема. Связи между подсистемами. Разработка моделей.

2.7. Объектно-ориентированное проектирование и программирование (2 ч)

Структурное, модульное и объектно-ориентированное проектирование и программирование. Методы, системы и средства программной реализации разрабатываемых проектов.

2.8. Сложные системы и становление объектного подхода (3 ч)

Проектирование систем и сложность программного обеспечения. Становление объектного подхода, его компоненты и применение.

2.9. Классы и объекты (2 ч)

Сущность понятий классов и объектов. Идентификация классов и объектов, ключевые абстракции и механизмы.

2.10. Методология и процессы (2 ч)

Диаграммы классов, объектов, перехода состояний и процессов. Проектирование и реализация классов и объектов.

2.11. Основные методы проектирования и программная реализация проектов (4 ч)

Объектно-ориентированное проектирование и реализация программных систем. Управление проектом. Преимущества и недостатки объектно-ориентированного проектирования.

3. *Сетевые информационные технологии (6 ч)*

Общие сведения о компьютерных сетях. Сетевые модели и протоколы. Средства объединения компьютерных сетей. Электронная почта и Internet.

4. Технологии поддержки и проектирование баз данных (БД) (8 ч)

4.1. Методы, технологии и инструментальные средства проектирования БД (4 ч)

Файловые системы, базы данных, системы управления базами данных. Проектирование БД, основные этапы и методы представления данных. Проектирование БД на основе принципов нормализации отношений и проектирование в терминах сущность-связь. Стандарт IDEFIX. CASE-системы.

4.2. Распределенная обработка данных. Архитектура клиент-сервер (4 ч)

Реляционный подход и средства манипулирования данными. Понятие транзакции и управление в системах баз данных. Распределенная обработка и архитектура клиент-сервер. Управление транзакциями и синхронизация данных.

5. Технологии и системы искусственного интеллекта. Системы поддержки принятия решений (4 ч)

Искусственный интеллект и основные направления исследований. Методы представления знаний и экспертные системы (ЭС). Инженерия знаний. Основные механизмы и работа ЭС. Системы поддержки принятия решений.

6. Технологии обеспечения безопасности информационных систем (2 ч)

Понятие информационной безопасности. Цель защиты и критерии оценки. Программно-технические средства обеспечения безопасности.

7. Перспективные направления в развитии информационных технологий (2 ч)

Анализ развивающихся технологий. Выделение перспективных направлений и прогнозы развития по материалам научно-технических изданий, конференций и периодической печати.

1.4. Перечень лабораторных работ

По курсу запланировано 8 четырехчасовых лабораторных работ. В табл.2 приведен перечень работ с указанием цели каждой работы. Техническим обеспечением лабораторного практикума является класс персональных ЭВМ с объемом оперативной памяти от 16 Мб и сервером с архивированными лицензионными копиями программного обеспечения.

Два часа учебного времени отводится на защиту лабораторных работ. Методическое обеспечение практикума поддерживается соответствующей разработкой указаний к лабораторным работам.

Таблица 2

№	Название работы	Цель работы	Перечень необходимых программных средств
1	2	3	4
1	Информационное моделирование	Анализ и классификация информационных объектов. Описание предметной области при помощи иерархии классов. Реализация иерархии в терминах и конструкциях C++	IDE Borland C++ версия 2.0 или выше
2	Методы классов и механизм наследования при моделировании поведения программных объектов	Освоить проектирование и программную реализацию методов, моделирующих поведение объектов, механизмов, наследование и модификация атрибутов и свойств	IDE Borland C++ версия 3.0 или выше
3	Полиморфизм и его использование при разработке программ и систем	Изучить методы полиморфизма в ООП. Освоить их использование при создании иерархии классов и программную реализацию на C++	То же

Окончание табл. 2

1	2	3	4

4	Моделирование событий	События, их распространение, обработка и использование. Механизмы исключения. Программная реализация процессов	IDE Borland C++ версия 3.0 или выше
5	Параметрический полиморфизм	Параметризация функций и классов. Повторное использова-ние параметризованных классов	То же
6	Разработка программных систем	Разработка и программная реализация систем для решения реальных задач	То же
7.	Проектирование баз данных	Проектирование БД с использованием CASE средств. Реализация БД в Sybase SQL Anywhere	Sybase SQL Anywhere 5.0
8	Распределенная обработка данных	Реализация БД в архитектуре клиент-сервер. Распределенная обработка данных. Репликацион-ные системы. Репликации в Sybase SQL Anywhere	То же

Для заочного отделения количество часов лекционного времени, отводимых для каждой темы, указано в табл.1. Нагрузка по лабораторным занятиям планируется в объеме 6 часов (см. табл.1).

Распределение нагрузки:

на установочной сессии прорабатываются общие теоретические вопросы об изучаемом предмете, состав и содержание основных тем курса. Объем - 2 часа лекционных занятий;

на зачетной сессии – использование компьютерных технологий для реализации прикладных задач, базовые вопросы проектирования и реализации программных систем, сетевые технологии, распределенная обработка данных и системы искусственного интеллекта, согласно табл.1. Объем лекционной нагрузки – 4 часа.

2. Справочник по содержанию курса

2.1. Современные информационные технологии

В современном понимании **информационную технологию** можно определить как совокупность методов, способов, приемов и средств обработки документированной информации, включая прикладные программные средства, и регламентированного порядка их применения.

Экономическая информация подвергается, как правило, всем процедурам технологического процесса, хотя в ряде случаев некоторые из них могут отсутствовать. Последовательность процедур также может быть различной, отдельные звенья могут повторяться. Состав и особенности выполнения процедур во многом зависят от экономического объекта и процессов, протекающих в среде его обитания.

ИТ имеют различные **уровни представления**:

концептуальное представление. На этом уровне определяются среда обитания объекта, целевые задачи, базовые принципы и средства реализации ИТ. Определяется вид структурной организации управления: децентрализованное, централизованное или иерархическое;

описание информационных потоков. Определяются объемы, периодичность получения, необходимость накопления, пути перемещения, места обработки, хранения, накопления информации;

описание методов получения, обработки и распространения информации;

описание инструментальных средств (универсальных и специальных);

Объектно-ориентированная технология основывается на объектной модели, основными принципами которой являются: абстрагирование, инкапсуляция, модульность, иерархичность, типизация, параллелизм и сохраняемость. Объектно-ориентированный анализ и проектирование принципиально отличаются от традиционных подходов структурного проектирования. Отличие обусловлено тем, что объектно-ориентированное проектирование базируется на методологии объектно-ориентированного программирования.

2.2. Объектно-ориентированный анализ и проектирование[2,3]

Объектно-ориентированный анализ - метод, используемый для отождествления важных сущностей в задачах реального мира, понимания их взаимодействия. Он направлен на создание моделей, близких к реальности, с использованием объектно-ориентированного подхода. На результатах ООА формируются модели, на которых основывается объектно-ориентированное проектирование, в свою очередь создающее базу для окончательной реализации

системы с использованием методологии объектно-ориентированного программирования.

При описании больших систем необходимо рассматривать ряд четко определенных предметных областей (доменов). Каждая из них должна рассматриваться как отдельный мир, населенный собственными концептуальными сущностями или объектами.

Объект в ООА - это такая абстракция множества предметов реального мира, которой свойственно следующее:

все предметы в этом множестве – экземпляры – имеют одни и те же характеристики;

все экземпляры подчинены и согласованы с одним и тем же набором правил и линий поведения.

ООА - метод для отождествления важных сущностей в задачах реального мира, для понимания и объяснения того, как они взаимодействуют между собой. Он описывается в три этапа:

1. **Информационные модели** – абстрагирование реальных сущностей в терминах объектов и атрибутов. Отношения между сущностями формализуются в связях, которые основываются на линиях поведения, правилах и физических законах, превалирующих в реальном мире.

2. **Модели состояний** описывают поведение объектов и связей во времени. В ООА каждые объект и связь имеют жизненный цикл – регулярную составляющую часть динамического поведения.

3. Все процессы, совершающиеся в системе, устанавливаются в действиях. **Модели процессов** описывают алгоритмическую или функциональную природу действий. Цель моделирования заключается в расчленении каждого действия на фундаментальные процессы, которые, вместе взятые, определяют требуемое функциональное содержание системы. При описании больших систем приходится рассматривать ряд предметных областей, которые должны четко определять множество сущностей, их представляющих.

Классификация – средство упорядочения знаний. В ООА определение общих свойств объектов помогает найти общие ключевые абстракции и механизмы.

Методы классификации:

классическое распределение по категориям (категорию формируют сущности, обладающие свойством или совокупностью свойств);

концептуальное объединение (формируются концептуальные описания классов, а затем классифицируются сущности в соответствии с этим описанием);

теория прототипов (класс определяется одним объектом-прототипом, и новый объект можно отнести к классу, при условии, что он наделен сходством с прототипом).

Взаимодействие классов и объектов как в ООА, так и в объектно-ориентированном проектировании рассматривается обычно в двух измерениях: логическом (физическом) и статическом (динамическом). Оба этих аспекта необходимы для определения структуры и поведения объектной системы.

Под **проектированием** обычно понимают процесс создания описаний нового или модернизируемого объекта (изделия, процесса), достаточных для изготовления или реализации этого объекта в заданных условиях.

В классической теории **проектирование** определяют как процесс, заключающийся в преобразовании исходного описания в окончательное на основе выполнения комплекса работ исследовательского, расчетного и конструкторского характера. Проектирование сложных объектов базируется на следующих основных **принципах**:

декомпозиция и иерархичность описаний объектов;

многоэтапность и итерационность проектирования;

типизация и унификация проектных решений и средств проектирования.

Объектно-ориентированное проектирование – методология проектирования, соединяющая в себе процесс объектной **декомпозиции** и приемы представления как **логической** и **физической**, так **статической** и **динамической** модели проектируемой системы.

Логическое представление описывает перечень и смысл ключевых абстракций и механизмов, определяющих предметную область и архитектуру системы, **физическое** – конкретную программно-аппаратную платформу реализации системы.

Отдельные проекции (ракурсы) моделей отображаются при помощи **диаграмм**. В проектировании используются следующие диаграммы:

диаграмма классов;

диаграмма объектов;

диаграмма модулей;

диаграмма процессов.

Первые две диаграммы – часть логического представления системы, потому что они служат для описания ключевых абстракций проекта. Последние две – часть физической структуры системы, потому что они описывают конкретные программные и аппаратные компоненты реализации проекта.

Все четыре диаграммы являются статическими. Для описания динамической модели используются дополнительные диаграммы:

диаграммы переходных состояний;

временные диаграммы.

Объектно-ориентированное программирование – методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Процесс ООП – поступательный и итеративный:

идентификация классов и объектов данного уровня абстракции;

идентификация семантики классов и объектов;

идентификация связей между классами и объектами;

использование классов и объектов.

Жизненный цикл разработки программных систем:

анализ $\rightarrow$ проектирование $\rightarrow$ программирование $\rightarrow$ тестирование $\rightarrow$ эксплуатация.

2.3. Объектно-ориентированное проектирование и программирование[3-6]

Модульность, **структурный стиль** и **нисходящее проектирование**, определяемые как основные принципы **структурного программирования**, обеспечили колоссальный толчок в развитии теории и практики разработки программных систем, позволили сократить сроки их создания, упростить отладку и сопровождение. Но оказалось, что структурное программирование не в силах обеспечивать разработку тех громадных программ в несколько десятков и более тысяч операторов языка высокого уровня, которые необходимы для создания серьезных программных комплексов, разрабатываемых большими группами исполнителей. Такую задачу призваны разрешить технологии **объектно-ориентированного проектирования** и разработки программных систем. И знание основ данной технологии необходимо всем, кто активно использует компьютер в своей профессиональной деятельности.

ОБЪЕКТ

Все, что окружает нас в реальном мире, может быть **объектом**: автомобиль, документ, компьютер, человек, интегральная схема. Объекты реального мира обладают свойствами, как, например, цвет или размер. Они обнаруживают поведение, начинают функционировать или менять состояние в ответ на определенный набор внешних воздействий. В силу определенной стабильности их можно использовать многократно и не нужно создавать каждый раз заново. Мы можем классифицировать объекты объединяя некоторые типовые из них можно создавать более сложные. Их существование позволяет разработчику сосредоточится на решении поставленной задачи, вместо того чтобы заново их изобретать. Было бы неплохо, если бы такими же замечательными свойствами обладали и программные объекты. Реализацию таких свойств в программных объектах обеспечивают три фундаментальные метода ООП: **инкапсуляция** (encapsulation), **наследование** (inheritance), **полиморфизм** (polymorphism).

ИНКАПСУЛЯЦИЯ

Свойство инкапсуляции, присущее реальным объектам, позволяет нам видеть и различать объекты, фиксировать их реакции в ответ на внешние воздействия, наблюдать развитие процессов.

Программные объекты моделируются на основе аналогий с существованием и взаимодействием объектов реального мира. Их наделяют нужными свойствами и поведением. Основой для этого служит **метод инкапсуляции**.

Понятие **инкапсуляция** означает, что в качестве единого целого, называемого объектом, рассматривается некоторая структура данных, определяющая его свойства, или атрибуты, и некоторая группа методов (функций) для манипулирования этими данными, задающая поведение объекта.

В реализации C++ данные хранятся в структурах, напоминающих обычные структуры языка C, а поведение объектов реализуется при помощи функций (member function) или методов, связанных с этими данными специальным описанием, задающим область действия этих функций. Кроме того, программисту предоставляются широкие возможности по управлению доступом к данным и методам объекта, что повышает надежность и модифицируемость программ. Ограничения на доступ к информации объекта аналогичны ограничениям реального мира, где, как правило, нет необходимости знать во всех подробностях внутреннее

устройство какого-либо объекта, скажем, телефонного аппарата, заключенного в капсулу (корпус), а достаточно лишь знать его функциональные свойства (т.е. его реакцию на допустимые способы внешнего воздействия).

НАСЛЕДОВАНИЕ

В реальном мире механизм **наследования** позволяет закрепить и передать основные черты, присущие объектам, от одного поколения к другому. Хорошей аналогией наследования может служить таксономическая схема, которой пользуются зоологи и ботаники для классификации живых организмов. В данной схеме группы более низкого уровня наследуют характеристики более высоких уровней.

Программные объекты выстраиваются в иерархию таким же образом. Механизм **наследования** помогает сделать разработку более экономной и читаемой, т.к. объекты пользуются одними и теми же атрибутами и формами поведения без дублирования реализующих их программных кодов.

Примером может служить некоторая абстракция окна на экране, обладающего определенными данными (высота, ширина, рамка, цвет и т. п.) и поведением, как, например, способность перемещаться, менять цвет фона и т.д. Мысленно перемещаясь вниз по иерархии, можно представить объект меню, наследующий все характеристики окна и, кроме того, обладающий рядом собственных (частных) характеристик-данных (например, фрейм структуры меню и его собственные строки-элементы) и форм поведения, как-то: возможность прорисовки меню, перемещения курсора выбора, запуска некоторых процессов и т.п. Продолжая опускаться вниз по иерархии, можно рассмотреть окно текстового (либо какого-либо другого) редактора, как наследующего характеристики окна и меню, так и располагающего рядом собственных параметров и методов.

Здесь необходимо подчеркнуть, что и меню и редактор можно назвать окнами, так как они имеют все атрибуты окна (положение, высоту, ширину и т.д.). Кроме того, все названные объекты имеют и некоторые общие черты поведения, например способ перемещения на экране дисплея, но при этом они различаются по виду и другим способам функционирования. Такая иерархия может иметь много уровней, ее можно расширить не только вниз, но и вверх, начав, например, не с окна, а с абстракции точки на экране.

По терминологии ООП языка C++ ОКНО, МЕНЮ, РЕДАКТОР и т.д. – это **классы**. Классы образуют (описывают) иерархию, которая и определяет

наследование свойств между ними. Так, класс МЕНЮ является производным классом и наследует свойства от класса ОКНО, который является для него базовым. В свою очередь класс ОКНО РЕДАКТОРА является производным от базового класса МЕНЮ и т. д. Следует заметить, что сами по себе классы - это не объекты, а шаблоны для создания объектов (или новый тип данных). При необходимости создается экземпляр класса (объект), который используется в программе. Отношение между классом и экземпляром класса такое же, как между типами данных и переменной.

Классы, как уже отмечалось, наследуют не только атрибуты-данные, но и поведение. Например, если мы предусмотрели функцию перемещения окна по экрану, то меню и редактор унаследуют эту функцию. Это значит, что любое меню и окно любого редактора можно перемещать по экрану, не программируя заново функцию перемещения, а воспользовавшись ее кодом, разработанным для базового класса. Это свойство, называемое многократностью использования кода, не только позволяет избежать ненужного дублирования программных кодов, но и гарантирует, что если функция запрограммирована корректно, то она будет правильно работать со всеми объектами, входящими в иерархию. Кроме того, все такие объекты будут двигаться по экрану одинаково, а это большое преимущество для конечного пользователя.

В некоторых объектно-ориентированных системах реализуется только одиночное наследование. В Borland C++ иерархия расширяется множественным наследованием. То есть меню может наследовать свойства и поведение не только от окна, но одновременно и от других классов. Возможность множественного наследования позволяет непосредственно комбинировать характеристики двух или нескольких различных классов, что уменьшает объем необходимых модификаций программ.

ПОЛИМОРФИЗМ И ПОЗДНЕЕ СВЯЗЫВАНИЕ

Благодаря **полиморфизму** все в реальном мире развивается и совершенствуется. Явление полиморфизма позволяет объектам приспосабливаться к меняющимся внешним условиям, приобретать определенную устойчивость и мобильность. Трудно переоценить роль этого фундаментального свойства и как метода в теории и практике ООП. Он основывается на понятиях раннего и позднего связывания.

Что же касается терминов "ранее связывание" и "позднее связывание", то они относятся к этапу, на котором обращение (вызов) к функции, к процедуре связывается с ее реальным адресом.

Для раннего связывания адреса всех функций и процедур должны быть известны в тот момент, когда происходят компиляция и компоновка программы. Это позволяет приписать каждому обращению к процедуре соответствующий адрес. В большинстве традиционных языков программирования, включая Си и Паскаль, используется только раннее связывание. В случае позднего связывания адрес процедуры не связывается с обращением к ней до того момента, пока обращение не произойдет фактически, т.е. в процессе выполнения программы.

Вернемся к нашему примеру с окнами. В такой сложной динамической системе нельзя заранее предсказать, сколько окон будет на экране, каких они будут типов и в какой последовательности будет осуществляться работа с ними.

В программе, где используется только раннее связывание, вся информация о количестве, координатах и типах окон должна хранится в основной программе, как и все возможные действия над окнами. При каждом изменении программа должна разобраться, что именно и с каким окном произошло, и вызвать соответствующие процедуры для выполнения надлежащих действий с данным окном. Такая программа становиться очень сложной, теряет гибкость. Стоит добавить новый тип окна или изменить поведение одного из них, и придется вносить коррективы во многие места такой программы.

Коренным образом меняется ситуация в случае позднего связывания. Рассмотрим случай, когда одно из окон перекрывает другое. Если "верхнее" окно будет передвинуто или закрыто, то нижнее следует перерисовать для восстановления ранее перекрытой части. Так как меню перерисовывается иначе, чем, например, окно редактора, то каждый объект в оконной иерархии должен знать, как перерисовать себя. Таким образом, в каждом классе будет своя функция-метод, связанная с объектом, которая знает, как перерисовать данный объект на экране. Механизм полиморфизма позволяет называть этот метод одним и тем же именем в различных классах объектов. Кроме того, он позволяет полиморфно вызывать метод, реализуя его контекстную привязку к объекту. Следовательно, нет необходимости анализировать, к какому типу окна он относится, как это требуется при раннем связывании. Происшедшее событие просто инициирует команду "Перерисовать", и объекты корректно перерисовывают себя на экране. Таким образом, если пакет

содержит семь типов окон, то в нем будет семь различных правил перерисовки, но все они будут называться одинаково: "Перерисовка".

Это множественность форм, которые может принимать правило с одним и тем же именем, называется **полиморфизмом**, от греческого polymorphos - многообразный.

Полиморфизм – исключительно мощный метод обобщения однотипных задач для многих разных объектов. Он повышает степень абстрагирования при создании программного обеспечения, т.к. программист заботится только о том, чтобы указать правильное действие, а не о том, как его выполнить. Это возможно благодаря позднему связыванию.

Полиморфизм является одним из фундаментальных свойств ОПП. Суть его заключается в том, что одни и те же имена (например, объявление переменных или методов) могут соответствовать различным классам объектов.

Поэтому объект, объявленный с таким именем, будет по-разному реагировать на некоторое множество допустимых событий, в зависимости от класса объекта, который это имя представляет в текущий момент.

РАСШИРЯЕМОСТЬ КОДА

Итак, сочетание наследования и полиморфизма дает пользователю объектно-ориентированной программы возможность расширить эту программу, не имея ее исходного кода. Это возможно потому, что наследование действует и после компиляции исходной программы. Т.е. если программист в C++ располагает описанием интерфейса с некоторым классом, он может определить производный класс, наследующий все данные и поведение базового, а затем выборочно перегрузить (переопределить) некоторые свойства, вводя характеристики, свойственные данному классу. Это возможно даже в том случае, если базовый класс компилировался раньше, чем производный класс был написан. Такую возможность дает позднее связывание, которое происходит во время выполнения программы, т.к. изнутри объектного модуля (базовой иерархии классов) вполне могут полиморфно вызываться функции, которых еще не существовало во время его компиляции.

Таким образом, если программный пакет создается с использованием разработанной (например, фирменной) библиотеки объектов (имеется библиотека объектных модулей), это не означает, что описания объектов должны браться только из библиотеки. Если характеристики одного или нескольких объектов библиотеки не

устраивают программиста, объекты могут быть унаследованы и модифицированы в проектируемых собственных классах.

КЛАСС И ОБЪЕКТЫ КЛАССА

Класс в ООП подразумевает собой некоторое описание, определяющее множество объектов, связанных общностью структуры и поведения. В большинстве случаев класс можно понимать как новый тип данных (за исключением понятий наследования и иерархичности).

C++ класс, определяемый посредством ключевых слов struct, union или class, включает в себя функции (называемые методами) и данные (называемые элементами данных), создавая новый тип объектов.

Компоненты класса имеют ограничения на доступ. Эти ограничения определяются ключевыми словами "**private**:", "**protected**:", "**public**:".

Для ключевого слова class "по умолчанию" все компоненты будут private. Это означает, что они (их имена) будут недоступны для использования вне компонентов класса. Ограничение доступа "по умолчанию" для некоторого компонента можно изменить, записав перед ним атрибут модификации доступа - ключевое слово public или protected и двоеточие. Таким образом, упрощенную форму описания класса можно записать в виде:

```
Class имя класса {
        данные и функции с атрибутом private (по умолчанию)
protected:
        данные и функции с атрибутом protected
public:
        данные и функции с атрибутом public
} объекты этого класса через запятую;
```

Пусть необходимо создать класс объектов точка (point) на экране дисплея. Точка на экране определяется координатами X и У, имеет цвет, например С, и функцию (метод), отображающую ее на экране. Тогда, согласно определению, создадим формальное описание класса Point:

```
class Point {
        int X,Y,C;
public:
        void show (void); //void - пустое значение, используемое в качестве
```

```
//типа возвращаемого значения и в качестве списка передаваемых
//параметров
{ putpixel(X,Y,C); }}; //putpixel - стандартная функция отображения
//точки на экране с координатами X,У и цветом, заданным в С.
```

Список объектов может быть и пустым, как в данном случае. Тогда описание класса представляет собой шаблон, который можно будет использовать в будущем для создания конкретных экземпляров класса - объектов.

Основное отличие между классами в С++ и структурами в Си заключается в методе инкапсуляции, ограничивающем права доступа к элементам класса. В С++ доступ к элементам структуры класса (class) может определяться описанием отдельных элементов (данных или функций) с атрибутом доступа public, private или protected, в то время как элементы структуры языка С доступны для любого выражения или функции в их области действия.

Обычно ограничения на уровень доступа в С++ касаются элементов данных: данные имеют атрибут private или protected, а методы - public.

Смысл атрибутов доступа следующий:

private - член класса с атрибутом private может использоваться только методами собственного класса и функциями-"друзьями" этого же класса; по умолчанию все члены класса, объявленного с ключевым словом class, имеют атрибут доступа private;

protected - то же что и private, но дополнительно член класса может использоваться методами и функциями - “друзьями" производного класса, для которого данный класс является базовым;

public - член класса может использоваться любой функцией программы, т.е. защита на доступ снимается.

Явно ограничения на доступ могут переопределяться записью атрибута доступа перед компонентами класса.

В С++ элементы класса типа структуры (struct) и объединения (union), в отличие от типа class (по умолчанию - private), по умолчанию принимаются как public. Для ключевого слова struct атрибут можно явно переопределять на private или protected. Для ключевого слова union явное переопределение атрибута доступа невозможно.

ПРИМЕРЫ ОБЪЯВЛЕНИЯ КЛАССОВ

1) для ключевого слова class:

```
class X {int i; // i - переменная целого типа в контексте класса X имеет
                //атрибут доступа private по умолчанию
public:
    int j;          //j и k имеют явно заданный
    float k;        //атрибут доступа public
protected: int l; char m;};
```

, здесь l и m имеют явно заданный атрибут доступа protected,

int – объявление переменной целого типа;

float - объявление вещественной переменной (значение представлено числом с плавающей точкой);

char - объявление символьной переменной.

2) для ключевого слова struct:

```
struct Y {    int i;// переменная целого типа i имеет атрибут доступа, по
                    //умолчанию, public
private:      int j;        // j и k имеют явно заданный
float k;                    // атрибут доступа private
protected:
int l;                      // l и m имеют явно заданный
char m; };                  //атрибут доступа protected
```

3) для ключевого слова union:

```
union Z { int i; };     //i имеет, по умолчанию, атрибут доступа
                        // public и не может быть переопределено
```

Из приведенных примеров видно, что в C++ можно использовать две наклонные черты для введения однострочного комментария. Допускается использование пары символов (/* */), но их лучше применять для многострочных комментариев.

Помимо данных класс включает и функции-члены класса (методы). Методы класса связываются с конкретным классом специальным оператором :: (два рядом расположенных двоеточия). Знак :: называется областью действия оператора (контекст). Он говорит компилятору о том, что данная версия функции принадлежит заданному классу.

Пример

X::f1(int x); - функция f1 принадлежит классу X; (говорят, функция f1 задана в контексте класса X)

Так, пример моделирования физической точки (пикселя) на экране, рассмотренный выше, мог бы быть записан следующим образом:

```
class Point { int X,Y,C;
public:void show (void); } p1; // void - пустое значение, используемое в
//качестве типа возвращаемого значения и в качестве списка передаваемых
//параметров
void Point::show(void){putpixel(X,Y,C); }
```

Знак :: определяет принадлежность функции show() контексту класса Point (Point::show()), хотя определение функции физически расположено вне описания класса.

В описании класса объявлен объект (переменная) p1 - типа Class Point, непосредственно при описании класса. Кроме того, после введения описания класс можно использовать как новый тип данных и объявлять объекты следующим образом:

```
Point P2,Center; //указывать ключевое слово class уже нет необходимости
```

Отметим, что описание класса Point задает только формальный шаблон для множества объектов данного класса. Объявленные переменные p1, P2, Center - объекты типа Point, с элементами которых осуществляется работа в программе. Так можно записать P2=P1, что будет обозначать присвоение координат из P1 (x и y) переменным объекта P2. В то же время запись Point=P2; - бессмысленна. Переменные X и У определяют атрибуты (свойства) класса (то, что класс имеет), а методы (поведение) - то, что класс делает.

Добавим к описанию класса Point два простых метода GetX и GetY.

Запишем:

```
class Point { int X,Y,C;
public: void show (void);     // void - пустое значение, используемое здесь
// в качестве типа /возвращаемого значения и в качестве списка
// передаваемых параметров
int GetX() {return X;};          //определение метода внутри класса
int GetY();};                    //объявление метода внутри класса
void Point::show(void)
{ putpixel(X,Y,C); }             //определение методов вне
int Point ::GetY( ) { return Y; }       //описания классов
```

Метод GetX( ) в примере определен как встроенная функция. Такой способ определения метода – внутри описания класса – используется в случае "небольших"

функций (на практике – до 40 операторов). Компилятор при этом способе выполняет не стандартный вызов функции, а реализует макрорасширение команды вызова кодом функции, т.е. встраивает код вызываемой функции в точку вызова вместо обычного перехода по адресу размещения функции с последующим возвратом в точку вызова. Это снижает накладные расходы, связанные с обращением к функции и выходом из нее. На встраиваемые функции налагается ряд ограничений. Если ограничения не позволяют компилятору выполнить макрорасширение вызовов встроенной функцией, это не рассматривается как ошибка. Выдается соответствующее предупреждение, а функция реализуется по типу метода, определенного вне описания (но в контексте) класса, и ее вызов оформляется стандартным образом (по аналогии с методом void show(void)).

Методы GetY() и show() объявляются внутри класса и определяются в произвольном месте программы вне тела описания класса. Использование операции :: в определении методов (например, Point::GetY) указывает, что GetY принадлежит классу Point, т.к. в принципе могут существовать и другие версии GetY, принадлежащие другим классам. Кроме того, операция определяет и то, что Y в "return Y" является элементом Y из класса Point. Отметим, что внутреннее определение GetX не требует модификатора "Point::".

ВЫЗОВ КОМПОНЕНТОВ КЛАССА

Каким бы из рассмотренных способов ни объявлялся метод, важно то, что он функционирует внутри области действия класса Point независимо от его физического расположения. Если определить объект Point P1, то к доступным компонентам (объявленным как public:) можно обращаться так же, как к компонентам структур или смесей в C (связывание операцией "." - точка).

P1.GetX(); P1.show();

Обобщенная форма вызова:

имя-объекта-класса.имя-метода(описание аргументов);

Переменные X и Y в классе Point являются частными (private). Это значит, что обращение к ним вида P1.X или P1.У приведет к возникновению ошибки, так как к переменным private имеется доступ только из функции, принадлежащей данному классу, или в функции, объявленной другом (friend) данного класса. Так, внутри

метода GetY() имеется обращение к частному компоненту класса переменной Y. Префикс точка здесь не используется. Если же функция обращается к компоненту другого класса, уточнение является обязательным.

Отметим, что аналогично языку С, можно объявлять и использовать указатель на тип класс (признак указателя знак "*" в объявлении переменной).

```
Point * ptr2;   //объявление указателя ptr2 на тип Point;
ptr2=&P1;      // присваивание указателю ptr2 адреса расположения в
//памяти объекта P1, объявленного ранее
```

Теперь можно применять операцию выбора метода через операции "." или "->".

```
Ptr->GetX();  //вызов метода GetX() для объекта P1, соответствует
// вызову P1.GetX();
```

КОНСТРУКТОРЫ И ДЕСТРУКТОРЫ

В предыдущем примере мы использовали значения частных переменных X,Y,C для рисования точки при помощи функции putpixel(), не задаваясь вопросом, каким образом будет осуществляться инициализация (присваивание значений) названных переменных. Один из возможных способов передать необходимые значения частным переменным – через какой-либо из методов класса, объявленный как public. Это может быть, например, метод show(), объявленный в классе следующим образом:

```
void show(int K,int L,int M);
```

Вне класса метод мог бы иметь следующий код:

```
void Point::show(int K,int L,int M)
{ X=K; Y=L; C=M;    putpixel(X,Y,C); }
```

Такая инициализация представляется достаточно неудобной, так как оставляет пользователю груз контроля за состоянием переменных всякий раз, когда предстоит вызов какого-либо метода для объекта.

В С++ имеется два специальных типа методов – конструкторы и деструкторы, определяющие, каким образом объекты класса создаются, инициализируются, копируются и разрушаются. Конструкторы и деструкторы обладают большинством характеристик обычных методов, но имеют и ряд особенностей.

Рассмотрим функцию, называемую конструктором.

Конструктор имеет следующие отличительные особенности:

всегда выполняется при создании нового объекта класса, т.е. когда под объект отводится память и когда он инициализируется;

может определяться пользователем, либо создаваться C++ по умолчанию;

не может быть вызван явно из пределов программы (не может быть вызван как обычный метод). Он вызывается явно компилятором при создании объекта и неявно при выполнении оператора new (данный оператор используется при динамическом распределении памяти под объект) для выделения памяти объекту или при копировании объекта;

имеет всегда то же имя, что и класс, в котором он определен;

никогда не должен возвращать значения (не может описываться с каким-либо типом возврата, даже void). Он всегда вызывается при объявлении экземпляра некоторого класса (т.е. при создании объекта);

не наследуется;

не может быть объявлен как const, virtual или static.

Противоположные действия, по отношению к действиям конструктора, выполняют функции-**деструкторы** (destructor), или разрушители, которые уничтожают объект. Например: конструктор может выделить память для объекта, а деструктор - освободить ее. Аналогично конструктору деструктор может вызываться явно или неявно. Неявный вызов деструктора связан с прекращением существования объекта из-за завершения области его определения. Например, объект локальный внутри функции (объявленный внутри функции), и функция возвращает управление (завершается). Явное уничтожение объекта выполняет оператор delete. Компилятор генерирует деструктор по умолчанию, если он не объявлен в программе. Деструктор имеет такое же имя, как и класс, но перед именем записывается знак тильда (~).

Кроме того, деструктор не может:

иметь аргументы или возвращать значение;

наследоваться;

объявляться как const или static, но может быть объявлен как virtual.

Рассмотрим пример простой программы, демонстрирующей работу конструктора и деструктора.

```
// РАБОТА КОНСТРУКТОРА И ДЕСТРУКТОРА
#include<iostream.h>
//подключение системного файла с описанием функций ввода-вывода потоком
```

```
#include<string.h>
//системный файл с описаниями функций работы со строками
class string { int i;    public:
        string (int j);          //конструктор (объявление)
        ~strig();                       //деструктор (объявление)
        void shov_i(void);    //ф-я член - класса (метод) объявление прототипа
};                      //конец объявления класса
string::string(int j)    //определение конструктора вне описания класса
{ i=j; cout<<"работает конструктор \n"; }
void String::Show_i(void)   //определение метода вне класса
{cout << "i="<<i<< “\n”;}     //тело метода
string::~string()               //определение деструктора
{cout <<"работает деструктор \n";} //вывод потоком (при помощи
// переопределенной операции <<)
void main (void) {
        string my_ob1 (25); //инициализация объекта my-ob1
        string my_ob2 (36); //инициализация объекта my-ob2
        my_ob1.show_i();    // вызов функции show_i() класса string,для my_ob1
        my_ob2.show_i();}   // вызов функции shon_i() класса string, для my_ob2
```

Результаты работы данной программы:

```
работает конструктор
работает конструктор
i=25
i=36
работает деструктор
работает деструктор
```

Деструкторы выполняются в обратной последовательности по отношению к конструкторам. Первым разрушается объект, созданный последним.

Помимо работы конструктора и деструктора данный пример демонстрирует и новые возможности ввода/вывода в C++. Включенный файл iostream.h описывает функции, обеспечивающие поддержку ввода/вывода потоком. Строка

```
cout<<"i="<<i<< “\n”;
```

приводит к выводу символьной строки i= значение_i и обработке символа “\n”– новая строка. Оператор ввода значения переменной для int i1; выглядит следующим образом: cin>>i1;

Отметим, что в C++ тип данных можно объявить в любом месте функции (не обязательно в ее начале).

Чтобы реализовать данный пример при помощи ключевого слова struct, достаточно изменить описание класса следующим образом:

```
struct string { private:        int i;
public:         string(int j);  //объявление конструктора
                ~string();      // объявление деструктора
void show_i (void);};           // объявление метода класса
```

Смеси (union) в C++ также близки к классам. Фактически смесь - это та же структура (struct), в которой все компоненты-данные помещаются в одно и то же место памяти.

Рассмотрим пример объявления смеси в C++

```
/* СМЕСЬ В C++ */
#include<iostream.h>
#include<string.h>
union my_union
{ char str[22];         //компоненты
struct str1_2           //смеси
{ char str1[9];         //имеют
char str2[12];          //атрибут public
} my_s;                         // my_s - объект типа str1_2
my_union(char*s);   //конструктор класса
void print(void);               //функция - метод класса
};my_union::my_union(char*s) //определение конструктора
{strcpy(str,s);}                //функция копирования строки s в строку str.
void my_union::print(void)
{ cout<<my_s.str2<<”\n”;
        //вывод элемента структуры str2 и перевод строки
my_s.str2[0]=0;                         //запись признака конца строки
cout<<my_s.str1<< “\n”; }   //вывод значения str1 объекта my_s
void main (void)
```

{my_union ob ("Факультет исторический”);

//создание и инициализация объекта ob

ob.print(); // вызов ф-ии print() для ob }

РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ:

1.Создается объект ob и для него вызывается конструктор, который копирует строку "Факультет исторический" в компонент класса str[22];

2. Вызывается компонент-функция ob.print. Выводится текст строки начиная с 6-го символа до конца строки текст "исторический". Затем вместо символа "и" записывается 0 - конец строки, и на новой строке выводится текст с начала строки str1 до признака конца строки (0, записанного нами в str2[0]), т.е. слово "Факультет".

НАСЛЕДОВАНИЕ И ЗАЩИТА

Обычно при разработке нетривиальных программ **классы образуют иерархическую структуру**, отображающую существующую иерархию объектов. Иерархия классов формируется созданием производных классов на основе базовых, обладающих наиболее общими данными и методами. В свою очередь производный класс может использоваться как базовый для других классов. Здесь класс, на основе которого строится другой класс, называется базовым, а построенный класс – производным.

Шаблон объявления класса можно представить следующим образом: ключ_класса имя_производного_класса: необязательный модификатор доступа имя_базового_класса {тело_производного_класса} объекты производного_класса (через запятую);

Пример:

class Point : public Location {... тело класса Point ...};

Класс Location является базовым и наследуется с атрибутом public. Класс Point - производный класс. Список объектов опущен (после закрывающей фигурной скобки стоит точка с запятой). Двоеточие (:) отделяет производный класс от базового. Атрибут класса (модификатор прав доступа) может задаваться ключевыми словами public и private. Атрибут может опускаться – в этом случае принимается атрибут по умолчанию (для ключевого слова class – private, для struct – public). Смесь (union) не может быть базовым или производным классом.

Атрибут (модификатор прав доступа) используется для изменения прав доступа к наследуемым элементам класса в соответствии с правилами, указанными в табл. 3.

Таблица 3

Ограничения на доступ в базовом классе	Модификатор наследования прав	Ограничения на доступ в производном классе
private	private	Нет доступа
protected	private	private
public	private	private
private	public	Нет доступа
protected	public	protected
public	public	public

Отметим, что в производных классах права на доступ к элементам базовых классов не могут быть расширены, а только более ограничены. Наследование может осуществляться с атрибутом доступа public или private по отношению к базовому классу.

Атрибут private (см. табл. 3), используемый по умолчанию в объявлениях производного класса с ключевым словом "class" или задаваемый явно, трансформирует элементы базового класса с атрибутами доступа public или protected в элементы private производного класса. Доступ же к элементам private базового класса из производного невозможен по определению private атрибутов.

Атрибут public, используемый явно или по умолчанию при объявлении производного класса с ключом structure, не изменяет ограничения на доступ для атрибутов, наследуемых из базового класса, т.е. доступными являются элементы, объявленные в базовом классе как protected или public. Доступ к private невозможен по определению private элементов класса.

Иллюстративный пример наследования прав доступа:

```
class A {int a1;
public: int a2;
void f1(void) };
class b: A { int b1;
public:
void f1 (void)
```

```
{ a1=1;
// ошибка, a1 - private - переменная класса A и доступна только для
//методов и дружественных функций собственного класса.
b1=0;  //доступ к переменной типа private из метода класса;
a2=1;  // a2 унаследована из класса A с атрибутом доступа
        //private и поэтому доступна в методе класса;
} };
void main(void)
{ A a_ob1;              // объявление объекта a_ob1 класса A и
b b_ob1;                // объекта b_ob1 класса b
b_ob1.a2+=1;                 //ошибка, т.к. a2 private
a_ob1.a2+=1;}                //допустимая операция (a_ob1.a2=a_ob1.a2+1)
```

При создании иерархии классов права доступа должны передаваться с осторожностью. Обычно в C++ принята практика защиты данных класса от методов, не принадлежащих данному классу и не являющихся другом данного класса.

Рассмотрим еще один пример, демонстрирующий наследование прав доступа к компонентам базовых классов:

```
#include <iostream.h>       //подключение файлов описания
#include <graphics.h>       //прототипов библиотечных функций
#include <conio.h>
class Location {     protected:
int x,y;          // эти компоненты будут доступны в производном классе
public:
Location(int InitX,int Init Y); // конструктор для класса Location
int getx() { cout << "x= " << x <<"\n"; return x; }
                          //метод возвращает и распечатывает значение x
int gety() { cout << "y= " << y <<"\n"; return y; }
                          //метод возвращает и распечатывает значение y
void show_mess() { cout << "Нажмите любую клавишу"; }
                  //метод выдает сообщение на экран
};                //конец описания класса Location
/* Класс Point является производным от базового класса Location */
/*************************************************************************/
class Point : public Location         //определение наследования
```

```
{ int color;                    //собственное тело класса Point
public:
Point(int Init X, int InitY, int InitC); // конструктор
void putpixel() { ::putpixel(x,y,color); }
        //метод вывода изображения точки
        // (использует библиотечную функцию putpixel
        //для вывода точки с координатами x, y и цветом - color
};                              //конец описания класса Point
Location::Location(int Init X, int InitY)      // конструктор класса Location
{x = Init X; y = InitY);
cout << "работает конструктор Location\n"; }
Point::Point(int Init X, int InitY, int InitC) :
Location(Init X, InitY) {color = InitC;
        // конструктор Point сначала вызывает
        //выполнение конструктора базового класса Location
cout << "работает конструктор Point\n"; }
void main(void)
{ int gd = DETECT, gn;
initgraph(&gd,&gn,""); // инициализация графического режима
Point my_point(300,200,4);          //определение объекта my_point
my_point.putpixel();                //рисование точки для объекта my_point
my_point.getx();                    //вызов метода базового класса getx
my_point.gety();                    //вызов метода базового класса gety
void show_mess();           //вызов метода выдачи сообщения
getch();                    // приостановка до нажатия любой клавиши
clossegraph();}             // закрытие графической системы
```

Приведенный пример иллюстрирует работу механизма наследования. В базовом классе Location объявлены переменные x и y с атрибутом доступа protected. Это значит, что они будут доступны для методов производного класса. С атрибутом public в Location объявлен конструктор и 3 метода getx() и gety(), show_mess().

Класс Point является производным от класса Location. Так как базовый класс имеет конструктор, то и все производные классы должны иметь конструктор. Обратим внимание на то, что конструктор класса Point сначала инициирует вызов конструктора класса Location и передает ему параметры, необходимые для создания экземпляра объекта базового класса, который (неявно) будет существовать в составе

создаваемого объекта класса Point. Значения переменных InitX и InitY, используемые для приема извне и передачи значений внутренним переменным класса, интерпретируются здесь как горизонтальная и вертикальная координаты точки, а InitC - цвет точки на экране.

В C++ допускается образование производного класса от нескольких базовых классов (будет рассмотрено дальше). Общий синтаксис конструктора производного класса представляется следующим образом:

имя_ конструктора_ производного_ класса (список_ аргументов_ для_ производного_ класса): имя_ базового_ класса_1 (список_ аргументов_ для_ базового_ класса_1,..., имя_ базового_ класса_N (список_ аргументов_ для_ базового_ класса_N) {тело_ конструктора_ производного_ класса;}

Конструкторы производных классов всегда сначала вызывают конструктор базового класса. Если базовый класс, в свою очередь, является производным, то процесс вызова конструкторов продолжается по всей иерархии. Если в некотором классе конструктор не определен, то инициализируется конструктор по умолчанию (без аргументов). Заметим, что ссылка на конструктор базового класса дана не в объявлении конструктора производного класса, а в его определении.

Рассмотрим головную программу.

Т.к. мы используем графический режим работы, то необходимо инициализировать графическую систему (предполагается работа под MS/DOS). Функции поддержки и определение данных для графической системы располагаются в заголовочном файле graphics.h.

Функция initgraph имеет 3 параметра. Это указатели на переменные: номер графического драйвера, номер режима для этого драйвера и путь к директорию, содержащему bgi файл данного драйвера. Если первой переменной было присвоено значение detect (или 0), то сначала запускается функция автоматического тестирования аппаратуры с целью определения типа дисплейного адаптера. По результатам тестирования функция initgraph переходит к загрузке соответствующего bgi файла. Третий параметр в функции initgraph(&gd,&gn,""); определяет путь доступа к директорию, где располагается bgi файл драйвера. Отсутствие информации в кавычках "" говорит о том, что файл располагается в текущем директории (там же, где соответствующий исполняемый модуль программы (exe-модуль)).

Дальше определяется объект my_point(300,200,4). Конструктор класса Point вызывает конструктор базового класса Location, передает ему параметры для инициализации переменных экземпляра объекта класса Location, входящего в состав создаваемого объекта my_point, а затем выполняется собственный код конструктора

Point, завершая построение объекта. Внешне эти процессы будут сопровождаться выводом текста:

работает конструктор Location

работает конструктор Point

Затем для объекта my_point вызывается метод putpixel(), который выводит на экран точку красного цвета с координатами 200,300.

Следует обратить внимание на то, что функция класса Point putpixel имеет то же имя, что и библиотечная функция. Поэтому, для исключения рекурсивных вызовов и ошибок в программе, записывается префикс ::(разрешение области действия) перед библиотечной функцией. Имя приобретает контекст глобального, что устраняет возможные неоднозначности при выполнении программы.

Далее работают методы getx и gety для объекта my_point.

Выводятся строки:

x=300

y=200

Метод show_mess выводит сообщение

Нажмите любую клавишу

Дальше следует функция останова getch() - до нажатия любой клавиши и функция closegraph() - закрывающая графическую систему.

Выше говорилось о полиморфизме, как об одном из фундаментальных свойств ООП и о механизмах раннего и позднего связывания. В C++ свойство полиморфизма распространяется на функции и операторы. При раннем связывании полиморфизм проявляется как перегрузка функций и операторов (операций) на этапе компиляции. При позднем связывании - в вызовах виртуальных функций. В этом случае связывание вызова с кодом конкретной функции осуществляется на этапе выполнения программы.

Преимущество раннего связывания: высокая скорость выполнения программы, малые затраты памяти. Недостаток - потеря гибкости при программировании задач.

Позднее связывание предполагает вызовы виртуальных функций. Преимущество позднего связывания - большая гибкость программирования при некотором снижении скорости выполнения.

ПЕРЕГРУЗКА ФУНКЦИЙ

В языке С, как и в большинстве других процедурных языков, функции должны иметь уникальные имена. В С++ имеется возможность использовать несколько функций с одним и тем же именем. Когда несколько функций определяются с одним и тем же именем в рамках одной области действия, то говорят, что они перегружены.

При вызовах перегруженных функций путем сравнивания типов действительных передаваемых параметров с формальными (указанными в объявлении функции) выбирается нужный вариант кода функции.

Например, известные функции:

char * itoa(int num, char * s, int radix);

char * ltoa(long num, char * s, int radix);

char * ultoa(unsign long num, char * s, int radix); преобразуют значения чисел типов int, long и unsign long в строку символов, помещаемых в буфер, на начало которого указывает s; radix - система счисления (2 <= radix <= 36).

Все функции описаны в заголовочном файле stdlib.h стандартной библиотеки Borland C++. Определенное неудобство работы с таким набором функций очевидно. Необходимо помнить, какая из функций какому типу данных соответствует. Используя механизм перегрузки (переопределения) функций, можно добиться определенного упрощения ситуации. Мы можем ввести описание новой функции, например, с именем "toa", которая позволит выполнять преобразования, автоматически определяя, какой конкретный код функции необходимо использовать:

```
#include <iostream.h>
#include <stdlib.h>
char *toa(int num, char *s, int r)
{return itoa(num,s,r); }
char *toa(long num, char *s, int r)
{return ltoa(num,s,r); }
char *toa(unsigned long num, char *s, int r)
{return ultoa(num,s,r); }
void main(void)
{ char str[81];
int r;
cout << "Задайте систему счисления от 2 до 36" << “\n”;
cin >> r;
cout << "int = " << toa(3248,str,r) << “\n”;
cout << "long = " << toa(773681L,str,r) << “\n”;
```

```
cout << "unsigned long = "<<toa(3456789000UL,str,r)<< “\n”;}
```

В примере вызов той или иной функции будет инициироваться в зависимости от типа первого передаваемого аргумента. Все функции возвращают указатель на полученную строку. Протокол работы программы может быть следующий:

```
Задайте систему счисления от 2 до 36.
10 <ввод>
int = 3248
long = 773681
uhsigned loug = 3456789000
```

Рассмотрим еще один пример перегрузки функций.

Представим, что в некоторой задаче необходим следующий алгоритм: если числа, с которыми оперирует функция, целые, выполняется сложение чисел, если числа вещественные, функция вычитает числа. Данный алгоритм в C++ можно реализовать следующим способом:

```
#include <iostream.h>        // первое описание функции
 iut fun (int i, int j)
 { cout << "функция оперирует целыми числами\n";
 return i+j;}                // второе описание функции
 double fun (double i, double j)
 { cout << "функция оперирует вещественными числами\n"; return i-j; }
 void main (void) {Cout << "2+3 = " << fun (2,3) << "\n";          // целые числа
  cout<<"2.50-3.6="<< fun (2.5,3.6) << "\n" ;}     // вещественные числа.
```

Использование перегруженных функций упрощает разработку программ, улучшает их мобильность, читаемость, сокращает число библиотечных функций. Наиболее распространенный случай использования перегруженных функций - перегрузка конструктора. Такая перегрузка обеспечивает несколько вариантов создания новых объектов класса.

Пример перегрузки конструктора:

```
#include <iostream.h>
class Over { int i;
char *str; public:
Over() { str = "первый конструктор\n"; i = 0; }; // это первый конструктор
Over(char *S) { str = S; i = 50; };
//это второй конструктор, строка символов может задаваться
```

```
        // извне методов класса
        Over(char *S,int X) { str = S; i = X; };
        //это третий конструктор, принимающий параметры S и X
        // и инициализирующий частные переменные класса
        Over(int *Y) { str = "четвертый конструктор\n"; i = *Y; };
        //конструктор принимает в качестве
        //параметра указатель и присваивает значение, извлекаемое
        // из переданного адреса, частной переменной i.
        void print(void) { cout << "i = " << i << "; str = " << str; }};
        //объявление и определение метода класса
 void main(void) { int a = 10,*b;     //переменная типа int и указатель на тип int
 b = &a;                              //инициализация указателя адресом переменной
//объявление четырех объектов: my_over, my_over1 и my_over2;
 Over my_over,                        //для конструктора без параметров
my_over1("для конструктора с одним параметром\n"),
my_over2("для конструктора с двумя параметрами\n",100),
my_over3(b);                         // для конструктора с указателем
//вызовы метода print() для объектов:
my_over.print();              // для объекта, созданного конструктором Over()
my_over1.print();     // для объекта, созданного конструктором Over(char *S)
my_over2.print();     // для объекта, созданного конструктором Over(char *S,int X)
my_over3.print();     // для объекта, созданного конструктором Over(int *Y) }
```

В примере определено 4 конструктора: без параметров, с одним параметром - ссылкой на тип строка, с двумя параметрами и с одним параметром - ссылкой на int.

Объявление четырех экземпляров объектов класса приводит к вызову соответствующих конструкторов и инициализации данных информацией, передаваемой при объявлении объектов.

Результат работы программы:

```
i = o ; str = первый конструктор
i = 50; str = для конструктора с одним параметром
i = 100;str = для конструктора с двумя параметрами
i = 10; str = четвертый конструктор
```

Замечание:так как деструктор не имеет списка аргументов, то нет способа выбрать ту или иную функцию при попытке перегрузить деструктор. Поэтому перегрузка деструктора невозможна.

ПЕРЕГРУЗКА ОПЕРАТОРОВ

Практически в любом процедурном языке используются изначально перегруженные операторы (операции). Так, арифметические операции "знают", как манипулировать с различными типами данных (например, в языках Си, Паскаль операция "+" знает, как обрабатывать типы int, float и др.). Очевидно, что хотя используется одна и та же арифметическая операция, генерируемый исполняющий код операций разный, так как целые и вещественные числа представляются в памяти по-разному. Мы уже использовали перегруженный знак операции сдвига ">>" и "<<" для выполнения ввода-вывода в C++. Кроме того, C++ дает возможность перегружать практически все предопределенные (имеющиеся в конкретном языке) операции. Рассмотрим, как реализуется представленная пользователю возможность переопределения операций.

Для переопределения операции следует описать функцию, которая в качестве своего имени использует ключевое слово operator и символ перегружаемой операции. Если некоторый оператор перегружен для какого-либо класса, то он позволяет реализовать новые действия в контексте этого класса.

Представим задачу, в которой необходимо часто вычислять среднее значение параметров двух объектов. Можно выполнять эту операцию при помощи вызова специально разработанной функции, но гораздо удобнее переопределить некоторую операцию (оператор-функцию) и использовать ее в дальнейшем для нахождения средних значений. Для этого необходимо выбрать одну из уже существующих операций в языке C++. Пусть этой операцией будет, например, операция "^" - (поразрядное исключающее или). Переопределим ее таким образом, чтобы данная операция вычисляла средние значения атрибутов (некоторых показателей a и b, присущих совокупности объектов - классу объектов) двух суммируемых объектов.

Пример:

```
#include <iostream.h>        //файл описаний иерархии потоков ввода-вывода
class my_obj { int a,b;      //атрибуты объектов public:
    my_obj *ptr;             //указатель на объект класса
    my_obj(int x=0,int y=0)
    //конструктор принимает два параметра с заданными по умолчанию
    //значениями
    {a=x; b=y;}              //инициализация переменных
    my_obj & operator^ (my_obj obj_two) //заголовок функции
```

```
//переопределения операции ^. Имя функции "operator^", функция
//принимает объект obj_two в качестве аргумента и возвращает ссылку
//на объект класса my_obj.
{my_obj sum;                    //sum - объект для получения результата
sum.a=(a+obj_two.a)/2;          //вычисление усредненных значений
sum.b=(b+obj_two.b)/2; return *(ptr=&sum); }
//возврат результирующего объекта
my_obj operator= (my_obj obj_sec)      //переопределение оператора =
{a=obj_sec.a;                   //для присваивания объектов класса
b=obj_sec.b; return (*this);}   //возвращается ссылка на объект,
//для которого вызывается функция переопределения "="
friend ostream& operator<< (ostream & a,my_obj obj_rez); }; //переопределение
оператора
        //вывода для вывода объектов класса реализуется
        //дружественной функцией
ostream& operator<< (ostream & a,my_obj obj_rez)
        //определение функции
{ cout<<"\nзначение атрибутов объекта:\na="<<obj_rez.a<< ";
b="<<obj_rez.b; return a; }
void main() { my_obj a1(10,20), a2(45,30),sum1;
        //объявление объектов
sum1=a1^a2;       //вычисление среднего значения
cout<<sum1;}              //вывод значений атрибутов результата
```

Результаты работы программы:

значение атрибутов объекта:

a=27; b=25

Чтобы получить возможность записать равенство sum1=a1^a2, где a1, a2 и sum1 – объекты класса my_obj (см. пример) и распечатать полученное среднее значение результирующего объекта в удобочитаемом виде, были введены также переопределения для операции присваивания "=" и операции вывода "<<" (для случаев, когда слева и справа в указанных операциях операнды являются объектами класса).

В переопределении операций "^" и "=" переопределяющая функция реализована как метод класса. Поэтому в списке передаваемых параметров функции

определен только второй операнд, в примере это объекты класса my_obj - obj_two и obj_sec соответственно. В выражениях они играют роль операндов, записываемых справа от переопределяемой операции. Первый параметр функции (левый операнд в выражениях с переопределенной операцией) не объявляется в списке передаваемых параметров. Считается, что в переопределяющем методе класса первый параметр должен быть объектом класса по умолчанию. Тип второго параметра задан в круглых скобках в списке передаваемых параметров метода.

Для переопределения операции вывода требовалось указать типы обоих передаваемых в функцию operator<< параметров. Поэтому данная функция реализована как friend - функция.

Следует отметить, что вводимое переопределение оператора не может изменить ни числа аргументов, ни правил приоритета оператора по сравнению с его нормальным использованием. Переопределение возможно только для уже существующих в языке операций, кроме операций "." задание элемента класса, ".*" - прямое обращение к элементу класса, "::" - определение области доступа, "?:" - условная операция.

ВИРТУАЛЬНЫЕ ФУНКЦИИ

Программирование решения задачи в ООП следует начинать с разработки иерархической структуры классов и объектов, которые участвуют в реализуемом процессе.

Общие атрибуты - данные и особенности поведения, присущие всем объектам формируемой иерархии классов, удобно описывать на самых верхних уровнях иерархии. При этом методы, код которых можно определить без знания особенностей конкретных объектов, объявляются и определяются как обычные методы классов. Они распространяют свое действие по иерархии через механизм наследования. Вызовы таких функций требуют определения связей на этапе компиляции, что известно как метод раннего или статического связывания. Накладываемые таким связыванием ограничения не позволяют пользователю легко добавлять новые образцы классов и расширять свою программу.

C++ снимает указанные ограничения, представляя механизм позднего (или динамического) связывания с помощью специальных методов, называемых виртуальными функциями. При динамическом связывании вызовы функций связываются с конкретным кодом на этапе исполнения. На практике это означает

следующее: решение, какую конкретно функцию вызывать, откладывается до момента выявления типа объекта на стадии исполнения программы. Вызывается тот вариант кода исполняемой функции, который соответствует обрабатываемому в данный момент объекту.

Виртуальные функции (методы) объявляются со спецификацией virtual в базовом классе. Они переопределяются в одном или более произвольном классах. При этом их имена, тип возвращаемого значения, число и типы передаваемых аргументов не меняются (полностью совпадают). Функция объявляется виртуальной добавлением модификатора virtual в первое объявление метода.

Например, пусть реализуется некоторая иерархия классов, реально проявляющих себя, как некоторое множество графических примитивов, отображаемых на экране дисплея. Для их прорисовки разработаны методы (для каждого примитива свой код отображения примитива на экране). Все эти методы имеют одну и ту же сигнатуру реализующих функций (имя функции, тип возвращаемого значения, количество и тип принимаемых параметров), и их прототип объявлен в базовом классе с ключевым атрибутом virtual.

Таким образом, данная совокупность методов удовлетворяет определению виртуальных. Пусть иерархия классов имеет в некотором базовом классе Base (самом верхнем в иерархии, где метод впервые встречается) прототип "virtual void my_virt()" и некоторый реализующий код. В каждом из производных классов, где нужна прорисовка, метод переопределяется с той же сигнатурой (можно уже без ключа virtual) и собственным реализующим кодом.

При компиляции создается специальная таблица виртуальных методов. Связывание вызова виртуального метода с конкретным кодом реализуется с ее помощью при выполнении программы, в зависимости от того, для объекта какого класса вызывается метод. Таким образом, вызывается именно тот метод, который необходим для прорисовки конкретного объекта.

В программах на C++ это достаточно элегантно реализуется с помощью, например, механизма указателей. Так, если объявить указатель на базовый класс Base (Base * ptr), в котором введено определение виртуального метода для прорисовки объекта - my_virt, то вызов рД>my_virt будет выбирать различные версии виртуальной функции в зависимости от того, на объект какого класса мы установим указатель. Пусть в программе объявлены объект базового класса my_obj_1 - (Base my_obj_1) и объект производного класса my_obj_2 - (Derive my_obj_2). В каждом из

классов прорисовка объекта выполняется своим собственным виртуальным методом с именем my_virt(). Тогда при инициализации указателя адресом объекта базового класса (ptr=&my_obj_1) запись ptr->my_virt приведет к вызову метода из класса Base, в то время как при инициализации (ptr=&my_obj_2) такая же запись ptr->my_virt вызовет метод из производного класса Derive.

При использовании виртуальных функций имеют место следующие ограничения:

1) прототипы виртуальной функции в базовом классе и во всех производных классах должны соответствовать друг другу (тип возвращаемого значения, имя функции, число и тип принимаемых параметров должны совпадать). В противном случае функция рассматривается как перегруженная, а не виртуальная;

2) виртуальная функция должна быть компонентом класса;

3) конструктор не может быть виртуальной функцией, хотя деструктор можно объявить со спецификатором virtual.

Если функция не описана (пропущена) в производном классе, то используется версия базового класса.

Использование виртуальных функций.

Обычно при проектировании программы общие атрибуты иерархических классов представляются в определении базовых классов, частные - в особых производных классах.

Функции, которые можно определить без знания особенностей конкретного объекта, используемого в программе, объявляются как обычные методы классов или функции программ.

Функции, определения которых зависят от конкретных параметров объектов, удобно объявить виртуальными.

Отметим, что механизм виртуальных функций и множественного наследования дает естественную возможность расширения библиотечных приложений путем наследования всего, что присуще базовым классам, с одновременной возможностью включения дополнительных средств, чтобы заставить новые объекты работать надлежащим образом.

Рассмотрим демонстрационный пример использования виртуальных функций:

```
struct B {
virtual void vf1();
virtual void vf2();
```

```
virtual void vf3();
void f();}
class D: public B {
virtual void vf1();
//виртуальная ф-я, спецификатор virtual допустим, но избыточен
void vf2(int);
//другой список аргументов, функция не является виртуальной
char vf3();//ошибка, т.к. изменяется только тип возвращаемого значения,
//что недопустимо
void f();};                    //обычный метод класса D
void main (void) { B b, *bp; //объект и указатель класса B
bp=&b;                              //присвоение указателю адреса объекта
bp->vf1();                     //вызывается B::f1();
D d;                           //объявлен объект d класса D
bp=&d;        //присвоение указателю на базовый класс B адреса объекта
//производного класса d;
bp->vf1();                     //вызывается D::vf1();
bp->vf2();                     //вызывается B::vf2();, т.к. функция
//vf2(int) из D имеет другой список аргументов и не является
//виртуальной функцией;
bp->f(); }      //вызов B::f() - не виртуального метода из класса B, так как
//невиртуальные методы класса D для указателя из класса B неизвестны.
```

Указатель "видит" только имена методов, которые объявлены в его собственном классе (классе, в котором объявлен указатель). Переопределенная функция vf1() в классе D автоматически становится виртуальной. Используемый для vf1() спецификатор virtual, вообще говоря, является избыточным.

Интерпретация вызова виртуальной функции зависит от типа объекта, для которого она вызывается. Связывание осуществляется в момент реального вызова метода. В случае вызова невиртуальной функции интерпретация зависит только от типа указателя или ссылки, с помощью которого они вызываются. Вызывается объект из класса, в котором объявлен указатель.

Функция, объявленная виртуальной, сохраняет это свойство (свойство виртуальности) для любого производного класса, на любом уровне вложенности.

Если в некотором производном классе виртуальная функция не переопределена, то используется ее версия из базового класса.

АБСТРАКТНЫЕ КЛАССЫ

Для ситуаций, в которых при пропуске переопределения виртуальной функции в производном классе нельзя использовать ее версию из базового класса, можно объявить так называемую чистую (pure) виртуальную функцию (функцию без постороннего эффекта).

Синтаксис объявления:

virtual тип_возвращаемого_значения имя_функции (список_передаваемых_параметров)=0 , где "=0" - означает, что объявлена чистая виртуальная функция.

Такое объявление требует от любого производного класса либо наличия определенной версии кода реализации виртуальной функции, либо объявления ее в этом классе чистой виртуальной функцией.

Если класс имеет хотя бы одну чистую виртуальную функцию, то он называется абстрактным. Для абстрактного класса нельзя создавать объекты. Он может использоваться только в качестве базового для других классов. Абстрактный класс не может использоваться как тип аргумента и как тип возврата функций. Однако допускается объявлять указатели на абстрактный класс, если при его инициализации не требуется создание хотя бы временного объекта такого класса.

Нами рассмотрены основные методы реализации идей ООП в C++. Они основаны на введении нового типа данных - класса и новых методов манипулирования с объектами класса.

Так как класс - это новый тип данных, то с объектами класса возможны и привычные манипуляции, выполняемые над данными, как, например:

определение массива объектов, указателей на объект, передача объектов в качестве параметров функции, и т.п.

Так, если объявить класс:

Class My_ob { .....}, то возможны следующие действия:

void F1(My_ob p) {....} //передача объекта функции;

My_ob Mas[10]; //объявление массива из 10 объектов;

My_ob a,*b;

b=&a; // b - является указателем на объект a,

тогда обращение к его компонентам можно осуществлять с помощью знака ->. Если К - компонент объекта a, то после инициализации b=&a;, к нему можно обратиться следующим образом: b->К.

Если объявить: My_ob mas[10],*p;

и присвоить: p=&mas[0]; , то (p+1) является указателем на следующий элемент mas[1], т.е. выполняется смещение указателя на размер объекта.

Продемонстрируем выполнение различных операций с объектами следующим примером:

```
 #include <iostream.h>
class My
 { int i;
public:
My() { i = 0; } // это конструктор
void put_i(void) { cout << "i = " << i << "\n"; }     //метод вывода значения i
void set_i(int Y) { i = Y; }};            //метод инициализации значения i
void function(My x)  // функции function передается объект x типа My
{ My *y;                        //y - указатель на объект типа My
y = &x;                         // теперь y - указатель на объект x типа My
x.put_i();                      // обращение к объекту по имени
y->put_i();};                   // обращение к объекту через указатель
void main(void)
{ My mas[4],*p;                          // объявление массива mas из четырех
// объектов и указателя p на объект типа My
mas[0].set_i(20);                       // присвоение значений компоненту i
mas[1].set_i(50);                       // объектов mas[0], mas[1] и mas[2]
mas[2].set_i(100);
p = &mas[1];                    // p устанавливается на адрес объекта mas[1]
p->put_i();                     // вызов функции, вывод значения 50
(p+1)->put_i();                         // вывод значения 100
(p-1)->put_i();                         // вывод значения 20
function(mas[1]);               // вызов функции и передача ей объекта
        // mas[1] в качестве параметра (выведутся значения 50 и 50)
function(mas[3]); }             // вызов функции и передача ей объекта
                // mas[3] в качестве параметра (выведутся значения 0 и 0)
```

Результаты выполнения программы:

i=50

i=100

1=20

i=50

i=50

i=0

i=0

Описание примера приведено в комментариях к нему.

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

До сих пор мы рассматривали наследование только от одного базового класса. При моделировании реальных объектов и процессов приходится иметь дело с данными и свойствами, унаследованными от нескольких базовых классов.

Рассмотрим пример, демонстрирующий, каким образом реализуется множественное наследование при моделировании иерархии объектов.

```
#include <iostream.h>
class Base_1          //объявление базового класса Base_1
{ int a;              // частная переменная
protected:
int b;
// защищенная переменная, имеется доступ из функций производного класса
public:
Base_1(int x,int y);  //объявление конструктор
~Base_1();            // деструктор
void show1(void)              // метод класса Base_1
{ cout << "a = " << a << "; b = " << b; } };
class Base_2          // объявление базового класса Base_2
{protected:
int c;
//доступ к переменной только из методов собственного и производных классов
public:
Base_2(int x);                // конструктор класса
~Base_2();            // деструктор
```

```
void show2(void) { cout << "; c = " << c << "\n"; }          //метод };
class Derive : public Base_1, public Base_2
                    //класс производный от Base_1 и Base_2.
{ int p;
public:
Derive(int X,int Y,int Z);      // конструктор
~Derive();                  //деструктор
void show3(void) { cout << "a + b + c = " << p+b+c; }};
          // метод производного класса
Base_1::Base_1(int x,int y) { a=x; b=y;
cout << "\nконструктор Base_1"; }          //определение конструктора Base_1
Base_1::~Base_1() { cout << "\nдеструктор Base_1"; }
// определение деструктора Base_1
Base_2::Base_2(int x) { c=x;
cout << "\nконструктор Base_2"; }          //определение конструктора Base_2
Base_2::~Base_2()
{ cout << "\nдеструктор Base_2"; }          //определение деструктора Base_2
Derive::Derive(int X,int Y,int Z) : Base_1(X,Y), Base_2(Z)
//определение конструктора производного класса Derive
{ p=X; cout << "\nконструктор Derive\n"; }
Derive::~Derive()                    //деструктор Derive
{ cout << "\nдеструктор Derive"; }
void main(void) { Derive my_d(3,5,7);      //создание и инициализация объекта
// вызывается конструктор класса Derive, который сначала
//вызывает выполнение базовых конструкторов класса Base_1 и класса Base_2
my_d.show1();         // выполнение метода класса Base_1 для объекта my_d
my_d.show2();                  // метод класса Base_2 для my_d
my_d.show3();                  // метод класса Derive для my_d }
```

Результаты работы программы:

```
конструктор Base_1
конструктор Base_2
конструктор Derive
a=3; b=5; c=7
a+b+c=15
```

деструктор Derive

деструктор Base_2

деструктор Base_1

Отметим, что вместо частной переменной a класса Base_1 при вычислении суммы значений a+b+c используется переменная p класса Derive, доступ к которой обеспечивается из функции члена show3();. Значение 3, присваиваемое переменной a класса Base_1 и p класса Derive, выбирается из значений первого аргумента в определении объекта my_d (3,5,7) класса Derive.

СТАТИЧЕСКИЕ ОБЪЕКТЫ

Рассматривая примеры, мы не заботились о памяти, которую необходимо выделить для определяемых объектов. Тем не менее данные программы работают, если попробовать ввести их исходные тексты и выполнить в одном из компиляторов, например фирмы BORLAND для C++. Объекты, которые мы использовали в программе, являются автоматическими. Память для них выделяется стандартным образом на этапе компиляции. Для каждого объекта использовалась своя копия элементов данных класса.

В случае если возникает потребность в определении некоторых разделяемых данных, которыми бы пользовались совместно все объекты данного класса, можно использовать декларацию (объявление) статических переменных (static).

Пример объявления:

```
class my_class {my_class * prt;
        static my_class my_stat;
};
```

Такое объявление гарантирует, что будет только один экземпляр переменной my_stat , а не по экземпляру для каждого объявленного объекта.

Так как объявление сделано в шаблоне класса, то чтобы сделать его доступным извне, необходимо объявить его как public и при обращении уточнять именем класса, например:

my_class::my_stat

Переменные, объявленные со спецификатором Static, получают распределение памяти в начале выполнения программы и сохраняются до выхода из нее. При отсутствии явного инициализатора или конструкторов в C++ объекты со

статической продолжительностью инициализируются нулем или NULL (для указателей).

Пример использования статических переменных:

```
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
class a1 { public: static int s; static void f1 (void); };
int a1::s=5;                    //определение статической переменной
void a1::f1 (void) { s+=21;             //определение метода
printf("s=%d ; ",s);                    //вывод значения переменной
getch();                        //останов до нажатия клавиши
}
 void main(void) { a1::s+=1;            //+1 в статическую переменную
a1 ob_a1,ob_a2;/            /определяем два объекта класса
a1 ob_a1.f1();              //вызов метода для объекта ob_a1
ob_a1.s+=5;                 //+5 в статическую переменную
ob_a2.f1();                 //вызов метода для ob_2
}
```

Результат выполнения программы:

s=27 ; s=53 ;

Объяснение выполнения программы дано в комментариях.

ДИНАМИЧЕСКИЕ ОБЪЕКТЫ

Динамические объекты создаются во время выполнения программы с выделением памяти из пула свободной памяти. Такой прием используется, в случае если мы не можем определить объем запоминаемых в памяти данных до начала выполнения программы (работа со списками, записями переменной длины, сложными структурами данных).

В C++ для динамического размещения объектов можно использовать как функции языка C (например malloc), так и свои собственные более мощные операторы. Это операторы new и delete.

Формат их использования:

1) указатель_на_выделенную_память = new имя_типа (инициализирующее значение);

2) delete указатель_на_выделенную_память.

Имя может быть любого типа, кроме функции. Допустимым является указатель на функцию. Инициализирующее значение не является обязательным.

Оператор new выделяет требуемую память и возвращает указатель на ее начало. При ошибке и в случае невозможности выделения нужного объема памяти оператор возвращает значение NULL.

Оператор delete освобождает ранее выделенную оператором new память по заданному указателю на ее начало.

Преимущества операторов new и delete по сравнению с функциями C (типа malloc и free):

1. Оператор new вычисляет размер элемента, для которого необходимо выделить память (нет необходимости использовать операторы определения размера типа sizeof).

2. Отсутствует необходимость в преобразовании типов, поскольку автоматически возвращается указатель на нужный элемент.

3. Допускается инициализация выделенного блока памяти.

4. Возможна перегрузка операторов new и delete относительно заданного класса.

И наиболее важное!

5. Они гарантируют вызов конструкторов и деструкторов.

Пример:

Circle * Acircle=new Circle(120,80,50);

Указателю Acircle на тип Circle присваивается адрес блока памяти, достаточный для размещения одного объекта типа Circle.

Для размещения массива используется форма:

mas=new int[100];

Объекты, распределяемые new, за исключением массивов, могут инициализироваться выражением в круглых скобках. При создании многомерных массивов с помощью new следует указывать все размерности, т.е.:

mas_prt=new int [3] [10] [12];

Для иллюстрации сказанного воспользуемся введенными ранее определениями классов Location и Point. Т.е. предположим, что определение данных класса Location - содержит информацию о координатах точки X и Y , а класс Point, который является производным от класса Location, наследует все то, что имеет класс Location, и добавляет специфическую для вывода точки информацию. Если представить, что определения классов Location и Point содержатся в файле Point.h, то создать динамический объект можно выполнив следующие действия.

Пример:

```
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
#include "Point.h"
int main (void)
{ Point * APoint=new Point(50,100);
//вызов конструктора и создание динамического объекта
                                //с размещением его адреса в APoint
int gd=DETECT,gm;
initgraph(&gd,&gm,"");              //инициализация графического режима
APoint -> Show();           //метод класса Point для вывода точки
cout <<"Вывод точки с координатами (50,100).\
 Нажми любую клавишу для продолжения";
//символ обратный слеш "\" использован для операции склеивания текстовой
// строки в программе
getch();
delete APoint;                      //явное удаление объекта
closegraph();                       //закрыть графическую систему
return(0); }
```

Пример демонстрирует создание нового динамического объекта, его инициализацию и организацию вывода точки при помощи метода Show класса Point.

Для статических и автоматических объектов память распределяется компилятором. Для статических объектов конструктор вызывается перед main , а деструктор – после main.

В случае автоматических объектов конструктор вызывается при выполнении определения объекта, а деструктор – по окончании области действий, т.е. при завершении блока, включающего определение объекта. При этом освобождается занимаемая объектом память. Выполняются либо конструкторы и деструкторы, определенные программистом, либо неявные, вызываемые компилятором по умолчанию.

Создавая динамические объекты с помощью new, программист полностью берет на себя ответственность за его последующее уничтожение, т.к. C++ не может знать, когда объект больше не нужен. Уничтожение осуществляет функция delete. При

выполнении команды delete вызывается соответствующий деструктор. Оператор delete можно применять только к указателям, инициализированным при помощи оператора new, или к указателю NULL. Для массивов их в операторе delete необходимо задавать явно.

Еще раз отметим, что C++ не гарантирует выполнение деструктора для объекта, созданного при помощи оператора new.

Например:

```
void main (void)
{ table * p=new table(100);
table * q=new table(200);
delete p;
delete p;}
```

В данной функции объект с адресом в q не удаляется из памяти, а объект p удаляется дважды. Использование таких функций не может закончиться ничем хорошим.

Имеется интересная возможность определить, вызван ли конструктор оператором new или каким-либо другим способом. Если конструктор вызван через оператор new, то указатель this на входе в конструктор имеет значение 0. В другом случае он указывает на пространство, занятое объектом. Отсюда следует, что можно написать конструктор, который будет отводить память, например, только тогда, когда он был вызван оператором new. К сожалению, для деструктора такой возможности нет.

Анализом рассмотренного примера можно завершить теоретический материал с названием "Введение в ООП на C++". Он дает представление о базовых элементах ООП. Их усвоение позволит вам самостоятельно расширять и углублять знания в области проектирования и реализации программ на C++.

2.4. Сетевые информационные технологии[7-15]

Компьютерные сети позволяют соединить между собой группу компьютеров для совместного использования информации.

Задачи, которые решаются с помощью компьютера, работающего в составе сети:

разделение файлов;

передача файлов;

доступ к информации и файлам;

разделение принтера;

электронная почта.

Компьютерная сеть – это совокупность компьютеров, кабелей, сетевых адаптеров, работающих под управлением сетевой операционной системы и прикладного программного обеспечения.

Модели сетей:

одноранговая (прямое взаимодействие компьютеров друг с другом на равных условиях);

с выделенным сервером (рабочие станции клиентов сети подключаются к выделенным серверам).

В сети каждый компьютер называется **рабочей станцией**, за исключением одного или нескольких компьютеров, которые предназначены для выполнения функций файл-серверов. Файловый сервер обслуживает рабочие станции. Каждые рабочая станция и файловый сервер содержат карты адаптеров, которые с помощью сетевых кабелей соединяются между собой. Описание способа физического соединения компьютеров в сети называется **топологией**.

Организация ISO (International Standards Organization – Международная организация по стандартизации) опубликовала модель архитектуры вычислительной сети, названной OSI (Open System Interconnection – Связь открытых систем). **Модель OSI** разделяет коммуникационные функции в сетях на семь уровней **протоколов** (перечень коммуникационных правил и форматов, которым должны следовать компьютеры в сети, чтобы обмениваться данными). Взаимосвязь уровней друг с другом осуществляется через интерфейсы. **Уровни OSI**: прикладной, представления, сеансовый, транспортный, сетевой, канальный и физический.

Интернет представляет собой “сеть сетей”, которая охватывает весь земной шар. **Интранет** – это частная корпоративная сеть (обычно в пределах одной компании), и вход в нее из Интернета закрыт для посторонних.

World Wide Web (WWW или **Web**) представляет из себя миллионы связанных между собой документов, которые расположены на компьютерах по всему миру.

Browser (броузер) – это программное обеспечение, которое позволяет просматривать и взаимодействовать с документами Web.

Для создания документов Web используется специальный язык **HTML** (HyperText Markup Language), который позволяет встраивать в них специальную информацию для броузеров.

URL (Uniform Resource Locator) – компьютерный адрес ресурса Интернета.

TCP/IP (Transport Control Protocol / Internet Protocol) – два протокола, лежащих в основе Интернета, позволяющий связывать между собой разнородные и однородные сети.

Уровни протокола TCP/IP: прикладной, транспортный, сетевой, канальный и физический.

Протокол HTTP (HyperText Transfer Protocol) является основным протоколом Web. Серверы и броузеры используют его для передачи Web-документов по Интернету. HTTP можно представить как надстройку над протоколом TCP/IP. Программа-броузер устанавливает соединение с сервером Web по протоколу TCP/IP. После соединения броузер и сервер могут обмениваться сообщениями как посредством HTTP, так и TCP/IP.

Электронная почта – система передачи сообщений, функционирующая в какой-либо коммуникационной среде. Некоторые системы электронной почты работают через обычную телефонную сеть, другие - в рамках локальной сети под управлением операционной системы.

2.5. Технологии проектирования и поддержки баз данных (БД) [16-23]

Файл - это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные. Правила именования файлов, способ доступа к данным, хранящимся в файле, и структура этих данных зависят от конкретной системы управления файлами и, возможно, от типа файла. **Система управления файлами** берет на себя распределение внешней памяти, отображение имен файлов в соответствующие адреса во внешней памяти и обеспечение доступа к данным. Другими словами, **файловые системы** обычно обеспечивают хранение слабоструктурированной информации, оставляя дальнейшую структуризацию прикладным программам.

Информационные системы, напротив, ориентированы на хранение, выбор и модификацию постоянно хранимой информации, имеющей сложную структуру. На начальном этапе использования вычислительной техники проблемы структуризации данных решались индивидуально в каждой информационной системе. Стремление

выделить общую часть информационных систем, ответственную за управление сложноструктурированными данными, явилось первой побудительной причиной создания систем управления базами данных (СУБД). СУБД решают множество проблем, которые затруднительно или вообще невозможно решить при использовании файловых систем.

Основные функции СУБД:

управление данными во внешней памяти;

управление буферами оперативной памяти;

управление транзакциями;

журнализация и восстановление БД после сбоев;

поддержание языков БД.

Проектирование БД в общем случае включает три самостоятельных этапа: концептуальное, логическое и физическое. На этапе концептуального проектирования осуществляются сбор, анализ и упорядочивание (редактирование) требований к данным, построение локальной и композиционной модели данных. На этапе логического проектирования осуществляется выбор модели СУБД и описание данных при помощи ее логических структур (таблиц, файлов, списков и т.д.). На этапе физического проектирования решаются вопросы, связанные с производительностью системы, определяются структуры хранения данных на физических носителях, методы доступа и т.п.

Основными критериями, которым должна удовлетворять спроектированная БД, будем считать обеспечение функциональных требований приложений, целостности и согласованности информации.

В теории проектирования реляционных баз данных, основанной на принципе нормализации отношений, обычно выделяется такая последовательность **нормальных форм**:

первая нормальная форма (1NF);

вторая нормальная форма (2NF);

третья нормальная форма (3NF);

нормальная форма Бойса-Кодда (BCNF) (правильнее было бы считать эту нормальную форму третьей, однако по историческим причинам третья ступень оказалась занятой к моменту изобретения BCNF, из-за чего она и получила нестандартное название);

четвертая нормальная форма (4NF);

пятая нормальная форма, или нормальная форма проекции-соединения (5NF или PJ/NF).

Основные свойства нормальных форм:

каждая следующая нормальная форма в некотором смысле улучшает свойства предыдущей;

при переходе к следующей нормальной форме свойства предыдущих нормальных форм сохраняются.

В основе классического процесса проектирования лежит метод нормализации, который опирается на декомпозицию (на основе проекции) отношения, находящегося в предыдущей нормальной форме, в два или более отношения, удовлетворяющих требованиям следующей нормальной формы.

Наиболее важные на практике нормальные формы отношений (первые три формы) основываются на фундаментальном в теории реляционных баз данных понятии функциональной зависимости. Материал изложен в работах [18,19,22].

Потребности проектировщиков баз данных в более удобных и мощных средствах моделирования предметной области породили направление семантических моделей данных. Одна из наиболее популярных семантических моделей данных - **модель "Сущность связи"** (часто ее называют кратко **ER-моделью**). На использовании разновидностей ER-модели основано большинство современных подходов к проектированию баз данных (главным образом реляционных).

Основными понятиями ER-модели являются сущность, связь и атрибут.

Сущность - это реальный или представляемый (виртуальный) объект, информация о котором должна сохраняться и быть доступна. При этом имя сущности - это имя типа, а не некоторого конкретного экземпляра этого типа

Связь - это графически изображаемая ассоциация, устанавливаемая между двумя сущностями. Эта ассоциация всегда является бинарной и может существовать между двумя разными сущностями или между сущностью и ей же самой (рекурсивная связь). В любой связи выделяются два конца (в соответствии с существующей парой связываемых сущностей), на каждом из которых указывается имя конца связи, степень конца связи (сколько экземпляров данной сущности связывается), обязательность связи (т.е. любой ли экземпляр данной сущности должен участвовать в данной связи).

Атрибутом сущности является любая деталь (свойство, характеристика сущности), которая служит для уточнения, идентификации, классификации, числовой характеристики или выражения состояния сущности.

Как и в реляционных схемах баз данных, в ER-схемах вводится понятие нормальных форм, причем их смысл очень близко соответствует смыслу реляционных нормальных форм.

В **первой нормальной форме** ER-схемы устраняются повторяющиеся атрибуты или группы атрибутов, т.е. производится выявление неявных сущностей, "замаскированных" под атрибуты. Во **второй нормальной форме** устраняются атрибуты, зависящие только от части уникального идентификатора. Эта часть уникального идентификатора определяет отдельную сущность. В **третьей нормальной форме** устраняются атрибуты, зависящие от атрибутов, не входящих в уникальный идентификатор. Эти атрибуты являются основой отдельной сущности. Материал излагается в [22-24,26].

CASE-технология представляет собой методологию проектирования ИС, а также набор инструментальных средств, позволяющих в наглядной форме моделировать предметную область, анализировать эту модель на всех этапах разработки и сопровождения ИС и разрабатывать приложения в соответствии с информационными потребностями пользователей. Большинство существующих CASE-средств основано на методологиях структурного (в основном) или объектно-ориентированного анализа и проектирования, использующих спецификации в виде диаграмм или текстов для описания внешних требований, связей между моделями системы, динамики поведения системы и архитектуры программных средств.

Под **транзакцией** понимается неделимая с точки зрения воздействия на БД последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации), такая, что либо результаты всех операторов, входящих в транзакцию, отображаются в БД, либо воздействие всех этих операторов полностью отсутствует. Понятие транзакции имеет непосредственную связь с понятием целостности БД. Очень часто БД может обладать такими ограничениями целостности, которые просто невозможно не нарушить, выполняя только один оператор изменения БД. Поэтому для поддержания подобных ограничений целостности допускается их нарушение внутри транзакции с тем условием, чтобы к моменту завершения транзакции условия целостности были соблюдены. В системах с развитыми

средствами ограничения и контроля целостности каждая транзакция начинается при целостном состоянии БД и должна оставить это состояние целостным после своего завершения. Несоблюдение этого условия приводит к тому, что вместо фиксации результатов транзакции происходит ее откат и БД остается в таком состоянии, в котором находилась к моменту начала транзакции, т.е. в целостном состоянии.

Сериализация транзакций - это механизм их выполнения по некоторому сериальному плану. Обеспечение такого механизма является основной функцией компонента СУБД, ответственного за управление транзакциями. Система, в которой поддерживается сериализация транзакций, обеспечивает реальную изолированность пользователей.

Основная реализационная проблема состоит в выборе метода сериализации набора транзакций, который не слишком ограничивал бы их параллельность.

Существуют два базовых подхода к сериализации транзакций: подход, основанный на синхронизационных захватах объектов базы данных, и подход, основанный на использовании временных меток. Суть обоих подходов состоит в обнаружении конфликтов транзакций и их устранении.

Одним из основных требований к развитым СУБД является **надежность** хранения баз данных. Это требование предполагает, в частности, возможность восстановления согласованного состояния базы данных после любого рода аппаратных и программных сбоев. Очевидно, что для выполнения восстановлений необходима некоторая дополнительная информация. В подавляющем большинстве современных реляционных СУБД такая избыточная дополнительная информация поддерживается в виде журнала изменений базы данных.

Общей целью **журнализации** изменений баз данных является обеспечение возможности восстановления согласованного состояния базы данных после любого сбоя. Поскольку основой поддержания целостного состояния базы данных является механизм транзакций, журнализация и восстановление тесно связаны с понятием транзакции. Общие принципы восстановления следующие: результаты зафиксированных транзакций должны быть сохранены в восстановленном состоянии базы данных; результаты незафиксированных транзакций должны отсутствовать в восстановленном состоянии базы данных.

Основной принцип архитектуры **клиент-сервер** заключается в том, что система разбивается на две части, которые могут выполняться в разных узлах сети, - клиентскую и серверную. Прикладная программа или конечный пользователь

взаимодействует с клиентской частью системы, которая в простейшем случае обеспечивает просто надсетевой интерфейс.

Клиентская часть системы при потребности обращается по сети к серверной части. В качестве основного интерфейса между клиентской и серверной частями выступает язык баз данных **SQL**.

Этот язык, по сути дела, является текущим стандартом интерфейса СУБД в открытых системах. Собирательное название SQL-сервер относится ко всем серверам баз данных, основанных на SQL. Серверы баз данных, интерфейс которых основан исключительно на языке SQL, обладают своими преимуществами и своими недостатками. Очевидное преимущество – стандартность интерфейса. В пределе, хотя пока это не совсем так, клиентские части любой SQL-ориентированной СУБД могли бы работать с любым SQL-сервером вне зависимости от того, кто его произвел.

Недостаток тоже довольно очевиден. При таком высоком уровне интерфейса между клиентской и серверной частями системы на стороне клиента работает слишком мало программ СУБД. Это нормально, если на стороне клиента используется маломощная рабочая станция. Но если клиентский компьютер обладает достаточной мощностью, то часто возникает желание возложить на него больше функций управления базами данных, разгрузив сервер, который является узким местом всей системы.

Одно из перспективных направлений развития СУБД - гибкое конфигурирование системы, при котором распределение функций между клиентской и пользовательской частями СУБД определяется при установке системы. В типичном на сегодняшний день случае на стороне клиента СУБД работает только такое программное обеспечение, которое не имеет непосредственного доступа к базам данных, а обращается для этого к серверу с использованием языка SQL.

Основная задача систем управления распределенными базами данных состоит в обеспечении средства интеграции локальных баз данных, располагающихся в некоторых узлах вычислительной сети, с тем чтобы пользователь, работающий в любом узле сети, имел доступ ко всем этим базам данных как к единой базе данных.

При этом должны обеспечиваться:

простота использования системы;

возможности автономного функционирования при нарушениях связности сети или при административных потребностях;

высокая степень эффективности.

2.6. Технологии и системы искусственного интеллекта. Системы поддержки принятия решений[24-28]

В 50-х годах исследователи в области искусственного интеллекта пытались строить разумные машины, имитируя мозг. Считалось, что если взять сильно связанную систему модельных нейронов, которой вначале ничего неизвестно, применить к ней программу тренировки из поощрений и наказаний, то в конце концов она будет делать всё, что ни задумает её создатель.

Типичной системой, основанной на данном кибернетическом представлении, является **система Розенблата - PERCEPTRON.**

Система PERCEPTRON представляла собой самоорганизующийся автомат, который можно считать грубой моделью сетчатки глаза человека. Его можно было научить распознавать образы, но, как позже показали в своих работах Минский и Пейперт, это был лишь ограниченный класс зрительных образов.

На новые рубежи указали Аллен Ньюэлл и Герберт Саймон из университета Карнеги-Меллона (США), работа которых завершилась созданием **системы GPS** - универсального решателя задач. Они отбросили мысль о сети нейронов, указав, что даже задача конструирования нервной системы муравья, в которой участвует менее тысячи нейронов, выходит за пределы доступной технологии. Они считали, что мышление человека основано на определенном сочетании простых задач манипулирования символами, таких как сравнение, поиск, модификация символа и т.п., - операций, которые могут выполняться компьютером. Решение в пространстве они представляли как поиск(перебор) в пространстве возможных решений по эвристическим правилам, которые помогают направить поиск к искомой цели

Система GPS была универсальной в том отношении, что "не было конкретного указания, к какой области относится задача". Пользователь должен был задать "проблемную среду" в терминах объектов и тех операторов, которые к ним применимы. GPS, как и большинство ее современниц, функционировала лишь в ограниченной области математических головоломок, в таком формализованном микромире, где возникающие проблемы (например, задача "Ханойская башня", задача о миссионерах и людоедах), с точки зрения людей, проблемами и не являются. Реальные задачи система GPS не могла решать.

В 70-х годах группа ученых, возглавляемая Эдвардом Фейгенбаумом из Станфордского университета, вместо того чтобы отыскивать очень эффективные и универсальные эвристики, занялись сужением рассматриваемых вопросов. Они считали, что то, чем, по видимому, располагает специалист, - это набор разнообразных умений, т.е. большое число приёмов и неформальных правил. На этом пути и родилась **экспертная система**, выглядевшая почти как карикатура на специалиста-человека, который узнает всё больше о все меньшем. Прототипом всех экспертных систем является интерпретатор для масс-спектограммы DENDRAL.

Экспертная система (ЭС) - это интеллектуальная система, способная делать логические выводы на основании знаний в конкретной предметной области, предназначенная для оказания консультационной помощи специалистам.

Экспертные системы относятся к категории машинных программ, которые способны выполнять самые разнообразные функции, а именно: консультировать и давать советы, анализировать и классифицировать, обучаться и обучать, проводить поиск, обмениваться информацией, представляя ее в требуемой форме, идентифицировать и интерпретировать, осуществлять диагностику и тестирование, а также составлять проекты, объяснять, исследовать, прогнозировать, вырабатывать концепции, обосновывать, контролировать, планировать и составлять "расписания".

Экспертные системы призваны решать те задачи, где, как принято считать, невозможно обойтись без эксперта-человека. Различают два **типа ЭС**. Системы первого типа предназначены для специалистов, чей профессиональный уровень не слишком высок. В базах знаний таких систем хранятся знания, полученные от специалистов экстракласса. Системы второго типа призваны помогать специалистам высокой квалификации, выполняя для них значительную часть рутинных операций и просмотр больших массивов информации. Особенностью ЭС является наличие в ней системы объяснений, повышающей "консультационную силу" ЭС. Так как ЭС усиливает интеллект человека, необходимо, чтобы система могла оперировать со всеми элементами, составляющими процесс решения задачи. Как правило, ЭС создаются при участии специалистов, которые разъясняют ход своих мыслей в процессе решения конкретных задач. Если такой скрупулезный анализ последовательности действий эксперта удается провести, то составленная машинная программа сможет не хуже, чем эксперт-человек, решать задачи, сформулированные достаточно строго. Специалистам обычно приходится решать плохо определенные задачи, для которых характерно отсутствие четкой внутренней

структуры. Чтобы преодолеть подобную неорганизованность задачи, специалисты используют **эвристики** - эмпирические правила, которые применяются людьми в тех случаях, когда нехватка времени или нечеткое понимание сути проблемы делает невозможным анализ всех имеющихся параметров. Подобным же образом экспертные системы используются для решения задач, запрограммированных эвристическими алгоритмами. В процессе решения задач специалист-человек может:

1. Применять свои знания и опыт для оптимального решения задач. Делать достоверные выводы и умозаключения, исходя из неполных или ненадежных данных.
2. Объяснять и обосновывать свои действия.
3. Общаться с другими экспертами и приобретать новые знания.

Кроме того, специалист-человек может:

4. Заново систематизировать свои знания.
5. "Нарушать" правила. В его распоряжении практически столько же исключений из правил, сколько и самих правил. Эксперт разбирается в правилах не только по их форме, но и по содержанию.
6. Определять степень своей компетентности в каждом конкретном случае. Эксперт представляет себе, какие задачи не входят в сферу его компетенции и в каких случаях следует обращаться за консультацией к другим источникам.
7. Плавно снижать уровень компетентности в каждом конкретном случае. Если процесс решения какой-либо задачи, лежащей на "стыке" разных областей, не укладывается в рамки профессионального опыта эксперта, то его недостаточная компетентность проявляется не во внезапном отказе от принятия решения, а в постепенном ухудшении качества решения.

Современные экспертные системы способны в полной мере имитировать лишь первые три из перечисленных возможностей специалиста.

В связи с этим возникает вопрос: зачем разрабатывать экспертные системы и не лучше ли обратиться к человеческому опыту, как это было ранее?

Существуют веские доводы в пользу применения искусственной компетентности с целью усилить возможности человеческого рассуждения. Основные из этих доводов можно представить в табл. 4.

Таблица 4

Человеческая компетентность	**Искусственная компетентность**
Непрочная.	Постоянная

Трудно передаваемая	Легко передаваемая
Трудно документируемая	Легко документируемая
Непредсказуемая	Устойчивая
Дорогая	Приемлемая по затратам

Но если искусственная компетентность обладает такими преимуществами, можно ли полностью отказаться от экспертов-людей, заменив их экспертными системами? Вероятно, можно отказаться от наиболее квалифицированного эксперта, но во многих ситуациях необходимо оставить в системе место для эксперта со средней квалификацией и использовать экспертные системы для усиления и расширения профессиональных возможностей такого пользователя.

Основные доводы в пользу такого утверждения можно представить в табл. 5.

Таблица 5

Человеческая компетентность	**Искусственная компетентность**
Творческая	Запрограммированная
Приспосабливающаяся	Нуждается в подсказке
Использует чувственное.восприятие	Использует символьный ввод
Широкая по охвату	Узконаправленная
Использует общедоступные.знания (здравый смысл)	Использует специализированные знания

Сведения из внешнего мира воспринимаются человеком с помощью одного из пяти органов чувств (таких как зрение) и затем помещаются в буфер кратковременной памяти для анализа. В другой области памяти (долговременной) хранятся символы и смысловые связи между ними, которые используются для объяснений новой информации, поступающей из кратковременной памяти. В долговременной памяти хранятся не столько факты и данные, сколько объекты и связи между ними, т.е. **символьные образы**. Символьные образы объединены в так называемые **чанки** . Чанки хранятся совместно со взаимосвязями между ними. В каждый момент человек может обрабатывать и интерпретировать не более четырех - семи чанков.

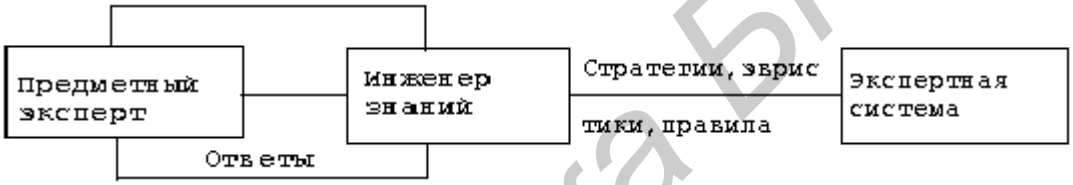
Способность формировать чанки отличает эксперта в конкретной области от неэксперта. Эксперт – это человек, который упорно развивает свою способность

объединять в чанки большие объемы данных и устанавливать иерархические связи между ними для того, чтобы быстро извлекать эти данные из памяти и с их помощью распознавать новые ситуации по мере поступления информации о них в мозг.

Средний специалист в конкретной предметной области помнит от 50000 до 100000 чанков, которые могут быть использованы для решения той или иной проблемы. Накопление в памяти человека и построение указателей для такого объема данных требуют от 10 до 20 лет.

Основой для создания экспертных систем служит **ИНЖЕНЕРИЯ ЗНАНИЙ**.

Создателя ЭС называют инженером знаний . Он "извлекает" из экспертов процедуры, стратегии, эмпирические правила, которые они используют при решении задач в некоторой предметной области, и встраивает эти знания в экспертную систему.



Подобно специалисту - человеку в экспертных системах могут использоваться как **"глубинные", так и поверхностные представления знаний**. К глубинным относятся причинные модели, категории, абстракции и аналогии. В них отображаются понимание структуры и назначение конкретных знаний. Поверхностные представления – это зачастую просто эмпирические ассоциации (взаимосвязи). В них указывается взаимосвязь между исходными посылками и выводами. Причинность скорее подразумевается, а не заложена явно в структуре правила.

При использовании глубинных представлений экспертные системы обладают большими возможностями трактовки знаний. Система, основанная на поверхностных представлениях, "осознает" только наличие эмпирической ассоциации - она не может объяснить причинную связь, а лишь воспроизводит такую ассоциацию. Однако поверхностные представления, отличаясь более низкой стоимостью реализации, тем не менее способны обеспечить приемлемое качество объяснений, а также определенную возможность накопления знаний.

Наилучший подход к созданию экспертных систем состоит, по-видимому, в использовании глубинных представлений там, где это обосновано, и поверхностных представлений - во всех остальных случаях.

Очевидно, что способ организации и накопления знаний является одной из самых важных характеристик экспертной системы.

В основе ЭС находится обширный запас знаний о конкретной области - **база знаний (БЗ)**. База знаний в ЭС может быть представлена как база фактов, база правил, база процедур. Факты представляют собой краткосрочную информацию в том смысле, что они могут изменяться, например, в ходе консультации. Правила представляют более долговременную информацию о том, как порождать новые факты или гипотезы из того, что сейчас известно.

Процедуры используются в процессе управления выводом для организации запросов, приема/передачи сообщений, выполнения вычислений и т.п. Структура базы знаний, как и остальных компонентов ЭС, существенно зависит от способа представления знаний.

Экспертные системы, использующие поверхностные представления, реализуются на основе правил продукции, т.е. на основе **продукционной модели** базы знаний. Для реализации глубинных представлений, как правило, используются **фреймы или семантические сети.**

Для продукционных систем, основанных на правилах, база знаний включает в себя базу правил и рабочую базу.

База правил содержит конструкции ЕСЛИ-ТО, которые получили название **продукционных правил**. Каждое продукционное правило складывается из двух частей. Первая из них - **антецендент**, или посылка правила, - состоит из элементарных предложений, соединенных логическими связками И, ИЛИ и т.д. Вторая часть, называемая **консеквентом** или заключением, состоит из одного или нескольких предложений, которые образуют выдаваемое правилом решение либо указывают на действие, подлежащее выполнению. Антецендент представляет собой образец правила, предназначенного для распознавания ситуации, когда оно должно сработать. Правило срабатывает, если факты из рабочей памяти при сопоставлении совпали с образцом, после чего правило считается сработавшим.

Другой важной частью системы, основанной на знаниях, является **рабочая память**. Под рабочей памятью будем понимать базу, содержащую совокупность исходных и выводимых фактов(данных). В ней хранятся множество фактов, описывающих текущую ситуацию, которые были установлены к определенному моменту. Содержимое рабочей памяти со временем изменяется, обычно увеличиваясь по мере срабатывания правил. После срабатывания каждого правила в рабочую базу добавляются новые факты, т.е. новые факты являются результатом вывода, который состоит в применении правил к имеющимся фактам.

Механизм вывода, или механизм принятия решений, выполняет следующие основные функции:

просмотр существующих фактов из рабочей памяти и правил из базы знаний и добавление (по мере возможности) в рабочую память новых фактов (компонент вывода);

определение порядка просмотра и применения правил (управляющий компонент).

Кроме того, механизм вывода управляет процессом консультации, сохраняя для пользователя информацию о полученных заключениях, и запрашивает у него информацию, когда для срабатывания очередного правила в рабочей памяти оказывается недостаточно данных.

Механизм вывода обеспечивает прямое использование БЗ для решения задач. Организация механизма вывода существенно зависит от способа представления знаний.

В качестве стратегии для логического выбора в целом используется **"прямая" или "обратная" цепочка рассуждений**. Прямая цепочка рассуждений связана с рассуждениями, ведущимися от данных (фактов) к гипотезам, тогда как обратная цепочка связана с попыткой найти данные для доказательства или опровержения некоторой гипотезы. Чисто прямая цепочка рассуждений ведет к неуправляемому режиму задания вопросов, тогда как обратная цепочка будет, как правило, приводить к настойчивому повторению вопросов, касающихся целей.

Для продукционных систем при прямом выводе используются факты, удовлетворяющие условиям некоторых правил, и выводятся новые факты, используемые в дальнейшей цепочке исследований.

При обратном выводе на основании некоторых косвенных данных выдвигается гипотеза, определяется набор правил, который может подтвердить данную гипотезу, и осуществляется исследование фактов для ее подтверждения.

Помимо рассмотренных основных компонентов экспертная система включает в себя подсистемы:

приобретения знаний;

объяснительную подсистему.

Подсистема приобретения новых знаний предназначается для добавления в базу знаний новых правил и модификации имеющихся. В ее задачу входит приведение правила к виду, позволяющему механизму вывода применять правило в

процессе работы. В простейшем случае в качестве такой подсистемы может выступать обычный редактор или текстовой процессор, который просто заносит правила в файл.

В более сложных системах предусмотрены средства для проверки вводимых или модифицируемых правил на непротиворечивость с имеющимися правилами.

Подсистемой объяснения называется компонент экспертной системы, который отвечает на вопросы пользователя о том, как именно получено решение. Эта подсистема должна быть способна в любой момент выдать обоснование принятого решения.

Для взаимодействия пользователя и системы необходимо включить средства общения на естественном языке.

Одной из чрезвычайно трудных сфер профессиональной деятельности человека является область принятия решений по управлению сложными иерархическими системами. **Принятие решения** является заключительной фазой в цикле управления и в силу этого должно опираться на результаты, полученные из других областей деятельности: контроля, диагностики, прогнозирования и планирования. Правильному выбору решения способствуют и результаты прогнозирования последствий их принятия.

Следует отметить, что, рассматривая одну и ту же ситуацию, разные эксперты часто приходят к различным заключениям и, соответственно, предложениям по решению возникающих проблем, т.к. здесь не последнюю роль играет не только знание некоторых формальных правил и алгоритмов принятия решений, но и интуиция эксперта, которая является некоторым подсознательным выводом, основанном на его личном профессиональном опыте. Задача усложняется и тем, что каждый специалист работает на своем уровне иерархии, в котором осуществляет все указанные виды деятельности, предлагает и обосновывает решения, приобретает необходимый опыт эксперта и, следовательно, не может одинаково успешно прогнозировать результаты принятия решения для всех иерархических подструктур системы. В силу этого решения часто не могут быть приняты без привлечения к анализу ситуации специалистов различного уровня и, возможно, различных прикладных областей знаний. Трудности проведения таких консультаций очевидны, и если ситуация в функционирующей системе достаточно часто меняется, то возникает необходимость в содержании целого штата сотрудников, без устали заседающих и решающих. Ситуацию реально могут изменить интеллектуальные системы поддержки

принятия решений (СППР). В этом направлении ведутся интенсивные разработки, что подает надежду на получение реальных результатов.

2.7. Технологии обеспечения безопасности информационных систем[29,30]

Виды уязвимости:

возможность искажения или уничтожения информации (т. е. нарушения ее физической целостности);

возможность несанкционированного использования информации (т.е. опасность утечки информации ограниченного пользования).

Безопасность — обеспечение защиты данных от случайного или преднамеренного разрушения, раскрытия или модификации.

Система безопасности — совокупность средств, методов, мер и мероприятий, создаваемая и поддерживаемая для предупреждения разрушения, раскрытия или модификации защищаемых данных.

Угроза — потенциально существующая опасность случайного или преднамеренного разрушения, раскрытия или несанкционированной модификации данных, обусловленная условиями хранения или обработки этих данных.

Если информация представляет ценность, то необходимо понять, в каком смысле эту ценность необходимо оберегать. Если ценность информации теряется при ее раскрытии, то говорят, что имеется опасность нарушения секретности информации. Если ценность информации теряется при изменении или уничтожении информации, то говорят, что имеется опасность для целостности информации. Если ценность информации в ее оперативном использовании, то говорят, что имеется опасность нарушения доступности информации. Если ценность информации теряется при сбоях в системе, то говорят, что есть опасность потери устойчивости к ошибкам.

Каналы утечки информации:

1. Прямое хищение носителей и документов, обращающихся в процессе функционирования автоматизированных систем.

2. Запоминание или копирование информации, находящейся на машинных и на немашинных носителях.

3. Несанкционированное подключение к аппаратуре и линиям связи или незаконное использование "законной" (т. е. зарегистрированной) аппаратуры системы (чаще всего терминалов пользователей).

4. Несанкционированный доступ к информации за счет специального приспособления математического и программного обеспечения.

5. Перехват электромагнитных волн, излучаемых аппаратурой автоматизированных систем в процессе обработки информации.

Наблюдается большая разнородность целей и задач защиты - от обеспечения государственной безопасности до защиты интересов отдельных организаций, предприятий и частных лиц, дифференциация самой информации по степени ее уязвимости.

Можно выделить три направления работ по защите информации: теоретические исследования, разработка средств защиты и разработка способов использования средств защиты в автоматизированных системах.

В теоретическом плане основное внимание уделяется исследованию уязвимости информации в системах электронной обработки информации, выявлению и анализу каналов утечки информации, обоснованию принципов защиты информации в больших автоматизированных системах и разработке методик оценки надежности защиты

К настоящему времени разработано много различных средств, методов, мер и мероприятий, предназначенных для защиты информации, накапливаемой, хранимой и обрабатываемой в автоматизированных системах, в том числе и передаваемой по линиям связи большой протяженности. Сюда входят аппаратные и программные средства, криптографическое закрытие информации, физические меры, организационные мероприятия и законодательные меры.

Криптография как наука включает в себя несколько областей математики: теория чисел, теория информации, теория вероятности, абстрактная алгебра и формальный анализ. Криптографическое закрытие (шифрование) информации заключается в таком преобразовании защищаемой информации, при котором по внешнему виду нельзя определить содержание закрытых данных, т.е. это преобразование исходного текста в шифротекст и наоборот.

Общепринятая криптографическая система включает пять **компонентов**:

M – пространство исходных текстов;

C – пространство шифротекстов;

K – пространство ключей;

E_k – семейство шифрирующих преобразований – преобразований, зависящих от ключа k ($k \in K$), исходного текста m ($m \in M$) в шифротекст c ($c \in C$) $E_k(M)=C$;

D_k – семейство дешифрирующих преобразований – преобразований, зависящих от ключа k ($k \in K$), шифротекста (c, $c \in C$) в исходный текст (m, $m \in M$) $D_k(C)=M$.

Типы криптосистем:

симметричные (одноключевые);

асимметричные или с открытым ключом (двухключевые).

Безопасность в интерактивной среде – скорее миф, чем реальность. Все данные проходят через бесчисленное множество компьютеров, пока не прибудут по назначению. Единственное, что может обеспечить конфиденциальность, – это малая значимость ваших данных по сравнению с прочими электронными посланиями. Основная идея электронной почты – создать хакерам и любопытствующим максимум неудобств и отпугнуть их от себя.

В технологии электронной почты для защиты писем используются:

фальшивая почта;

анонимная почта;

шифрование.

В первых двух случаях делается попытка исключить выявление отправителя.

Фальшивая почта – это возможность посылать почту через Интернет, используя измененный адрес отправителя. Анонимный ремейлер – это почтовый сервер, разработанный для приема посылаемого ему сообщения, удаления всей информации в заголовке и замены ее анонимной информацией.

Если необходимо передать секретную информацию, то единственный способ защитить ее – это шифрование почты перед отправлением.

Появление Интернета как коммерческого средства изменило понятие коммерции. Развиваются принципиально новые финансовые инструменты. В настоящее время большинство взаиморасчетов по сети происходит путем дебетования кредитной карточки покупателя.

Проект электронной коммерции призван приспособить нынешние коммуникации банков и клиринговых контор к незащищенным общественным сетям (например Интернет) с целью предоставления новых услуг, таких как использование электронных чеков. При этом должно существовать три типа расчетов: оплата по предложению, оплата по требованию и кредитная оплата. Кредитная карточка с микропроцессором содержит информацию на интегральной микросхеме. Кроме информации о количестве наличных карточка содержит информацию о владельце и

способе доступа к данным, содержащимся на карточке. Для подписания юридических документов в сети применяется цифровая подпись, которая однозначно идентифицирует создателя послания. Для обеспечения безопасности при передаче финансовой информации по каналам Интернета применяются специальные протоколы защиты (например Secure Sockets Layer - SSL).

3. Методические рекомендации и варианты контрольных работ

3.1. Методические рекомендации

Контрольная работа состоит из трех заданий:

1. Подробно раскрыть содержание теоретических вопросов **А** и **Б** номера варианта с использованием текстового редактора.

2. Разработать иерархию классов и программную реализацию номера варианта задания для программной реализации, используя базовые возможности языка ООП, изложенные в настоящей работе, и текст программной реализации учебного примера, представленный ниже.

Задания выполняются в соответствии с методическими рекомендациями, приведенными перед вариантами каждого задания.

Номер варианта заданий определяется по последним двум цифрам зачетки в соответствии с заданной табл.6

Таблица 6

Последняя цифра номера зачетки	Варианты заданий по предпоследней цифре номера зачетки	
	Четная цифра	Нечетная цифра
1	2	3
0	1	11
1	2	12
2	3	13
3	4	14
4	5	15
5	6	16
6	7	17

Окончание табл. 6

1	2	3
7	8	18
8	9	19
9	10	20

Например, при номере зачетки 94146 надо выполнить задание варианта 7, при номере зачетки 94156 - задания варианта 17.

Контрольная работа должна быть выполнена в отдельной тетради, на титульном листе которой указываются название кафедры, наименование дисциплины, факультета, номер курса, группы, фамилия имя, отчество студента, номер зачетной книжки. В конце контрольной работы приводятся распечатка программной реализации, список используемой литературы, ставятся подпись студента и дата выполнения.

ОПИСАНИЕ УЧЕБНОГО ПРИМЕРА РАСЧЕТА ЗАРАБОТНОЙ ПЛАТЫ

Для успешного выполнения практического задания настоятельно рекомендуется глубоко разобраться в учебном примере, набрать и откомпилировать его, устранив ошибки при компиляции. При подготовке учебного примера использовалась инструментальная среда программирования Borland C++ version 3.1.

Задание. Разработать и реализовать программу расчета зарплаты учителя в зависимости от ставки и стипендии студента в зависимости от среднего балла. Оклады всех категорий сотрудников хранятся в текстовом файле на диске в формате:

<номер_должности> <название_должности> <оклад>

Например: файл Оклады.txt содержит следующие строки:
1 служащий 1200000
2 учитель 1400000
3 студент 500000
4 инженер 1420000
. . .

Необходимо считать содержимое файла через поток во внутреннюю таблицу.

Данные о конкретных учителях и студентах ввести с клавиатуры через поток, используя перегруженный оператор <<. Программа должна выводить на экран данные о сотрудниках (учителях и студентах), используя перегруженный оператор >> в следующем формате:

<Фамилия> <Зарплата> <Должность> [Дополнительная информация], где [Дополнительная информация] для учителя - его ставка, для студента – номер группы и средний балл.

Оклады для учителя и студента необходимо брать из таблицы окладов по номеру должности.

Создать иерархию из классов Служащий, Учитель и Студент. Использовать механизм позднего связывания для функции расчета зарплаты в каждом классе.

Учебный пример создавался с помощью системы программирования Borland C++ 3.1 для операционной системы DOS на языке C++. Для загрузки интегрированной среды программирования Borland C++ 3.1 необходимо запустить файл BC.EXE.

Описания классов, типов, глобальных переменных и прототипы функций удобно разместить в так называемом заголовочном файле, который обычно имеет расширение (.h), подключив его с помощью директивы *#include* к основному модулю.

Например:

```
//ZP.CPP                    - главный модуль
#include “zp.h”              //в это место будет вставлен файл zp.h
. . .
void main(void)             //главная функция - точка входа в программу
{
. . .
}
```

Определим классы в файле zp.h:

PayTable	**- таблица окладов сотрудников**
Man	**- служащий**
Teacher	**- учитель**
Student	**- студент**

Таблицу окладов можно представить в виде массива структур следующего содержания:

```
struct PayPosition
{
        int Num;                    //номер должности (например - 3)
        char Position[30];  //название должности (например - student)
        int Pay;                    //оклад (например - 500 000)
};
```

и методов заполнения таблицы из файла на диске и извлечения из таблицы окладов по номеру должности.

Таким образом, таблица окладов представляется как

```
class PayTable
{
  struct PayPosition PayPos[5]; //таблица из пяти строк и трех столбцов
 public:
  FillPayTable(char* Path);       //заполнение таблицы данными из текстового
файла
  long getPayByNum(int Num);  //получение оклада по номеру должности
};
```

Формирование иерархии можно начать с определения базового класса *Man.* Этот класс содержит имя служащего (поле *Name*), его зарплату (поле *Salary*), методы ввода и вывода данных о конкретном служащем и метод расчета зарплаты

```
class Man
{
  char Name[40];          //имя служащего как массив из 40 символов
 protected:                   //разрешение доступа из производных классов
  long Salary;                    //к переменной, содержащей зарплату
public:
//конструктор создает служащего с именем Empty
//и зарплатой, равной нулю
  Man() {strcpy(Name,"Empty"); Salary = 0;};
//перегрузка операторов >> и << для обеспечения ввода и вывода
//данных о служащем
```

```
  friend istream& operator>>(istream& stream, Man& M);
  friend ostream& operator<<(ostream& stream, Man M);
//метод рсчета зарплаты служащего
  virtual void calcSalary(PayTable& pay_table);
};
```

Конструктор для инициализации поля *Name* вызывает стандартную функцию копирования строк *strcpy,* которая объявлена в библиотеке *string.* Эту библиотеку необходимо подключить к файлу zp.h с помощью директивы *#include:*

```
//ZP.H
#include <string.h>
. . .
```

Перегруженные операторы << и >> объявлены как дружественные (*friend*) для поддержания стандартного синтаксиса при работе с этими операторами. Для ввода данных о конкретном служащем необходимо изменить содержимое объекта класса *Man*, поэтому в метод *operator>>* передается ссылка на объект M (*Man& M*).

В метод *calcSalary* передается ссылка на объект класса *PayTable* для обеспечения доступа к полям и методам класса *PayTable* из класса *Man,* в частности для доступа к методу *getPayByNum.* Для поддержания механизма позднего связывания метод *calcSalary* объявлен как виртуальный в базовом классе и затем будет переопределен в производных классах.

Пример производного класса:

```
class Student: public Man              //наследование от базового класса Man
{
  long Group;                    //номер группы студента
  float Mark;                //средний балл
 public:
//конструктор создает студента, инициализируя поля Name, Salary, Group и
Mark
  Student():Man() {Group = 0; Mark = 0.0;};
  friend istream& operator>>(istream& stream, Student& M);
  friend ostream& operator<<(ostream& stream, Student M);
//перекрытый метод расчета зарплаты студента
  void calcSalary(PayTable& pay_table);
```

```
};
```

Аналогичным образом описывается класс Учитель, только вместо номера группы и среднего балла он содержит информацию о ставке учителя.

Теперь можно описывать методы классов. Рассмотрим файл zp.cpp. Вначале идет подключение необходимых для работы программы библиотек:

```
#include <conio.h>              //для функции очистки экрана - clrscr
#include <iostream.h>           //для работы с потоком
#include <fstream.h>            //для вывода из файла через поток
#include <stdlib.h>             //для функции atol – преобразование
                                // строки в тип long
#include "zp.h"                 //подключение файла с описанием классов

//методы класса PayTable
//*****************************************************************
//* Class:        PayTable
//* Method:       getPayByNum
//* Params:       int Num – номер должности
//* Return:       long – оклад
//* Description: выбирает величину оклада из таблицы по номеру
// должности
//*****************************************************************
long PayTable::getPayByNum(int Num)
{
  return PayPos[Num-1].Pay;
};
. . .

//методы класса Man
//*****************************************************************
//* Class:        Man
//* Method:       calcSalary
//* Params:      PayTable& pay_table – ссылка на таблицу с окладами
```

```
//* Return:        void
//* Description: вычисляет зарплату служащего и сохраняет значение в
//*                поле Salary
//*****************************************************************
void Man::calcSalary(PayTable& pay_table)
{
  Salary = pay_table.getPayByNum(MAN);
// MAN-константа, номер должности - служащий
};

//методы класса Teacher
//*****************************************************************
//* Class:         Teacher
//* Method:        calcSalary
//* Params:       PayTable& pay_table – ссылка на таблицу с окладами
//* Return:        void
//* Description: вычисляет зарплату учителя и сохраняет значение в
//*                поле Salary
//*****************************************************************
void Teacher::calcSalary(PayTable& pay_table)
{
  Salary = Rate * pay_table.getPayByNum(TEACHER);
// зарплата учителя = ставка * оклад учителя
};


//методы класса Student
. . .

//глобальные функции
//операция ввода данных о служащем
istream& operator>>(istream& stream, Man& M)
{
  cout << endl << "Name: ";              //вывод на экран
```

```
  stream >> M.Name;                    //ввод имени служащего
  return stream;                       //возвращаем поток
};

//операция ввода данных об учителе
istream& operator>>(istream& stream, Teacher& M)
{
  cout << endl << "-- Teacher --";          //вывод на экран
  stream >> (Man&)M;                   //ввод имени
  cout << "Rate: ";
  stream >> M.Rate;                    //ввод ставки
  return stream;
};
. . .

//операция вывода данных о служащем
ostream& operator<<(ostream& stream, Man M)
{
  stream << endl << M.Name << '\t' << M.Salary;
  //вывод в поток имени и зарплаты
  return stream;
};

//операция вывода данных об учителе
ostream& operator<<(ostream& stream, Teacher M)
{
  stream << (Man)M;              //вывод в поток имени и зарплаты
  stream << '\t' << "teacher" << '\t' << M.Rate;
 //вывод дополнительной информации
  return stream;
};
. . .
void main(void)                        //точка входа в программу
{
```

```
  clrscr();                          //очистка экрана
  PayTable pay_table;                //объявление объекта класса PayTable
  if(pay_table.FillPayTable("f1.txt"))
//заполнение таблицы значениями из файла f1.txt
  {                                  //ошибки при заполнении не возникло
    Student stud[10];        //создание десяти объектов класса Student
    Teacher teach[2];        //создание двух экземпляров класса Teacher
    for(int i = 0; i < 10; i++)      //организация цикла
    {
      cin >> stud[i];                //заполнение информации о студентах
      stud[i].calcSalary(pay_table);
          //расчет зарплаты для каждого студента
    };
    for(i = 0; i < 2; i++)
    {
      cin >> teach[i];               //заполнение информации об учителях
      teach[i].calcSalary(pay_table);
          //расчет зарплаты для каждого учителя
    };
    for(i = 0; i < 2; i++)
      cout << stud[i];               //вывод информации о студентах
    for(i = 0; i < 2; i++)
      cout << teach[i];                    //вывод информации об учителях
  }
  else                               //при работе с файлом произошла ошибка
  {
   cout << "Error working with file!";   //вывод сообщения об ошибке
  };
}
```


3.2. Теоретические вопросы

1. **А.** Понятие компьютерной сети. Глобальные и локальные сети.
   Б. Разработка информационных моделей (методы, описание процесса, средства).

2. **А.** Модели компьютерных сетей. Основные виды топологий.
   Б. Разработка модели состояний (методы, описание процесса, средства).

3. **А.** Основные компоненты компьютерной сети. Понятие протокола.
   Б. Разработка моделей процессов (методы, описание, средства).

4. **А.** Межсетевые связи. Понятие компьютерного моста, шлюза и модема.
   Б. Отношения между классами.

5. **А.** Семиуровневая модель ISO/OSI.
   Б. Классификация и тенденции развития современных информационных технологий.

6. **А.** Сетевой уровень семиуровневой модели OSI. Маршрутизация данных. Управление трафиком.
   Б. Понятие объектно-ориентированного анализа (классы, объекты). Основные концептуальные элементы объектно-ориентированного подхода.

7. **А.** Транспортный уровень семиуровневой модели OSI. Виртуальные соединения на транспортном уровне.
   Б. Модели (логическая, физическая, статическая и динамическая) в ООА.

8. **А.** Понятие сети Интернет, Интранет и World Wide Web.
   Б. Классификация. Методы классификации.

9. **А.** Основные сервисы Интернета.
   Б. Процесс объектно-ориентированного проектирования. Объектно-ориентированное проектирование в жизненном цикле разработки программного обеспечения.

10. **А.** Протоколы Интернета (TCP/IP, HTTP).

Б. Сетевая модель представления знаний.

11. **А.** Электронная почта.

Б. Представление знаний при помощи фреймов.

12. **А.** Реляционная модель данных. Основные понятия, термины, определения.

Б. Понятие информационной безопасности. Угрозы сохранности информации. Каналы утечки информации.

13. **А.** Суть и описание этапов концептуального проектирования БД.

Б. Цели и задачи защиты информации. Средства, методы и меры обеспечения информационной безопасности.

14. **А.** Суть и описание этапов логического проектирования БД.

Б. Криптография. Компоненты криптосистемы.

15. **А.** Суть нормализации отношений, приведение отношения к третьей нормальной форме.

Б. Типы криптосистем.

16. **А.** Моделирование информации в терминах сущность-связь. Создание модели базы данных.

Б. Машина вывода ЭС. Суть и алгоритм обратного вывода. Аспекты использования.

17. **А.** Различия в реализациях архитектуры “клиент-сервер” (· модель файлового сервера (File Server - FS); модель доступа к удаленным данным (Remote Data Access - RDA).

Б. Безопасность в Интернете (хакеры, взломщики, фрикеры и спаммеры).

18. **А.** Обработка распределенных данных.

Б. Структура и функции экспертной системы (ЭС).

19. **А.** Восстановление БД после сбоев. Средства восстановления.

Б. Системы искусственного интеллекта. Понятие базы знаний. Основные модели представления знаний.

20. А. Языки манипуляции данными. Язык SQL

Б. Безопасность электронной коммерции.

3.3. Задания для программной реализации

Чтение/запись файлов и ввод/вывод на экран выполнять через поток, используя перегрузку операторов << и >>. Создать иерархию как минимум из двух классов. Использовать механизм позднего связывания.

1. Разработать программу расчета зарплаты сотрудников фирмы.

Исходными данными являются:

текстовой файл Сотрудники со следующей обязательной информацией:

<Фамилия сотрудника> <Должность> <Разряд> <Ставка> .

Сформировать классы по должностям;

текстовой файл Оклады в формате

<Разряд> <Оклад>

Занести в таблицу.

Выходные данные:

вывод на экран информации о сотрудниках в следующей форме:

<Фамилия> <Зарплата>

2. Разработать программу расчета объема реализованной продукции в магазине за рабочую неделю.

Исходные данные:

текстовые файлы Кондитерский отдел, Бакалейный отдел и Молочный отдел со следующей обязательной информацией:

<Номер дня недели> <Объем продаж>

Сформировать классы Отдел, Кондитерский, Бакалейный и Молочный.

Выходные данные:

вывод на экран таблицы в формате:

<Название отдела> <Объем продаж за неделю>

Итого

3. Разработать программу расчета потребности в оборотных средствах.

Исходные данные:

текстовой файл Тип оборудования в формате:

<Тип оборудования> <Типовой норматив запчастей на ед. оборудования> <Коэффициент понижения>

Занести в таблицу;

ввести с клавиатуры некоторое количество оборудования разного типа со следующей обязательной информацией:

<Наименование оборудования> <Тип оборудования>

Сформировать классы по типам оборудования

Выходные данные:

текстовой файл Норматив в формате

<Тип оборудования> <Количество однотипного оборудования> <Потребность в оборотных средствах>

Итого

4. Разработать программу расчета потребности в спецодежде и обуви для двух отделов.

Исходные данные:

текстовой файл Спецодежда в формате:

<Наименование спецодежды> <Количество выдач в год> <Стоимость единицы>

Занести в таблицу;

ввести с клавиатуры потребности в спецодежде по каждому отделу со следующей обязательной информацией:

<Наименование спецодежды> <Количество работников>

Сформировать классы Отдел1 и Отдел2.

Выходные данные:

текстовой файл Потребность в формате

<Отдел> <Наименование спецодежды> <Количество> <Сумма>

Итого

5. Разработать программу расчета выручки от реализации продукции.

Исходные данные:

текстовой файл Стоимость изделия в формате:

<Тип изделия> <Наименование типа изделия> <Стоимость единицы изделия>

Занести в таблицу;

ввести с клавиатуры несколько экземпляров изделий определенных типов со следующей обязательной информацией:

<Наименование изделия> <Остатки на начало года> <План выпуска> <Остатки на конец года>

Сформировать классы по типу изделия. В базовом классе ввести виртуальную функцию расчета объема реализации в шт. = план выпуска–остатки на начало года+остатки на конец года

Выходные данные:

вывод на экран таблицы в формате:

<Наименование изделия> <Наименование типа изделия> <Объем реализации> <Стоимость объема реализации>

6. Разработать программу расчета начислений по двум отделам фирмы

Исходные данные:

текстовой файл Начисления в формате:

<Табельный номер сотрудника> <Зарплата> <Премии> <Прочие начисления>

Занести в таблицу;

ввести с клавиатуры по несколько сотрудников из Отдела1 и из Отдела2 со следующей обязательной информацией:

<Табельный номер сотрудника> <Фамилия> <Должность>.

Сформировать классы Отдел1, Отдел2 и базовый класс. В базовом классе ввести виртуальную функцию расчета начислений.

Выходные данные:

вывод на экран двух таблиц по отделам в формате:

<Фамилия> <Должность> <Начисления>;

вывод на экран общей суммы начислений по отделам.

7. Разработать программу расчета времени по норме и суммы выработки изделий.

Исходные данные:

текстовой файл Изделия в формате:

<Тип изделия> <Норма времени на единицу изделия> <Расценка на единицу>

Занести в таблицу;

ввести с клавиатуры несколько экземпляров годных изделий определенных типов со следующей обязательной информацией:

<Наименование изделия> <Характеристики>

Сформировать классы по типам изделий. Ввести виртуальные функции расчета времени по норме и расчета суммы.

Выходные данные:

вывод на экран таблицы Время по норме в формате:

<Тип изделия> <Количество годных изделий> <Время по норме>

вывод на экран таблицы Сумма в формате:

<Тип изделия> <Количество годных изделий> <Сумма>

8. Разработать программу расчета поступления товаров на склад.

Исходные данные:

текстовой файл Предприятия-поставщики в формате:

<Код предприятия> <Наименование>

Занести в таблицу;

текстовой файл Товары в формате:

<Код товара> <Наименование>

Занести в таблицу;

ввести с клавиатуры данные для двух складов со следующей обязательной информацией:

<Код поставщика> <Код товара> <Должно быть поставлено по договору>

<Фактически поступило>

Сформировать классы Склад1, Склад2 и базовый класс. В базовом классе ввести виртуальную функцию расчета отклонения по сумме.

Выходные данные:

вывод на экран таблицы Ведомость в формате:

<Наименование поставщика> <Наименование товара> <Отклонение>

9. Разработать программу расчета стоимости незавершенного производства.

Исходные данные:

текстовой файл Детали в формате:

<Номер детали> <Наименование> <Плановая стоимость за единицу>

<Нормативная стоимость за единицу>

Занести в таблицу;

ввести с клавиатуры данные о деталях для двух цехов со следующей обязательной информацией:

<Номер детали> <Количество деталей>

Сформировать классы Цех1, Цех2 и базовый класс. В базовом классе ввести виртуальные функции расчета нормативной и плановой сумм.

Выходные данные:

вывод на экран таблицы Сведения о стоимости незавершенного производства по каждому цеху в формате:

<Наименование детали> <Количество деталей> <Плановая стоимость>

<Нормативная стоимость>

10. Разработать программу расчета зарплаты рабочих.

Исходные данные:

текстовой файл Зарплата в формате:

<Разряд> <Зарплата>

Занести в таблицу;

ввести с клавиатуры данные о рабочих для двух цехов со следующей обязательной информацией:

<Фамилия> <Разряд> <Премия>

Сформировать классы Цех1, Цех2 и базовый класс. В базовом классе ввести виртуальную функцию расчета зарплаты рабочих.

Выходные данные:

вывод на экран таблицы Ведомость начисления зарплаты рабочим по каждому цеху в формате

<Фамилия> <Сумма к начислению>

11. Разработать программу расчета нормо-часов и зарплаты рабочих-сдельщиков.

Исходные данные:

текстовой файл Детали в формате:

<Номер детали> <Нормир. время за единицу> <Расценка за единицу>

Занести в таблицу;

ввести с клавиатуры данные о рабочих для двух цехов со следующей обязательной информацией:

<Фамилия> <Номер детали> <Количество принятых деталей>

Сформировать классы Цех1, Цех2 и базовый класс. В базовом классе ввести виртуальные функции расчета нормо-часов и зарплаты рабочих.

Выходные данные:

вывод на экран таблицы Сведения о выработке рабочих-сдельщиков по каждому цеху в формате:

<Фамилия> <Нормочасы> <Зарплата>

12. Разработать программу расчета себестоимости продукции двух цехов.

Исходные данные:

текстовой файл Изделия в формате:

<Код изделия> <Наименование> <Плановая себестоимость> <Фактическая себестоимость>

Занести в таблицу;

ввести с клавиатуры данные о количестве выпущенной продукции для двух цехов со следующей обязательной информацией:

<Код изделия> <Количество>

Сформировать классы Цех1, Цех2 и базовый класс. В базовом классе ввести виртуальную функцию расчета себестоимости.

Выходные данные:

вывод на экран таблицы Ведомость определения себестоимости продукции по каждому цеху в формате:

<Код изделия> <Наименование> <Количество выпущенной продукции> <Себестоимость>

13. Разработать программу расчета суммы естественной убыли.

Исходные данные:

текстовой файл Норма естественной убыли в формате:

<Период времени> <Наименование культуры> <Норма в %>

Занести в таблицу;

ввести с клавиатуры данные о количестве выпущенной продукции для двух культур со следующей обязательной информацией:

<Период времени> <Средний остаток> <Розничная цена за 1 кг>

Сформировать классы Картофель, Капуста и базовый класс. В базовом классе ввести виртуальную функцию расчета суммы естественной убыли = Средний остаток * Норма * Розничная цена.

Выходные данные:

вывод на экран таблицы Потеря вследствие естественной убыли по каждой культуре в формате:

<Период времени> <Культура> <Сумма убыли>

14. Разработать программу расчета суммы по накладной.

Исходные данные:

текстовой файл Товары в формате:

<Номенклатурный номер> <Наименование> <Цена за ед.> <Затребованное количество>

Занести в таблицу;

ввести с клавиатуры данные об отпущенном товаре для двух магазинов со следующей обязательной информацией:

<Номер товара> <Отпущено>

Сформировать классы Магазин1, Магазин2 и базовый класс. В базовом классе ввести виртуальную функцию расчета суммы по накладной = отпущенное количество товара * цена за ед.

Выходные данные:

вывод на экран Накладной для каждого магазина в формате:

<Наименование товара> <Затребованное количество> <Отпущенное количество> <Цена> <Сумма>

15. Разработать программу расчета процента выполнения плана за квартал для двух цехов

Исходные данные:

текстовой файл План в формате:

<Месяц> <Цех> <Выход продукции по плану>

Занести в таблицу;

ввести с клавиатуры данные о фактическом количестве выпущенной продукции для двух цехов со следующей обязательной информацией:

<Месяц> <Фактический выход продукции>

Сформировать классы Цех1, Цех2 и базовый класс. В базовом классе ввести виртуальную функцию расчета процента выполнения плана = фактический / по плану * 100.

Выходные данные:

вывод в файлы Цех1 и Цех2 следующей информации:

<Месяц> <% выполнения плана>

16. Разработать программу распределения сельскохозяйственных угодий РБ.

Исходные данные:

ввести с клавиатуры данные о конкретных землепользователях (например с. Озерный, с. Вишневка, ч.п. Сергеев) в формате:

<Пашни> <Сенокосы> <Пастбища> <Прочие угодья>,

Сформировать классы Совхозы, Колхозы, Частники и базовый класс. В базовом классе ввести виртуальную функцию расчета общей площади с-х. угодий.

Выходные данные:

необходимо просуммировать по каждой категории землепользователей площади пашен, пастбищ, сенокосов и прочих угодий и затем вывести в файл следующую информацию:

<Категория землепользователей> <Пашни> <Сенокосы> <Пастбища> <Прочие угодья> <Общая площадь>

17. Разработать программу расчета зарплаты работников предприятия.

Исходные данные:

текстовой файл Сверхурочники в формате:

<Табельный номер> <Сумма начисления за сверхурочную работу>

Занести в таблицу;

ввести с клавиатуры данные о начальниках и подчиненных предприятия со следующей обязательной информацией:

<Табельный номер> <Фамилия> <Начисления за повременную работу>

<Начисления за сдельную работу> <Сумма удержания>

Сформировать классы Начальники, Подчиненные и базовый класс. В базовом классе ввести виртуальную функцию расчета зарплаты = начисления за повременную работу + начисления за сдельную работу + начисления за сверхурочную работу (если есть) - сумма удержания.

Выходные данные:

вывести на экран Расчетно-платежные ведомости для начальников и подчиненных в формате:

<Табельный номер> <Фамилия> <Сумма начисления за повременную работу> < Сумма начисления за сдельную работу > <Сумма начисления за сверхурочную работу > <Общая сумма начисления> <Сумма удержания> <Сумма к выдаче>.

18. Разработать программу расчета суммы удержания работников предприятия.

Исходные данные:

текстовой файл Штрафники в формате:

<Табельный номер> <Сумма штрафа>

Занести в таблицу;

ввести с клавиатуры данные о начальниках и подчиненных предприятия со следующей обязательной информацией:

<Табельный номер> <Фамилия> <Аванс> <Подоходный налог>

Сформировать классы Начальники, Подчиненные и базовый класс. В базовом классе ввести виртуальную функцию расчета суммы удержания = аванс + налог+ +штраф (если есть).

Выходные данные:

вывести в файлы Ведомости удержания для начальников и подчиненных в формате

<Табельный номер> <Фамилия> <Аванс> < Подоходный налог> <Штраф> <Всего удержано>.

19. Разработать программу расчета коэффициента сортности в планируемом году и сравнения его с коэффициентом сортности в отчетном году.

Исходные данные:

текстовой файл Отчетный год в формате:

<Сорт> <Коэффициент сортности>

Занести в таблицу;

ввести с клавиатуры данные о сортах со следующей обязательной информацией:

<Удельный вес в общем объеме продукции> <Соотношение цен>

Сформировать классы Сорт1, Сорт2, Сорт3 и базовый класс. В базовом классе ввести виртуальную функцию расчета коэффициента сортности = удельный вес в общем объеме продукции * соотношение цен / 100. Объявить по одному экземпляру классов СортN.

Выходные данные:

вывести на экран сравнительную таблицу в формате

<Сорт> <Коэффициент сортности за отчетный год> <Коэффициент сортности за планируемый год> <Отклонение>.

20. Разработать программу учета оборудования фирмы по продаже оргтехники.

Исходные данные:

текстовой файл Гарантия в формате:

<Тип оргтехники> <Гарантийный срок>

Занести в таблицу;

ввести с клавиатуры данные об оргтехнике со следующей обязательной информацией:

<Номер товара> <Наименование> <Дата продажи>

Сформировать классы по типу оргтехники (например, Компьютеры, Принтеры) и базовый класс. В базовом классе ввести виртуальную функцию расчета даты окончания срока гарантии = дата продажи + гарантийный срок.

Выходные данные:

вывести на экран таблицы по компьютерам и принтерам, которые подлежат гарантийному обслуживанию на текущий день в формате:

<Номер товара> <Наименование> <Дата окончания гарантийного срока >

<Отклонение>

Литература

1. Евдокимов В.В. и др. Экономическая информатика. СПб.:Питер, 1996.
2. Шлеер С., Меллор С. Объектно-ориентированный анализ: моделирование мира в состояниях. Киев: Диалектика, 1993.
3. Буч Г. Объектно-ориентированное проектирование с примерами применения: Пер. с англ. – М.: Конкорд, 1992.
4. Скляров В. А. Язык С++ и объектно-ориентированное программирование: Справ. издание. Мн.: Выш. шк, 1997.
5. Пол И. Объектно-ориентрованное программирование с использованием С++. -Киев: ДиаСофт Лтд, 1995.
6. Неформальное введение в С++ и TURBO VISION. СПб: Галерея "ПЕТРОПОЛЬ", 1992.
7. Веттинг Д. Novell NetWare: Пер. с нем. Киев: Торгово-издательское бюро BHV, 1994.
8. Охрименко С.А. Новые информационные технологии. (Локальные сети персональных компьютеров): Обзор. информ. / Молд. НИИЦЭИ. – Кишинев, 1991.
9. Кликушин А.Г., Дегтярь Б.Б. Введение в локальные сети. М.: ПНФ МикроКомп ООО, 1991.
10. Домин Ф.А. Принципы организации вычислительных сетей. Киев: УМКВО, 1991.
11. Вычислительные машины, системы и сети / Под ред А.П. Пятибратова. - М.: Финансы и статистика, 1991 .
12. Алан Франк. Сервисы Internet и World Wide Web. M. Lan Magazine/Рус. Издание. Сент. 1996. Т. 2. №5.
13. Алан Франк. Серия публикаций по ВС. Lan Magazine/Рус. Издание. 1996. №1-8.
14. Гаффин Адам. Путеводитель по глобальной компьютерной сети Internet. М.: ТПП “Сфера”, 1995.
15. Гилстер П. Навигатор Internet // Путеводитель для человека с компьютером и модемом: Пер. с англ. М.: Джон Уайли энд Санз, 1995.
16. Джамса К., Лалани С., Уикли С. Программирование в Web для профессионалов. Мн.: ООО “Попури”, 1997
17. Нанс Б. Компьютерные сети. М.: БИНОМ, 1995.
18. Мейер Д. Теория реляционных баз данных. М.: Мир, 1987.

19. Хабард Дж. Автоматизированное проектирование баз данных: Пер. с англ.М.: Мир. 1984
20. Айтьян С.Х. Базы данных для персональных ЭВМ. М., 1988.
21. Грабер Мартин. Введение в SQL. М., 1996.
22. Лодыженский Г.М. Системы управления базами данных – коротко о главном. М.: СУБД, 1995. № 1-4.
23. Калянов Г.Н. Case структурный системный анализ (автоматизация и применение). М., Изд-во “Лори”, 1996.
24. Кузнецов С.Д. Системы управления базами данных. М.: СУБД, 1995. № 1-6.
25. Sybase SQL Anywhere. A System 11 Server Product User’s Guide Volume 1: Using SQL Anywhere. Copyright, 1995, By Sybase, Inc. USA.
26. Когаловский М.Р. Технология баз данных на ПЭВМ. М.: Финансы и статистика, 1992.
27. Искусственный интеллект / Под ред. Д.А. Поспелова. В 3 кн. М.: Радио и связь, 1990.
28. Осуга С. Обработка знаний: Пер. с япон.М.:Мир,1989.
29. Уэно Х., Кояма Т., Окамото Т. и др. Представление и использование знаний / Под ред. Х.Уэно, М.Исидзука; Пер. с япон. - М.: Мир, 1989.
30. Одинцов Б.Е.. Проектирование экономических экспертных систем. М.: Компьютер. Изд. объединение “ЮНИТИ”, 1996.
31. Эддоу М. Методы принятия решений. М.: Финансы и статистика, 1997.
32. Леонов А.П., Леонов К.А., Фролов Г.В. Безопасность автоматизированных банковских и офисных систем. Мн.: Национальная книжная палата,1996.
33. Стенг Д. Секреты безопасности сетей. Киев. 1996.
34. Тайли Э. Безопасность компьютера. Мн.: ООО “Попурри”, 1997.
35. Галатенко В. А. Информационная безопасность – основы //СУБД. 1996. №1, С. 6-28.
36. Жельников В. В. Криптография от папируса до компьютера. М.: ABF, 1997.
37. Мельников В. В. Защита информации в компьютерных системах. М.: Финансы и статистика, 1997.
38. Ярочкин В. И. Безопасность информационных систем. М.: Ось-89, 1996.

Св. план 1999, поз. 94

Учебное издание

СОВРЕМЕННЫЕ ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

Методические указания, программа и контрольные задания
по курсу
“Современные информационные технологии”
для студентов экономических специальностей
заочной формы обучения

Составители: Бахирев Андрей Владимирович,
Комличенко Виталий Николаевич,
Синицын Александр Георгиевич

Редактор Т.А. Лейко
Корректор Е.Н. Батурчик

Подписано в печать Формат 60x84 1/16.
Бумага Печать офсетная. Усл. печ. л.
Уч.-изд. л. 4,6. Тираж 120 экз. Заказ

Белорусский государственный университет информатики и радиоэлектроники
Отпечатано в БГУИР. Лицензия ЛП №156. 220027, Минск, П. Бровки, 6