

Министерство образования и науки Украины
Донбасская государственная машиностроительная академия

ПРОГРАММИРОВАНИЕ И АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ

КОНСПЕКТ ЛЕКЦИЙ

(для студентов направления «Системный анализ»)

Утверждено
на заседании кафедры ИСПР
Протокол № 2 от 9 сентября 2014г.

Краматорск 2014

УДК 681.3

Программирование и алгоритмические языки : конспект лекций (для студентов направления «Системный анализ» всех форм обучения). Часть 2 / Сост. А.Ю. Мельников – Краматорск : ДГМА, 2014. – 33 с.

Приведены лекционные материалы к курсу.

Составитель	Мельников А.Ю., канд. техн. наук, доцент
-------------	--

Отв. за выпуск	Мельников А.Ю., канд. техн. наук, доцент
----------------	--

СОДЕРЖАНИЕ

1 Основные понятия объектно-ориентированного подхода.....	4
1.1 Объектная модель.....	5
1.2 Классы и объекты.....	8
1.3 Классификация.....	12
2 Объектно-ориентированное программирование в среде C++.....	15
2.1. Класс как абстрактный тип данных.....	15
2.1.1 Класс как расширение понятия структуры. Статические компоненты класса.....	15
2.1.2 Конструкторы, деструкторы и доступность компонент...	18
2.1.3 Компонентные данные.....	21
2.1.4 Компонентные функции.....	22
2.1.5 Указатель this.....	24
2.1.6 Друзья классов.....	24
2.1.7 Расширение действия (перегрузка) стандартных опера- ций.....	25
2.2. Наследование и другие возможности классов.....	26
2.2.1 Наследование классов.....	26
2.2.2 Множественное наследование и виртуальные базовые классы.....	28
2.2.3 Виртуальные функции и абстрактные классы.....	30
2.2.4 Локальные классы.....	31
Список рекомендованной литературы.....	31

1 ОСНОВНЫЕ ПОНЯТИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДА

Все методы проектирования программных систем можно объединить в три группы:

- структурное проектирование сверху вниз, в основе которого лежит алгоритмическая декомпозиция и которое не может обеспечить создание сложных систем и неэффективно в объектно-ориентированных языках;
- метод потоков данных, когда система рассматривается как преобразователь входных потоков в выходные;
- объектно-ориентированное проектирование.

Объектно-ориентированное программирование (ООР – Object-oriented programming) – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования. Следует различать «объектные» («объектно-базированные») и «объектно-ориентированные» языки программирования.

Объектно-ориентированное проектирование (ООД – Object-oriented design) – это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

Объектно-ориентированный анализ (ООА – Object-oriented analysis) – это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

В результате объектно-ориентированного анализа формируются модели, на которых основывается объектно-ориентированное проектирование, создающее фундамент для окончательной реализации системы с использованием методологии объектно-ориентированного программирования. Все три методологии базируются на понятиях *объектной модели*.

1.1 Объектная модель

Объектная модель является концептуальной базой объектно-ориентированного стиля. Она состоит из четырех главных элементов (абстрагирование, инкапсуляция, модульность, иерархия) и трех дополнительных (типизация, параллелизм, сохраняемость), которые полезны, но не обязательны.

Абстрагирование выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов, и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя. Главный принцип абстрагирования – принцип наименьшего удивления, согласно которому абстракция должна охватывать все поведение объекта, но не приносить сюрпризов или побочных эффектов, лежащих вне ее сферы применимости.

Выбор правильного набора абстракций – самая главная задача.

Абстракции можно объединить в 4 группы.

Абстракция сущности: объект представляет собой полезную модель некой сущности в предметной области.

Абстракция поведения: объект состоит из обобщенного множества операций.

Абстракция виртуальной машины: объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня.

Произвольная абстракция: объект включает в себя набор операций, не имеющих друг с другом ничего общего.

Примеры абстракций в гидропонном хозяйстве: теплица, температура, влажность, освещение, кислотность, датчики, культуры, план выращивания.

Инкапсуляция – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Абстрагирование и инкапсуляция дополняют друг друга: первое направлено на наблюдаемое поведение объекта, а вторая занимается его внутренним устройством. *Интерфейс* отражает внешнее поведение объекта, описывая абстракцию поведения всех объектов данного класса; вну-

тренняя *реализация* описывает представление этой абстракции и механизмы достижения желаемого поведения объекта.

(Следует обратить внимание: «Инкапсуляция не спасает от глупости; она защищает от ошибок, но не от жульничества» – Б. Страуструп.)

Пример: простой нагреватель имеет расположение (конструктор), включатель, выключатель и проверку состояния. Усложненный нагреватель может быть связан с датчиком температуры и планом выращивания.

Модульность – это свойство системы разлагаться на внутренне связанные, но слабо связанные между собой модули. Логическое продолжение инкапсуляции, ее «практическая проекция».

Иерархия – это упорядочение абстракций, расположение их по уровням. Различают два вида иерархических структур: структура классов («is-a») и структура объектов («part of»).

Структура классов представляет иерархию наследования типа «обобщение-специализация», в которой подкласс (потомок) представляет собой специализированный частный случай своего суперкласса (предка). Пример: «токарь есть частный случай рабочего», «датчик температуры есть частный случай датчика».

Структура объектов представляет иерархию типа «агрегация», в которой один объект содержится внутри другого. Пример: «окно содержит полосы прокрутки, меню, строку состояния», «управляющий элемент в теплице содержит датчики температуры, влажности, кислотности и освещения».

Типизация – это способ защититься от использования объектов одного класса вместо другого, или, по крайней мере, управлять таким использованием. Типизация заставляет выражать абстракции так, чтобы язык программирования поддержал соблюдение принятых проектных решений. Различают сильную и слабую типизации.

Параллелизм – это свойство, отличающее активные объекты от пассивных. Предполагает параллельную (независимую) работу ряда активных объектов. При проектировании надо принимать меры, гарантирующие, что объект не будет растерзан на части несколькими независимыми процессами.

Сохраняемость – способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего первоначального адресного пространства.

Спектр сохраняемости объектов охватывает:

- промежуточные результаты вычисления выражений;
- локальные переменные в подпрограммах;
- собственные (глобальные) переменные и динамически создаваемые данные;
- данные, сохраняющиеся между сеансами выполнения программы;
- данные, сохраняемые при переходе на новую версию программы;
- данные, которые вообще переживают программу.

Традиционно, первыми тремя уровнями занимаются языки программирования, а последними – базы данных. Возможны смешанные решения: программисты разрабатывают специальные схемы для сохранения объектов в период между запусками программы, а конструкторы баз данных переиначивают свою технологию под короткоживущие объекты.

Существуют так называемые объектно-ориентированные базы данных (OODB), которые хранят не только данные, но и классы. Эта технология не прижилась, на практике предпочитают создавать объектно-ориентированную оболочку для реляционных баз данных.

Преимущества объектной модели

- 1 Позволяет в полной мере использовать возможности объектных и объектно-ориентированных языков программирования.
- 2 Повышает уровень унификации разработки и пригодность для повторного использования не только программ, но и проектов. Использование предыдущих разработок дает выигрыш в стоимости и времени.
- 3 Упрощает процесс внесения изменений из-за наличия стабильных промежуточных описаний. Это дает возможность не перерабатывать систему целиком даже в случае существенных изменений исходных требований.
- 4 Уменьшает вероятности допущения разнообразных ошибок.
- 5 Ориентирована на человеческое восприятие мира.

Некоторые факты из истории

1969 – Кэй (Kay) в своей диссертации предложил идею объектного подхода;

1979 – Джонс (Jones) ввел концепцию объектного подхода;

1981 – Буч (Booch) предложил методы OOD;

1988 – Шлеер и Меллор (Shlaer and Mellor) предложили методы OOA;

1988 – Страуструп (Stroustrup) опубликовал методическую статью по OOP;

1991 – Румбах (Rumbaugh) объединил OOA и OOD.

Организации, отвечающие за стандарты по объектной технологии: Object Management Group (OMG) и комитет ANSI X3J7.

1.2 Классы и объекты

Одними их базовых понятий объектной модели являются понятия классов и объектов.

Объект моделирует часть окружающей действительности и, таким образом, существует во времени и пространстве. Объект обладает состоянием, поведением и идентичностью; структура и поведение схожих объектов определяют общий для них класс; термины «экземпляр класса» и «объект» взаимозаменяемы.

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств. Пример: автомат по продаже напитков.

Поведение определяет совокупность действий и реакций объекта; выражается в терминах состояния объекта и передачи сообщений. Иными словами, поведение объекта – это наблюдаемая и проверяемая извне деятельность. Состояние объекта представляет суммарный результат его поведения.

Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию; это услуга, которую класс может предоставить своим клиентам. Типичный клиент может совершать операции 5 видов:

- 1 *Модификатор* – операция изменения состояния объекта.
- 2 *Селектор* – операция, считывающая состояние объекта, но не изменяющая его.
- 3 *Итератор* – операция, позволяющая организовать доступ ко всем частям объекта в строго определенной последовательности.
- 4 *Конструктор* – операция создания объекта и/или его инициализации.
- 5 *Деструктор* – операция очищения и/или разрушения объекта.

Совокупность всех методов и свободных процедур, относящихся к конкретному объекту, образует *протокол* этого объекта.

Объекты могут быть активными и пассивными. Активный объект может проявлять свое поведение без воздействия со стороны других объектов. Пассивный объект, напротив, может изменять свое состояние только под воздействием других объектов. Таким образом, активные объекты системы – источники управляющих воздействий.

Идентичность – это такое свойство объекта, которое отличает его от всех других объектов. Следует уметь отличать имя объекта от самого объекта. Если абстракция представляет собой собрание свойств без собственного поведения, это не объект, а структура.

Система реализуется только в процессе взаимодействия объектов. Отношения между объектами могут быть двух типов: связи и агрегация.

Связь – это специфическое сопоставление, через которое клиент запрашивает услугу у объекта-сервера или через которое один объект находит путь к другому.

Участвуя в связи, объект может выполнять одну из трех ролей:

- **актер**: может воздействовать на другие объекты, но сам никогда не подвергается воздействию других объектов (исключительно активный объект);
- **сервер**: может только подвергаться воздействию со стороны других объектов, но сам никогда не выступает в роли воздействующего объекта (исключительно пассивный объект);
- **агент**: может выступать как в активной, так и в пассивной роли.

На рис.1 представлены четыре объекта, каждый из которых характеризует описанные выше роли.

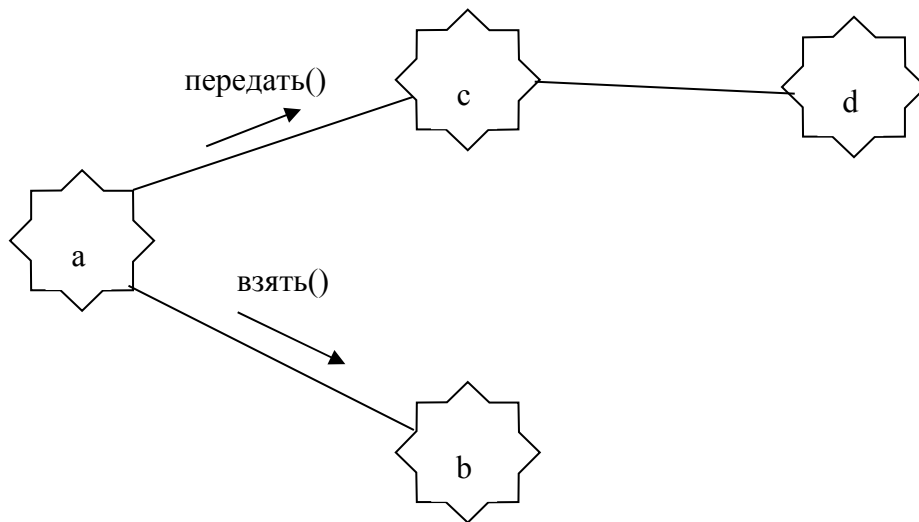


Рисунок 1 – Взаимодействующие объекты

Агрегация – специализированный частный случай ассоциации; описывает отношения целого и части, приводящие к соответствующей иерархии объектов.

Агрегация может означать физическое вхождение одного объекта в другой (автомобиль и двигатель), но не обязательно (акционер и акции). Пример агрегации с физическим вхождением представлен на рис. 2.



Рисунок 2 – Агрегация

Класс – это некое множество объектов, имеющих общую структуру и общее поведение. В то время как объект обозначает конкретную сущность, класс определяет лишь абстракцию существенного в объекте. Любой конкретный объект является просто экземпляром класса.

Поддерживаются 6 (шесть) видов отношений между классами.

1 **Ассоциация** – смысловая связь (по умолчанию – двусторонняя), фиксирующая участников, их роли и мощность отношения (1:1, 1:*, *:*):

* 1

Товар ————— Сделка

2 **Наследование** – такое отношение между классами, когда один класс повторяет структуру и поведение другого или других классов. Устанавливает отношение общего и частного. Классы, экземпляры которых создаются, называются конкретными, не создаются – абстрактными. Выводы: у любого класса может быть два вида клиентов – экземпляры (объекты) и подклассы (наследники).

Изображается в виде стрелки, направленной от потомка к предку:

Датчик ← Датчик температуры

3 **Агрегация** между классами имеет тот же смысл, что и в случае объектов. Различают агрегацию *по значению* (физическое включение) и *по ссылке*. Если есть сомнения в выборе между наследованием и агрегацией, лучше выбрать вторую.

Изображается в виде линии с закрашенным кругом:

Контроллер ●———— Нагреватель

4 Отношение **использования** между классами соответствует связи между их экземплярами (клиент-серверные отношения).

Изображается в виде линии с пустым кругом:

Контроллер ○———— План выращивания

5 **Инстанцирование** предполагает использование параметризованных классов, когда в качестве формального параметра классу передается указатель на какой-либо класс, а при обращении к его экземпляру формальный параметр замещается фактическим. Является продолжением (или частным случаем) отношения использования.

Изображается в виде прямоугольника-врезки в правом верхнем углу класса.

6 **Метакласс** – это класс, экземпляры которого являются классами. Изображается в виде закрашенного класса.

На этапе анализа и ранних стадиях проектирования решаются две основные задачи:

- выявление основных классов и объектов, составляющих словарь предметной области;
- построение структур, обеспечивающих взаимодействие объектов, при котором выполняются требования задачи.

Для оценки качества классов и объектов, выделяемых в системе, можно предложить следующие пять критериев: зацепление, связность, достаточность, полнота и примитивность.

Зацепление определяет степень глубины связей между отдельными модулями (чем меньше, тем лучше). Пример неправильного подхода: источник питания стереосистемы размещен в одной из колонок.

Связность – это степень взаимодействия между элементами отдельного модуля (класса, объекта), характеристика его насыщенности: «Класс Dog будет функционально связным, если он описывает поведение собаки, всей собаки и ничего, кроме собаки».

Достаточность – наличие в классе или модуле всего необходимого для реализации логичного и эффективного поведения.

Под **полнотой** подразумевается наличие в интерфейсной части класса всех характеристик абстракции; интерфейс должен гарантировать все для взаимодействия с пользователями.

Примитивными являются только такие операции, которые требуют доступа к внутренней реализации абстракции

1.3 Классификация

Классификация – это средство упорядочения знаний. Исторически известны только **три** подхода: классическая категоризация, концептуальная кластеризация, теория прототипов.

При **классическом подходе** все вещи, обладающие данным свойством или совокупностью свойств, формируют некоторую категорию. Причем наличие этих свойств является необходимым и достаточным условием, определяющим категорию.

Однако естественные категории нечетко отделены друг от друга. Большинство птиц летает, но не все. Стул может быть деревянным, метал-

лическим или пластмассовым, а число ножек необязательно равно четырем.

Концептуальная кластеризация – современный вариант классического подхода. Здесь сначала формируются концептуальные описания классов («кластеров объектов»), а затем мы классифицируем сущности в соответствии с их описаниями. Это именно понятие, а не признак или свойство: «Если песня скорее про любовь, чем про что-то другое, то мы помещаем ее в категорию *любовная песня*, хотя степень любовности едва ли можно измерить».

Концептуальную кластеризацию можно связать с теорией нечетких (многозначных) множеств, в которой объект может принадлежать к нескольким категориям одновременно с разной степенью точности.

Однако существуют некоторые абстракции, которые не имеют ни четких свойств, ни четкого определения. **Теория прототипов** определяет класс одним объектом-прототипом; новый объект относится к классу при условии, что он наделен существенным сходством с прототипом. Пример: игры.

На практике мы сначала идентифицируем классы и объекты по свойствам, важным в данной ситуации, то есть стараемся выделить и отобрать структуры и типы поведения с помощью словаря предметной области. Если таким путем не удалось построить нормальной структуры классов, мы пробуем концептуальный подход: уделяем внимание поведению объектов. Наконец, пробуем выделить прототипы и ассоциировать с ними объекты. Эти три способа классификации составляют теоретическую основу объектно-ориентированного анализа.

Границы между стадиями анализа и проектирования размыты, однако в процессе анализа мы моделируем проблему, *обнаруживая* классы и объекты, которые составляют словарь предметной области, в то время как на стадии проектирования мы *изобретаем* абстракции и механизмы, обеспечивающие требуемое поведение.

Существует *семь* проверенных практикой подходов к анализу объектно-ориентированных систем.

1 **Классические подходы** – опираются на классическую категоризацию.

Шлаер и Меллор: осязаемые предметы (датчики), роли (рабочий, студент), события (запрос, прерывание), взаимодействие (встреча, пересечение).

Росс (БД): люди (выполняют определенные функции), места (области, связанные с людьми или предметами), предметы (осязаемый материальный объект), организации (формально организованная совокупность людей и предметов, которая имеет определенную цель и не зависит от отдельных индивидуумов), концепции (принципы и идеи), события.

Коад и Йордан: структуры (отношения «целое-часть» и «общее-частное»), другие системы (внешние), устройства, события, роли, места, организационные единицы.

2 При **анализе поведения** именно динамическое поведение рассматривается как первоисточник классов и объектов; основан на концептуальной кластеризации. Инициаторы определенного поведения и его участники опознаются как объекты и делаются ответственными за определенные роли. Примеры: ввод/вывод, запрос.

3 **Анализ предметной области** – попытка выделить те объекты, операции и связи, которые эксперты данной области считают наиболее важными.

Этапы: построение скелетной модели предметной области; изучение существующих в данной области систем; определение сходства и различий между системами; уточнение общей модели для приспособления к нуждам конкретной системы.

Пример: предметная область – бухгалтерские отчеты; определяются абстракции и механизмы, обслуживающие все виды отчетов; цель исходной задачи – создание системы отчетов.

Эксперт – будущий пользователь системы (бухгалтер, диспетчер).

4 По отдельности все вышеперечисленные подходы сильно зависят от индивидуальных способностей и опыта аналитика. **Анализ вариантов** предусматривает упорядоченное последовательное их использование. Основан на перечислении и тщательной проработке сценариев работы системы.

5 *Метод CRC-карточек* удачно реализует на практике предыдущий подход (Class / Responsibilities / Collaborators – Класс / Ответственности / Участники). Сверху на карточке пишется название класса, снизу в левой половине – за что он отвечает, в правой – с кем он сотрудничает. Проход по сценарию приводит к дописыванию новых пунктов в существующих карточках или к созданию новых. Расположение карточек может показать поток сообщений между объектами и иерархию классов.

6 *Неформальное описание* – радикальная альтернатива классическому анализу (Аббот). Надо описать задачу или ее часть на обычном (человеческом) языке, а потом подчеркнуть существительные и глаголы. Существительные – кандидаты на роль классов, а глаголы могут стать именами операций. Метод можно автоматизировать. Используется в Токийском технологическом институте и в Fujitsu.

7 Вторая альтернатива классической технике – использование *структурного анализа* как основы объектно-ориентированного проектирования. После его проведения мы уже имеем модель системы, описанную диаграммами потоков данных; исходя из этого, можно приступить к определению классов и объектов любыми другими способами.

2 ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В СРЕДЕ C++

2.1 Класс как абстрактный тип

2.1.1 Класс как расширение понятия структуры. Статические компоненты класса

Класс – это производный структурированный тип, который задает некоторую совокупность типизированных данных и позволяет определить набор операций над этими данными.

Определяется с помощью конструкции:

ключ_класса имя_класса { список_компонентов };

где:

ключ_класса – одно из служебных слов class, struct или union;

имя_класса – произвольно выбранный идентификатор;

список_компонентов – определения и описания типизированных данных и принадлежащих к классу функций (**тело класса**).

Компонентами класса могут быть данные, функции, классы, перечисления, битовые поля, дружественные функции, дружественные классы и имена типов. Другими словами, компоненты класса – это типизированные данные (базовые и производные) и функции.

Принадлежащие классу функции называются **методами класса** или **компонентными функциями**. Данные класса называют **компонентными данными** или **элементами данных класса**.

Класс отличается от структуры возможностью включения компонентных функций.

```
struct complex1    // Класс «комплексное число»
{
    double real;      // Вещественная часть
    double imag;      // Мнимая часть
    void define (double re=0.0, double im=0.0)
    { real=re; imag=im; }
    void display ()
    { printf("real=%5.2f, imag=%5.2f\n",real,imag); }
};
```

Класс обладает правами типа, для описания объекта класса используется запись:

имя_класса имя_объекта;

Например:

```
complex1 x1,x2,D;
complex1 *p=&D;
complex1 dim[8];
```

К компонентам класса можно обращаться при помощи:

- 1) «квалифицированных» имен:

имя_объекта.имя_класса::имя_компонента

2) уточненных имен: *имя_объекта.имя_компонента*

Примеры:

```
x1.define();           // Параметры выбираются по умолчанию
x2.define(4.3,20.0);    // 4.3 + i 20.0
x2.display();           // Вывод на экран: real= 4.3, imag= 20.0
```

При использовании указателей на объект класса используется следующая запись:

указатель_на_объект_класса -> имя_компонента

Например:

```
p->real=2.3;
p->imag=6.1;
p->display();
```

Рассмотрим класс, описывающий товары на складе магазина. Компонентами класса будут:

- название товара;
- оптовая (закупочная) цена;
- розничная (торговая) наценка;
- функция ввода данных о товаре;
- функция вывода сведений о товаре с указанием розничной цены.

```
#include <stdio.h>
class goods // Класс «товары»
{
    char name[40]; // Название
    float price;    // Закупочная цена
    static int percent; // Торговая наценка в %
    void input ()
    {
        printf("Название товара: "); scanf("%s", name);
        printf("Закупочная цена: "); scanf("%i", price);
    }
}
```

```

void display ()
{ printf(“%10s\t, цена %5.2f\n”,name,price*(1+goods::percent*0.01)); }
};
void main()
{
goods wares[5];
...
}

```

Торговая наценка определена как статический компонент класса. Такие компоненты не дублируются при создании объектов (экземпляров) класса, т.е. каждый статический компонент существует в единственном экземпляре. Доступ к нему возможен только после инициализации:

тип имя_класса::имя_компонента инициализатор;

Например:

```
int goods::percent=12;
```

2.1.2 Конструкторы, деструкторы и доступность компонент класса

Главный недостаток обоих рассмотренных примеров: отсутствие автоматической инициализации создаваемых объектов – необходимо явно вызывать функции `define()`, `input()` или присваивать значения компонентным данным объекта.

Для автоматической инициализации компонентных данных объекта в определение класса можно явно включать специальную компонентную функцию, называемую **конструктор**:

имя_класса (список_формальных_параметров)
{ операторы_тела_конструктора };

Имя такой функции должно совпадать с именем класса. Она будет автоматически вызываться при определении или размещении в памяти каждого объекта класса. Переделаем часть первого примера:

```

complex1 (double re=0.0, double im=0.0)
{ real=re; imag=im; }

```

Объявление данных примет вид:

```
complex1 x1,x2(4.3,20.0);
```

Особенности конструкторов

1 Для конструктора не определяется тип возвращаемого значения (в т.ч. *void!*).

2 Конструктор всегда автоматически вызывается при определении (создании) объекта класса. При отсутствии параметров используются значения по умолчанию.

3 Конструктор существует для любого класса, причем он может быть создан без явных указаний программиста – т.н. конструктор копирования: `class F { ... public: F (const F&); ... }`

4 В классе может быть несколько конструкторов (перегрузка), но только один со значениями параметров по умолчанию.

5 Нельзя получить адрес конструктора (`&complex1`).

6 Параметром конструктора не может быть его собственный класс, но может быть ссылка на него, как у конструктора копирования.

7 Конструктор нельзя вызывать как обычную компонентную функцию.

Существует два способа инициализации данных объекта с помощью конструкторов.

Первый – передача значений параметров в тело конструктора (см. пред. пример).

Второй – применение списка инициализаторов данного объекта, помещаемого между списком параметров и телом конструктора:

имя_класса (список_формальных_параметров): список_инициализаторов

{ тело_конструктора };

Каждый инициализатор списка относится к конкретному компоненту и имеет вид:

имя_компонента_данных (выражение)

Например:

```

class AZ
{ int ii; float ff; char cc;
  public:
    AZ(int in, float fn, char cn): ii(5), ff(ii*fn+in), cc(cn) {}
    ...
}
AZ A(2,3.0,'d');    // Получаем: A.ii=5, A.ff=17, A.cc='d'

```

Деструктор (разрушитель объектов) класса имеет формат:

~имя_класса () { операторы_тела_деструктора };

Особенности деструкторов

- 1 Название состоит из значка «тильда» и имени класса (без пробелов).
- 2 У деструктора не может быть параметров (в т.ч. *void*).
- 3 Деструктор не имеет возвращаемого значения (в т.ч. *void*).
- 4 Вызов деструктора осуществляется неявно, автоматически, как только объект класса уничтожается.

Применение деструкторов имеет смысл при работе с динамической памятью:

```
~stroka() { delete [] ch; }
```

Видимость (доступность) компонент

Все компоненты класса могут иметь три степени доступа («видимости»):

public («общедоступный») – доступны для других компонентов данного класса, других классов и всех остальных компонентов программы;

protected («защищенный») – доступны для других компонентов данного класса и его потомков (при построении иерархии классов);

private («собственный») – доступны только для других компонентов данного класса.

Спецификатор доступа представляет собой одно из вышеуказанных слов, за которым следует двоеточие. Все компоненты от этого места до

конца определения класса или до другого спецификатора изменяют статус на вышеуказанный.

Основной принцип ООП – инкапсуляция (сокрытие) данных внутри объектов – не выполняется для класса *struct*, поскольку все компоненты получают статус *public*. Все компоненты класса, описанного при помощи служебного слова *class*, получают статус *private*.

```
struct complex1    // Класс «комплексное число»
{
// По умолчанию – public
complex1 (double re=0.0, double im=0.0)
{ real=re; imag=im; }
void display ()
{ printf("real=%5.2f, imag=%5.2f\n",real,imag); }
private:          // Изменить статус доступа на «частный»
double real;      // Вещественная часть
double imag;      // Мнимая часть
};
```

2.1.3 Компонентные данные

Определение данных класса аналогично описанию объектов базовых и производных типов, за некоторыми исключениями: так, при описании элементов класса не допускается их инициализация.

Ошибка: `struct complex1 { ... double real=0; ... };`

Для инициализации компонентных данных объектов должен использоваться автоматический или явно вызываемый конструктор соответствующего класса.

Для доступа к компонентным данным из операторов, выполняемых вне определения класса, непосредственное использование имен элементов недопустимо! Формально такой запрет можно обойти переопределением степени видимости на «public», но фактически это противоречит основному принципу ООП.

Статические компоненты класса имеют практический смысл при работе со связными списками, указывая на их начала:

```
class list
{
    ...
public:
    static list *begin;    // Начало списка
    ...
};
list *list::begin=NULL;    // Инициализация статического компонента
```

На статические данные класса распространяются правила статуса доступа. К моменту обращения к таким данным объекты могут быть еще не определены, либо их может быть несколько. Возможность обойтись без имени конкретного объекта при обращении к статическим данным класса обеспечивают статические компонентные функции:

имя_класса::имя_статической_функции

2.1.4 Компонентные функции

В отличие от обычных (глобальных) функций, компонентная функция имеет доступ ко всем компонентам класса, с любым статусом доступа.

При определении классов их компонентные функции могут быть специфицированы как подставляемые (inline). По умолчанию подставляемой считается функция, чье определение (не только заголовок, но и тело) полностью размещено в теле класса (см. все предыдущие примеры). *Преимущество*: быстрое обращение. *Недостаток*: такая функция не может содержать циклы и переключатели, а также не может быть рекурсивной.

Обычно используется другой подход: в теле класса размещается только прототип (заголовок) компонентной функции, а ее определение — вне класса, как определение обычной программной функции (с использованием «квалифицированного обращения»). Описание класса с внешним определением его компонентных функций дает возможность, не меняя ин-

терфейс объектов класса, по-разному определять его компонентные функции.

Приведенный ниже пример использует класс «точка на экране».

```
#include <graphics.h>
#include <conio.h>
class point
{
protected:
    int x,y;                // Координаты точки
public:
    point(int xi=0, int yi=0);    // Конструктор
    int getx();                  // Координата по X
    int gety();                  // Координата по Y
    void show();                 // Изобразить на экране
    void move(int xn=0, int yn=0); // Переместить в новое место
private:
    void hide();                 // Убрать точку с экрана
};
point::point(int xi=0, int yi=0)
{ x=xi; y=yi; }
int point::getx() { return x; }
int point::gety() { return y; }
void point::show() { putpixel(x,y,getcolor()); }
void point::hide() { putpixel(x,y,getbkcolor()); }
void point::move(int xn=0, int yn=0)
{
    hide();
    x=xn; y=yn;
    show();
}
void main()
{
    point A(200,100), B;
    int gd=0,gm;
```

```

initgraph(&gd,&gm,"");
A.show(); getch();
B.show(); getch();
B.move(200,200); getch();
A.move(); getch();
closegraph();
}

```

2.1.5 Указатель *this*

Когда функция, принадлежащая классу, вызывается для обработки данных конкретного объекта, этой функции автоматически и неявно передается указатель на тот объект, для которого функция вызвана (вспомните Sender:TObject в Delphi). Этот указатель имеет фиксированное имя *this* и незаметно для программиста определен в каждой функции класса следующим образом:

имя_класса * const this = адрес_объекта;

Имя *this* является служебным словом. Явно определить или переопределить этот указатель нельзя и не нужно.

Обращение по ссылке:

```

{
this->real=5; this->imag=2;
}

```

2.1.6 Друзья классов

Дружественной функцией класса называется функция, которая, не являясь его компонентом, имеет доступ к его защищенным (protected) и собственным (private) компонентам. Функция не может стать другом класса без его согласия. Для получения прав друга она должна быть описана в теле класса со спецификатором ***friend*** (т.е. в теле класса располагается прототип функции). Использование механизма дружественных функций поз-

воляет упростить интерфейс между классами: из классов можно убрать некоторые компонентные функции, предназначенные для доступа к определенным «скрытым» компонентам.

Особенности дружественных функций

- 1 Дружественная функция при вызове не получает указателя `this`.
- 2 На дружественную функцию не распространяется действие спецификаторов доступа; место расположения ее прототипа внутри класса безразлично.
- 3 Дружественная функция может быть как глобальной функцией программы, так и компонентной функцией другого класса.
- 4 Функция может быть дружественной по отношению к нескольким классам.

2.1.7 Расширение действия (перегрузка) стандартных операций

Язык C++ позволяет распространить действие любой стандартной операции на новые типы данных, вводимые пользователем. Такая возможность называется *перегрузкой стандартных операций*. Процесс осуществляется при помощи специального «*оператора-функции*»:

тип оператора знак_операции (спецификация_параметров) { операторы_тела }

где количество параметров зависит от «-арности» операции.

Например, для распространения действия операции «*» (умножить) на объекты класса `T`, может быть введена функция:

`T operator * (T x, T y)`

Таким образом, если описаны два объекта `A` и `B` класса `T`, то выражение `A*B` интерпретируется как вызов функции `operator * (A, B)`.

Пример перегрузки операции сложения для строк:

`stroka& operator + (stroka& A, stroka& B)`

2.2 Наследование и другие возможности классов

2.2.1 Наследование классов

Обоснованно введенный в программу объект призван моделировать свойства и поведение некоторого фрагмента решаемой задачи, связывая в единое целое данные и методы, относящиеся к этому фрагменту. Объекты взаимодействуют между собой и с другими частями программы с помощью сообщений. В каждом сообщении объекту передается некоторая информация. В ответ на сообщение объект выполняет некоторое действие, предусмотренное набором компонентных функций того класса, которому он принадлежит. Таким действием может быть изменение внутреннего состояния (изменение данных) объекта либо передача сообщения другому объекту.

Каждый объект является представителем конкретного класса. Объекты разных классов и сами классы могут находиться в отношении наследования, при котором формируется иерархия объектов, соответствующая заранее предусмотренной иерархии классов.

Иерархия классов позволяет определять новые классы на основе уже имеющихся. Имеющиеся классы обычно называют **базовыми (порождающими)**, а новые классы, формируемые на основе базовых, – **производными (порожденными), классами-потомками** или **наследниками**. Производные классы «получают наследство» – данные и методы своих базовых классов – и, кроме того, могут пополняться собственными компонентами (данными и методами). Наследуемые компоненты не перемещаются в производный класс, а остаются в базовых классах. Сообщение, обработку которого не могут выполнить методы производного класса, автоматически передается в базовый класс. Если для обработки сообщения нужны данные, отсутствующие в производном классе, то их пытаются отыскать автоматически и незаметно в базовом классе.

Пример – иерархия «точка – круг – круг_в_квадрате».

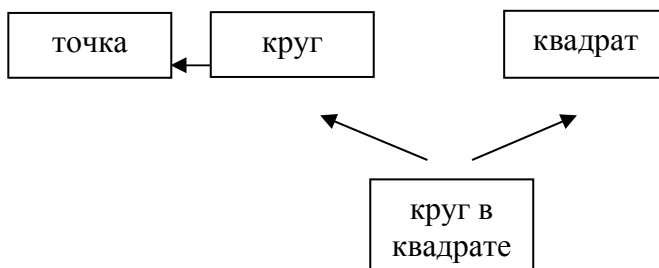
При наследовании некоторые имена методов (компонентных функций) и (или) компонентных данных базового класса могут быть по-новому определены в производном классе. В этом случае соответствующие компо-

ненты базового класса становятся недоступны из производного класса. Для доступа используется операция «::» – уточнение области видимости.

Наследование в иерархии классов может отображаться и в виде дерева, и виде более общего *направленного ациклического графа*, в котором показаны отношения между объектами.

C++ допускает *множественное наследование* – возможность для некоторого класса наследовать компоненты нескольких никак не связанных между собой базовых классов.

Например:



При описании производного класса приводится список всех его базовых классов:

```
class S: X, Y, Z {...};
```

Потомку доступны только те компоненты предков, которые имеют статус доступа *public* и *protected*. В порожденном классе все они получают статус *private*, если новый класс определен словом *class*, и статус *public*, если определен словом *struct*. Разумеется, статус компонент можно изменять явным указанием нового статуса – как отдельных компонент, так и всего наследуемого класса:

```
class S: X, protected Y, public Z {...};
```

Ни базовый класс, ни производный не могут быть объявлены с помощью ключевого слова *union*! Объединения не могут использоваться при построении иерархии классов.

Конструкторы и деструкторы при наследовании

Конструкторы и деструкторы не наследуются, а создаются заново в каждом классе-потомке.

Конструктор базового класса всегда вызывается и выполняется до конструктора производного класса. Деструктор производного класса вызывает деструкторы базовых классов, то есть порядок уничтожения объектов противоположен по отношению к порядку его конструирования.

Конструкторы и деструкторы автоматически получают статус доступа *public*.

Пример: `~krug_square() {~krug(); ~square();}`

В любом классе могут быть в качестве компонентов определены другие классы. В этих классах будут свои деструкторы, которые при уничтожении объекта охватывающего (внешнего) класса выполняются после деструктора охватываемого класса.

2.2.2 Множественное наследование и виртуальные базовые классы

Класс называют *непосредственным (прямым) базовым классом* (прямой базой), если он входит в список базовых при определении класса. В то же время для производного класса могут существовать косвенные или *непрямые* предшественники, которые служат базовыми для классов, входящих в список базовых.

А (базовый класс – прямая база для В)

↑

В (производный от А класс – прямая база для С)

↑

С (производный класс – с прямой базой В и косвенной А)

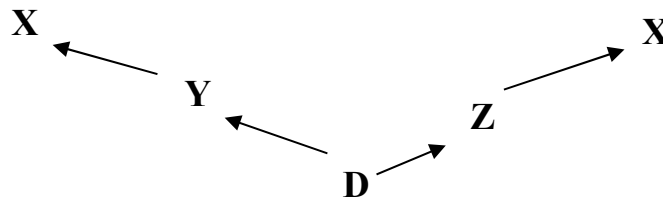
Класс может иметь несколько прямых базовых классов, т.е. может быть порожден из любого числа базовых классов (см. пример ранее). Такая ситуация называется *множественным наследованием*.

При множественном наследовании никакой класс не может больше одного раза использоваться в качестве непосредственного базового. Однако класс может сколько угодно раз быть непрямым базовым классом:

```
class X {...; f(); ...};
class Y: public X {...};
class Z: public X {...};
```

```
class D: public Y, public Z {...};
```

В данном примере класс X дважды опосредованно наследуется классом D:



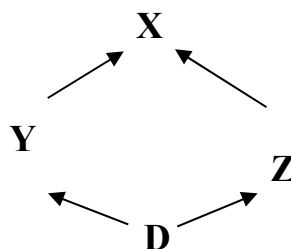
Проиллюстрированное дублирование класса соответствует включению в производный объект нескольких объектов базового класса. В таком случае для обращения к компонентам класса X необходима «полная квалификация», например:

D::Y::X::f() или D::Z::X::f()

Чтобы устранить дублирование объектов непрямого базового класса при множественном наследовании, этот базовый класс необходимо объявить виртуальным:

```
class X {...; f(); ...};
class Y: virtual public X {...};
class Z: virtual public X {...};
class D: public Y, public Z {...};
```

Теперь класс D будет включать только один экземпляр X, доступ к которому равноправно имеют классы Y и Z:



При множественном наследовании один и тот же базовый класс может быть включен в производный класс одновременно несколько раз, причем как виртуальный, так и не виртуальный. Возможны любые комбинации виртуальных и не виртуальных базовых классов.

2.2.3 Виртуальные функции и абстрактные классы

Иногда в базовый класс необходимо поместить функцию, которая должна по-разному выполняться в производных классах. Точнее, в каждом производном классе требуется свой вариант этой функции. Например, класс «графический объект» может определять какие-то действия над фигурой без конкретизации ее вида, а потомки этого класса – «квадрат», «треугольник» и т.п. – однозначно определяют форму и размеры этой фигуры. Классы, включающие такие функции, называются **полиморфными**.

Обычно выбор вызываемой функции определяется при написании исходного текста программы и не изменяется после компиляции. Такой режим называется **ранним** или **статическим связыванием**. Механизм виртуальных функций обеспечивает **позднее** или **динамическое связывание**.

```
class gr_obj
{
public:
    gr_obj(int col=7) {_color=col;}
    int color() { return _color; }
    virtual void draw() {}
    void show() { setcolor(color()); draw(); }
    void hide() { cback=getcolor(); setcolor(getbkcolor()); draw();
setcolor(color()); }
private:
    int _color,cback;
};
```

Особенности виртуальных функций

1 Виртуальной функцией может быть только нестатическая функция.

2 Виртуальной не может быть глобальная функция.

3 Функция, подменяющая виртуальную, в производном классе может быть описана как со спецификатором `virtual`, так и без него. В обоих случаях она будет виртуальной.

4 Виртуальная функция может быть объявлена дружественной в другом классе.

Чистой виртуальной называется компонентная функция, которая имеет следующее определение:

virtual тип имя (параметры) = 0;

В нашем примере: virtual void draw()=0;

Чистая виртуальная функция «ничего не делает» и недоступна для вызовов. Ее назначение – служить основой для подменяющих ее функций в производных классах. Класс, в котором есть хотя бы одна чистая виртуальная функция, называется **абстрактным**. Такой класс не может иметь экземпляров (объектов); он служит исключительно для построения иерархии классов.

2.2.4 Локальные классы

Класс может быть определен внутри блока – например, внутри тела функции. Такой класс называется **локальным**.

Локализация класса предполагает недоступность его компонентов вне области определения класса (вне тела функции или блока, в котором он описан или определен). Локальный класс не может иметь статических данных. Компонентные функции локальных классов могут быть только встроенными.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

- 1 **Мельников, А. Ю.** Объектно-ориентированный анализ и проектирование информационных систем : учебное пособие для студентов специальностей «Экономическая кибернетика» и «Интеллектуальные системы принятия решений» / А. Ю. Мельников. – Краматорск : ДГМА, 2006.–184с.
- 2 **Подбельский, В.В.** Язык C++ : учебное пособие / В. В. Подбельский. – 5-е изд. – М. : Финансы и статистика, 2001. – 560 с. – ISBN 5-279-02204-7.
- 3 **Страуструп, Б.** Язык программирования C++ / Б. Страуструп. – М. : Радио и связь, 1991. – 352 с.
- 4 **Павловская, Т.А.** C/C++. Программирование на языке высокого уровня / Т. А. Павловская. – СПб. : Питер, 2001. – 464с. – ISBN 5-318-00001-0.

5 **Павловская, Т.А.** С/С++. Структурное программирование : практикум / Т.А.Павловская, Ю.А.Щупак. – СПб. : Питер, 2002. – 240 с. – ISBN 5-94723-447-5.