МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Вл. Пономарев

# Объектно-ориентированное программирование на C++

Учебно-методическое пособие

Озёрск — 2012

УДК 681.3.06
П56

Пономарев В.В. Объектно-ориентированное программирование на C++. Учебно-методическое пособие. Озерск: ОТИ НИЯУ МИФИ, 2012. — 47 с., ил.

Пособие предназначено для изучения дисциплины «Объектно-ориентированное программирование» студентами, обучающимися по специальности «Программное обеспечение вычислительной техники и автоматизированных систем».

Рецензенты:
1)
2)

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

# Содержание

# Программное обеспечение

## *Предметная область*

На развитие языков программирования основное влияние оказывают два следующих фактора, представляющих предметную область программирования:

а) развитие технической базы вычислительных средств;

б) эволюция задач, которые предположительно могут быть решены при помощи средств вычислительной техники.

Заметим, что эти процессы являются взаимосвязанными. Развитие языков программирования ведет к совершенствованию технической базы, а изменения в средствах, предоставляемых вычислительными устройствами, ведет, в свою очередь, к постановке новых задач, которые требуется решить с помощью вычислительных средств.

Если рассмотреть в ретроспективе развитие языков программирования с точки зрения развития технической базы вычислительных средств, то можно отметить следующие важные этапы:

а) физическая модель программирования дает возможность решать задачи вычислительного характера, в основном в академических целях, в практическом плане — для решения задач в военной области.

Характерная особенность начального этапа развития языков программирования — отсутствие средств взаимодействия оператора с исполняемой программой.

б) процедурное программирование явилось ответом на развитие средств вычислительной техники в плане интерактивного взаимодействия между оператором и вычислительным процессом. Этому способствовало развитие средств управления вычислительными процессами на ЭВМ (например, появление операционных систем), а также появление новых технических средств для отображения и ввода информации — дисплеев, терминалов, принтеров, устройств для ввода графической информации и т.п.

в) современное состояние развития средств вычислительной техники таково, что ни одна из парадигм программирования не предоставляет адекватных средств, описывающих интерактивное взаимодействие пользователя с программными компонентами. Интерактивность взаимодействия пользователя с техническим устройством на настоящий момент является основным предметом программирования.

В то же время, по мере развития аппаратных средств и соответствующих им программных компонентов, изменялись и требования, предъявляемые к программному обеспечению.

Рассматривая этот процесс в ретроспективе, можно выделить следующие важные этапы:

а) на первоначальном этапе развития программирования к программам не предъявлялось никаких требований, за исключением работоспособности; понятие «программное обеспечение» еще не сформировалось; программы создаются в основном отдельными программистами; сложность программ в значительной мере определялась ограничениями аппаратного обеспечения, в основном объемом оперативной памяти;

б) по мере развития аппаратных средств и методов программирования появляется возможность проектирования больших программ (достигающих в объеме миллионы строк кода); на первый план выходят вопросы создания программного обеспечения коллективами программистов; разрабатываются методы технологии программирования, стандарты на языки программирования и на процессы проектирования; в целом этот этап можно охарактеризовать как осознание программирования как отрасли промышленного производства — программы становятся предметом технологического процесса, что влечет за собой установление норм качества программы, как промышленного продукта, и технологии их обеспечения;

в) в конце двадцатого столетия применение компьютеров охватывает практически все области профессиональной деятельности человека; пользователям все более требуются программы, позволяющие коллективно выполнять свою работу; программное обеспечение становится все более распределенным по множеству компьютеров, связанных глобальными и локальными сетями; программное обеспечение становится все более направленным на самые современные достижения в области визуализации и мультимедийных средств; резко возрастают требования к интеллектуальным возможностям программных компонентов;

***Выводы***

Предметная область программирования имеет тенденцию к непрерывному развитию. Этот процесс носит спиралевидный характер — появление новых технических средств приводит к появлению программ с новыми техническими возможностями, а экспансия программного обеспечения во все новые области применения стимулирует совершенствование технической базы. При этом развитие средств программирования объективно отстает от развития технической базы.

## *Качество программного обеспечения*

Целью программной инженерии (software engineering) является построение качественного ПО. Качество программного продукта можно оценить множеством различных характеристик. При этом одни характе

ристики оцениваются пользователями, другие — разработчиками программ. Первую категорию характеристик называют внешними, а вторую — внутренними.

Приведем наиболее важные внешние характеристики программных продуктов по Мейеру [1].

***Корректность***

*Корректность* (correctness) — это способность программного продукта выполнять задачу в соответствии с ее спецификацией.

В современном программировании невозможно обеспечить корректность программного модуля, функции, метода, основываясь только на корректности алгоритма. Обеспечение корректности является сложным вследствие иерархичности программного обеспечения — корректность программ, непосредственно взаимодействующих с пользователем, базируется на корректности библиотек, системного программного обеспечения, других программных компонентов промежуточного уровня, иначе говоря, функционирование приложений происходит в многоуровневой вычислительной среде (рисунок 1), на каждом уровне которой должна быть обеспечена должная корректность вычислений.

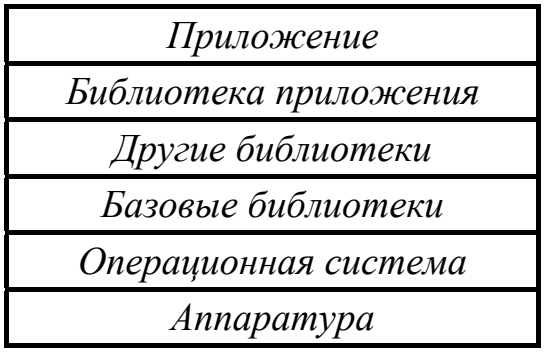


Рисунок 1 — Уровни вычислительной среды

***Устойчивость***

*Устойчивость* (robustness) — это способность программного продукта адекватно реагировать на аварийные ситуации.

Устойчивость дополняет корректность. Корректность описывает поведение системы в случаях, определенных спецификацией; устойчивость характеризует поведение системы за пределами этой спецификации.

***Расширяемость***

*Расширяемость* (extendibility) — это легкость адаптации программного продукта к изменениям спецификации.

Расширяемость зависит от масштаба — по мере роста объема программного продукта обеспечение расширяемости становится все более сложным. Для обеспечения расширяемости необходимо следовать двум принципам:

- архитектура программной системы должна быть простой;
- компоненты системы должны быть автономными.

***Повторное использование***

*Повторное использование* (reusability) — это возможность многократного использования программных компонентов для построения различных приложений.

При создании индустрии программного обеспечения необходимость повторного использования становится насущной проблемой. Это качество ПО представляется весьма важным, поэтому оно рассматривается отдельно далее.

***Совместимость***

*Совместимость* (compatibility) — это возможность программных компонентов сочетаться произвольным образом.

Для обеспечения совместимости важна однородность построения программных компонентов и стандартны на способы взаимодействия между ними. Эти подходы включают в себя:

- стандартные форматы файлов.
- стандартные структуры данных.
- стандартные пользовательские интерфейсы.

***Эффективность***

*Эффективность* (efficiency) — это способность программного обеспечения как можно меньше зависеть от ресурсов оборудования: процессорного времени, используемой памяти, пропускной способности устройств связи и т.п.

Обеспечение эффективности должно сопоставляться с другими целями, такими как расширяемость и возможность повторного использования. Оптимизация может сделать программный продукт настолько специализированным, что его нельзя будет повторно использовать.

***Переносимость***

*Переносимость* (portability) — это возможность использования программного продукта в различных программных и аппаратных средах.

К переносимости имеет отношение не столько физическое оборудование, сколько аппаратно-программный механизм (платформа), на основе которого строится программа.

#### *Простота использования*

*Простота использования* (easy of use) — это легкость, с которой люди с различными знаниями и квалификацией могут научиться использовать программный продукт и применять его для решения своих задач. Сюда относится также простота установки, работы и текущего контроля. Основным подходом для обеспечения простоты использования является структурная простота.

#### *Функциональность*

*Функциональность* (functionality) — это степень возможностей, обеспечиваемых программным продуктом.

Желание сделать программный продукт многофункциональным не просто усложняет его — продукт становится противоречивым, теряется его основная идея, а управление проектом затрудняется.

#### *Выводы*

Обеспечение характеристик качества программного продукта предъявляет противоречивые требования к нему. Наибольшую важность при разработке программного продукта имеют следующие четыре характеристики: корректность, устойчивость, расширяемость, повторное использование кода.

#### *Замечание*

Описанные выше качества программного продукта в той или иной мере обеспечиваются за счет применения объектно-ориентированной технологии производства программ. Однако не нужно понимать это так, что именно объектно-ориентированный подход дает возможность получить качественный продукт. Язык, технология, методология или парадигма программирования всего лишь средство, инструмент производства. Как и всякий инструмент, его нужно уметь правильно использовать. В умелых руках любая технология или модель, хотя бы и физическая, может обеспечить почти все необходимые качества продукта.

Тем не менее, именно объектно-ориентированная технология позволяет обеспечить качества, недостижимые при других способах производства программ. Одним из таких важнейших качеств является повторное использование. Другим качеством является расширяемость. В совокупности они позволяют выстраивать сложные иерархические структуры программ и данных, обеспечивая при этом и другие качества ПО.

## *Повторное использование ПО*

#### *Цели повторного использования*

Повторное использование программного обеспечения является едва ли не главной целью индустрии ПО. Если сравнивать эту индустрию с

другими отраслями производства, то приходится признать, что программистам чаще всего приходится писать каждую новую программу с нуля, начиная с функции *main*.

В идеале хотелось бы, чтобы программу можно было «складывать» из кубиков так, как каменщик выстраивает стену из кирпичей. Однако в программировании все настолько сложно, что пока это не удается.

Мечта о повторном использовании компонентов не является новой. В 1968 году на конференции НАТО по проблемам разработки ПО пропагандировалась идея «серийного производства компонентов ПО». Однако впечатляющих успехов в этой области пока не очень много. Одним из значительных достижений здесь является технология повторно используемых объектов COM от Microsoft (1993 г.), наиболее полно воплотившаяся в объектах-элементах управления COM (называемых также элементами управления ActiveX). Развитие этой технологии привело в конечном итоге к созданию самой совершенной и законченной на настоящий момент системы программирования Microsoft .NET (1999 г.).

В современном программировании повторное использование реализуется в основном использованием в приложениях множества сторонних библиотек, построенных как на основе процедурной модели программирования, так и на основе объектно-ориентированного подхода.

С появлением объектно-ориентированного подхода в программировании много надежд возлагалось на повторное использование кода классов. Однако настоящего повторного использования кода не произошло.

Одной из основных причин этого часто называется так называемая проблема NIH (Not Invented Here, «изобретено не здесь»). Она основана на том, что повторное использование классов затрудняется необходимостью изучения образа мышления программистов, создающих классы.

Улучшать возможности повторного использования ПО важно потому, что это дает определенные преимущества при проектировании. Согласно Мейеру [1], это ведет к:

- своевременности;
- сокращению объема работ по сопровождению ПО;
- надежности;
- эффективности;
- совместимости;
- сокращению инвестирования.

Можно выделить несколько направлений повторного использования.

1. Повторное использование персонала. Ввиду высокой текучести программистских кадров возможности такого подхода ограничены.

2. Повторное использование проектов и спецификаций. Этот вид повторного использования не лучше предыдущего. Здесь возникает риск

повторного использования неправильно работающих или уже вышедших из употребления элементов проекта (спецификации).

3. Использование образцов проектов (design patterns). Образец — это архитектурный принцип, применимый во многих прикладных областях; следуя образцу можно построить решение некоторой проблемы. К сожалению, образец является лишь «учебным пособием», а не инструментальным средством повторного использования. Применение образца требует дополнительной работы над ним.

4. Повторное использование исходного текста. Это наиболее простой подход. Однако он также обладает недостатками. Во-первых, он не позволяет скрыть информацию (реализацию). Во-вторых, сложные проекты трудно изучать вторичными пользователями. В-третьих, в сложных проектах взаимосвязи между частями кода могут быть неочевидными.

5. Повторное использование абстрактных модулей. Под абстрактным модулем понимаются повторно используемые единицы (units of reuse), входящие в состав непосредственно применяемых программных компонентов, доступ из внешнего мира к которым может осуществляться через описание, содержащее лишь множество свойств каждой такой единицы. Важным здесь является не использование текста модуля, а использование его описания (интерфейса).

Заметим, что настоящим повторным использованием ПО, подходящим для реализации технологии конструирования программ из готовых компонентов, следует, видимо, считать повторное использование абстрактных модулей.

Развитие Сети (wired society) привело к развитию именно технологий предоставления программных услуг посредством их описаний с соответствующей реализацией. Однако много нерешенных проблем еще существует. Они касаются как формы предоставления описания, так и формы предоставления реализации компонентов.

## *Сложность программного обеспечения*

Объектно-ориентированное программирование, как принято считать, появилось как ответ на все возрастающую сложность программ. Сложность современных программ поражает воображение. Например, утверждается, что объем кода операционной системы Windows составляет 29 миллионов строк кода на ассемблере, Си и C++. Конечно, не все программы (хотя назвать операционную систему программой в принципе неправильно) имеют такие объемы. Более того, последние годы развития производства программного обеспечения показывают, что построение больших по объему программ уступает место построению про

грамм, функционирующих в виде небольших взаимодействующих между собой компонентов, действующих на распределенных вычислительных системах, таких, как всемирная сеть Интернет. Однако, построение таких систем также является сложным, как в плане проектирования, так и в плане обеспечения необходимых качеств.

Сложность программного обеспечения вызывается четырьмя основными причинами [1]:

- сложностью реальной предметной области, из которой исходит заказ на разработку;
- трудностью управления процессом разработки;
- необходимостью обеспечить достаточную гибкость программы;
- неудовлетворительными способами описания поведения больших дискретных систем.

Выделяют следующие пять признаков сложной системы.

1) Сложные системы являются иерархическими и состоят из взаимозависимых подсистем, в которых, в свою очередь, также могут быть выделены подсистемы.

2) Выбор, какие компоненты в данной системе считаются элементарными, относительно произволен и в большой степени оставляется на усмотрение исследователя. Иерархические системы называют разложимыми, если они могут быть разделены на четко идентифицируемые части, и почти разложимыми, если их составляющие не являются абсолютно независимыми.

3) Связи внутри компонентов обычно сильнее, чем связи между компонентами. Это позволяет отделять «высокочастотные» взаимодействия внутри компонентов от «низкочастотной» динамики взаимодействия между компонентами.

4) Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных — разные сложные системы содержат одинаковые структурные части.

5) Любая работающая сложная система является результатом развития работавшей более простой системы. Сложная система, спроектированная «с нуля», никогда не заработает. Следует начинать с работающей простой системы.

## Модели и парадигмы программирования

Парадигма программирования — это концептуальная модель организации вычислений с помощью языка программирования. Модель программирования — это сложившаяся в данной парадигме система, определяющая стиль формирования компьютерных программ.

Языки программирования являются абстракциями организации вычислений. Целью языков является наиболее эффективное использование возможностей вычислительных средств, существующих в конкретной области применения. Языки программирования направлены обычно не на физическую модель вычислительной машины (вычислительного устройства), а на некоторую его абстракцию (модель). Качество этой абстракции определяет во многом качество проектирования компьютерных программ с помощью этой модели.

Существует несколько тысяч языков программирования, созданных для самых различных целей, и осуществляющих те или иные парадигмы программирования. Часто одни и те же языки могут выражать сразу несколько парадигм. Изучение языков программирования как таковых интересно и само по себе, однако целью их применения должно являться следование парадигме, выражающей концепцию организации вычислений. Поэтому знание конкретной парадигмы программирования представляется более важным, чем знание конкретных синтаксических конструкций, с помощью которых она выражается.

Существует несколько важных парадигм и моделей программирования. Общей теории моделей и парадигм, по-видимому, не существует, разные авторы приводят различное понимание этих концепций.

### *Императивная парадигма*

Императивная парадигма программирования долгое время является доминирующей вследствие простоты реализации. Суть парадигмы заключается в пошаговом изменении состояния вычислительной системы, и является методологией программирования, ориентированной на архитектуру фон Неймана, которая также долгое время является доминирующей среди вычислительных архитектур.

Императивная парадигма программирования предполагает управление вычислительной средой посредством команд. В качестве команд при этом могут выступать инструкции процессора, операторы языка высокого уровня, а также процедуры (функции). Программа на императивном языке программирования представляет собой последовательность этих команд, выполняемых в порядке их описания. Выполнение команд приводит к изменению состояния вычислительного устройства.

Императивная парадигма есть следствие естественного развития программирования вычислительной машины фон Неймана. Машина фон Неймана — это ячейки памяти, хранящие данные и инструкции процессора, позволяющие изменять состояние ячеек памяти. Языки императивного программирования моделируют ячейки памяти при помощи переменных, а инструкции процессора — при помощи операторов.

Основным вычислительным оператором императивного языка является оператор присваивания, изменяющий состояние ячеек памяти. Основным структурным оператором является условный или безусловный переход.

В процедурных языках появляются операторы условного ветвления и вызова процедуры. В структурных языках появляются условные операторы, циклические конструкции и механизмы программирования рекурсивных процедур.

В рамках данной парадигмы существует несколько моделей программирования, отражающих различные ступени исторического развития языков программирования. Большинство распространенных современных языков поддерживают императивную парадигму.

### Физическая модель

Физическая модель является прямым отражением физического устройства вычислительной машины на языки программирования.

Начальная стадия развития программирования (приблизительно 1945-1954 г.г.) предопределена физической сущностью вычислительной машины как совокупности ячеек памяти и алгоритма вычисления, заданного в виде числовых инструкций. Программирование ведется непосредственно в числовых значениях ячеек памяти. Задачи, решаемые с помощью ЭВМ, сводятся к программированию последовательности действий над числами — вычислительные задачи.

Уровень абстракции задачи максимально приближен к абстракции вычислительной машины как совокупности ячеек памяти и элементарных вычислительных операций. Определить физическую модель можно как модель программирования, направленную на прямое управление состоянием памяти машины.

Программа на физическом уровне — совокупность ячеек памяти и кодов машинных инструкций, использующих эти ячейки. Программирование для первых ЭВМ — это запись кодов инструкций с указанием адресов ячеек памяти.

Абстрактный пример программы в числах:

00000001 00000100

00000003 00000101

00000002 00000102

На начальном этапе развития программирования были созданы первые ассемблеры, псевдокоды и языки автоматического программирования. Их основным отличием от программирования непосредственно в числах являлось введение имен (мнемокодов) для обозначения инструкций и ячеек памяти (переменных).

Пример программы на ассемблере:

```
MOV R, B
ADD R, C
MOV A, R
```

Здесь R — некоторый регистр процессора, A, B, C — символические переменные, задающие адреса ячеек памяти. Программа моделирует оператор присваивания A = B + C.

Разработка программ на ассемблере сложна, потому что:

а) абстракция вычислительной машины напрямую зависит от ее архитектуры;

б) абстракция вычислительной машины никак не связана с предметной областью.

Несмотря на то, что физическая модель программирования давно не применяется для практического программирования, она используется в некоторых случаях, например, хакерами.

### Процедурная модель

Наиболее распространенная процедурная модель программирования появилась, очевидно, в связи с разработкой в 1954-1957 г.г. первого языка высокого уровня — языка FORTRAN (FORmula TRANslator — интерпретатор формул). Первые версии этого языка в целях повышения эффективности были ориентированы на архитектуру вычислительной машины IBM 407, но в результате эволюции конструкции языка стали более универсальными, что привело к созданию одной из наиболее успешных версий FORTRAN IV.

Несмотря на то, что уже существовали разработки языков, выполняющие преобразование арифметических выражений в машинный код, создание языка FORTRAN, предоставляющего возможность записи алгоритма вычислений с использованием условных операторов и операторов ввода-вывода, стало точкой отсчета эры алгоритмических языков программирования.

Процедурные языки представляют собой последовательность выполняемых операторов. Если рассматривать состояние вычислительной машины как состояние ячеек памяти, то процедурный язык — это последовательность операторов, изменяющих значение одной или не

скольких ячеек. К процедурным языкам относится большинство языков программирования, в том числе и современных.

Модель процедурного программирования основана на представлении программы в виде иерархии процедур и функций.

Абстракция задачи, используемая в процедурных языках, выражается в терминах операций и процедур, направленных на изменение элементов данных. В процедурном программировании появляются новые абстракции:

а) для данных — тип переменной;

б) для кода — оператор;

в) для программы в целом — процедура (функция).

Тип переменной абстрагирует программиста от ячейки памяти как совокупности бит и позволяет оперировать более абстрактными понятиями — целое и вещественное число, символ, строка символов. Тип — элемент абстракции элементов данных.

Оператор абстрагирует программиста от потока машинных инструкций. Оператор — элемент абстракции вычислительного действия.

Процедура абстрагирует программиста от потоков операторов. Процедура — элемент абстракции управления.

Программировать на процедурном языке проще, потому что:

а) абстракция вычислителя слабо зависит от его архитектуры;

б) абстракция вычислительного действия мощнее и интуитивно более понятна, чем инструкция процессора;

в) поток команд может быть описан в терминах предметной области.

Программировать на процедурном языке сложно, потому что:

а) физические типы данных отражают небольшое подмножество реальных объектов;

б) сложность управления программой растет экспоненциально с ростом количества элементов данных.

### Структурная модель

Структурное программирование, основанное на и вышедшее из процедурного программирования, приводит к развитию ряда важных методов проектирования и вводит новые абстракции:

а) для данных — структурная организация данных; появляются типы данных, для которых в физической машине не существует соответствующего физического представления;

б) для программы (проекта) — функционально-иерархическая декомпозиция, повторное использование проектных решений и технология тестирования программного обеспечения;

Достоинством структурного подхода к проектированию является систематический подход к разработке программного обеспечения.

Отмечают следующие недостатки структурного подхода:

а) функциональную модель трудно развивать, а архитектура приложения по мере развития стремится к неуправляемости;

б) функционально-ориентированный код трудно обобщать для многократного использования.

Задача программиста, использующего структурную модель, заключается в том, чтобы перейти от описания предметной области в терминах, присущей задаче, к описанию предметной области в терминах модели вычислителя (так же, как для процедурного программирования).

Большинство современных языков программирования поддерживают структурную модель.

#### Объектно-ориентированная модель

Объектно-ориентированная модель программирования сложилась в результате распространения языка C++ (1983-1986 г.г.). Примечательно, что автор C++ Б. Страуструп изначально называл свой язык «Си с классами», подчеркивая, видимо, что классы являются надстройкой.

В основе объектно-ориентированной модели программирования лежит абстракция предметной области (а не абстракция вычислительной машины) в виде иерархии классов и объектов — объектно-ориентированная декомпозиция. Задача описывается на абстрактном уровне как совокупность объектов, взаимодействующих друг с другом посредством сообщений. Взаимодействие объектов в разных языках осуществляется разными методами. В языке C++ одни объекты могут вызывать методы других.

Достоинством объектно-ориентированной модели программирования является возможность описывать программную модель в терминах, соответствующих предметной области.

Примерами современных императивных языков, допускающих объектно-ориентированную модель, кроме уже упомянутого языка C++, являются Object Pascal, PHP и другие.

### *Декларативная парадигма*

Декларативная парадигма программирования предполагает описание задачи в терминах объявлений и высказываний в символьной логике. Типичным примером декларативных языков являются языки логического программирования (языки, основанные на системе правил). Логическая модель программирования базируется на концепции «любая задача может быть сведена к цепочке логических рассуждений».

В отличие от императивных языков, операторы программы на языке логического программирования выполняются не в порядке их описания. Порядок их выполнения определяется системой реализации правил. Правила применяются до тех пор, пока это возможно, при этом конечное состояние вычислителя является результатом применения программы. Декларативная семантика оказывается проще семантики императивных языков, однако класс эффективно решаемых задач ограничен.

Наиболее характерным представителем языков логического программирования является PROLOG (1972 г.). Основные области его использования — системы искусственного интеллекта, такие, как экспертные системы, системы обработки текстов на естественных языках и т.п.

### *Аппликативная парадигма*

В языках функционального программирования (называемых также аппликативными языками) вычисления сводятся в определению функции, применение которой выполнит необходимое преобразование ячеек памяти. Разработка программы на функциональном языке заключается в описании сложных функций посредством более простых функций (функция является единственным элементом действия), которые применяются к исходным данным до тех пор, пока не будет достигнут результат.

Наиболее известным языком функционального программирования является язык LISP (LISt Processing — обработка списков), появившийся в 1958-1960 г.г. Его концепция основана на постулате «любая задача может быть сведена к работе со списком». Недостатки функционального программирования заключаются в сложности получения эффективных программ, а также в слабых средствах абстрагирования типов данных. Основное применение языки функционального программирования находят в системах искусственного интеллекта.

### *Объектно-ориентированная парадигма*

Объектно-ориентированная модель программирования сложилась в результате развития объектно-ориентированного подхода к проектированию (моделированию) сложных систем. Первым языком, поддерживающим объектно-ориентированную модель программирования, традиционно называют язык Simula (1960-1970 г.г.). Значительное влияние на развитие объектно-ориентированной модели оказал язык Smalltalk (1970-1980 г.г.), созданный под влиянием языков LISP и Simula.

Современная концепция объектно-ориентированного подхода к проектированию программ может быть выражена в трех важнейших понятиях: инкапсуляция, наследование и полиморфизм.

Инкапсуляция позволяет описывать абстрактные типы данных в виде логически и физически целостной совокупности элементов данных и процедур для управления ими. Инкапсуляция разделяет внутреннее устройство объекта от его видимого и используемого интерфейса (набора операций), что позволяет модифицировать объект без нарушения его функциональной работоспособности. В основе инкапсуляции лежит абстракция данных.

Наследование позволяет выстраивать иерархии классов и распределять, таким образом, базовую и специфическую функциональность между различными объектами. Наследование способствует более рациональному построению программ и дает возможность повторно и многократно использовать имеющийся код.

Полиморфизм заключается в возможности объектов разных типов проявлять себя одинаковым образом. При этом разнотипные объекты рассматриваются как одинаковые с точки зрения своего функционального поведения (интерфейса), но разные с точки зрения реализации этого поведения. В основе полиморфизма лежат наследование и механизм виртуальных функций, основанный, в свою очередь, на механизме позднего (динамического) связывания. Фактически полиморфизм проявляется в возможности объекта некоторого типа вести себя по-разному в зависимости от того, какой тип он представляет в данный момент исполнения программы.

В основе объектно-ориентированной парадигмы программирования лежит объектно-ориентированная декомпозиция. Разработка программы заключается в проектировании иерархии классов, описывающих предметную область в терминах действующих объектов и взаимоотношений между ними. Выполнение программы представляется как модель взаимодействующих объектов, являющихся представителями определенных классов. Взаимодействие осуществляется посредством вызовов одних объектов другими (через функции классов).

### Объектная модель

Концептуальная база объектно-ориентированной парадигмы программирования — это объектная модель. Выделяют следующие главные элементы этой модели [2]:

- абстрагирование (abstraction);
- инкапсуляция (encapsulation);
- модульность (modularity);
- иерархия (hierarchy).

Абстрагирование является одним из основных методов, используемых для решения сложных задач. Абстракция позволяет выделить су

щественные характеристики некоторого объекта, отличающие его от других видов объектов, и таким образом определить его концептуальные границы с точки зрения наблюдателя.

В объектно-ориентированном программировании выделяют два вида абстракций:

- класс как тип данных объектной природы;
- объект как представитель (экземпляр) класса.

Класс как тип данных вводит в систему программирования новый тип, иногда называемый пользовательским, а также абстрактным. Программирование, таким образом, ведет к формированию системы или иерархии типов и осуществляется внутри этой системы или иерархии.

## Классы и объекты

Класс является первым важнейшим понятием объектно-ориентированного программирования. Известные объектно-ориентированные системы программирования, такие, как Microsoft .NET или Java, используют классы в качестве основного инструментария. Языки программирования, такие, как C++, Object Pascal или PHP, используют классы в качестве дополнительного инструментария, позволяющего в итоге использовать объектно-ориентированную парадигму программирования в этих языках и соответствующих системах программирования.

Дополнением к понятию класса является понятие объекта, как представителя конкретного класса (для определения представителя класса более всего подходит английский термин *instance*, который можно перевести как «экземпляр»).

Заметим, что объектом называют также элемент системы программирования (программы), представляющий экземпляр встроенного типа, в особенности, если это пользовательский тип. Например, экземпляр типа, определенного как структура (*struct*) в языке Си, экземпляр записи (*record*) в языке Pascal и т.п., то есть любой действительный экземпляр любого действительного типа программы.

В принципе, объект как представитель класса как определяемого пользователем типа является таким же объектом. Характерная особенность объекта — занимаемая им память, в пределах которой хранятся значения. Особенность экземпляра класса в этом смысле конструктивная. Например, объект класса может и не занимать никакой памяти, то есть не задавать никаких значений. С другой стороны, объект класса может задавать не только значения, но и дополнительные, автоматически создаваемые структуры, характерные только для объектов класса.

Класс и объект понятия взаимодополняющие, но не равнозначные и не равноправные. С этой точки зрения класс есть описание типа, а объект — его конкретное воплощение в виде занимаемой памяти. Класс описывает некий абстрактный тип данных, обладающий поведением (и этим он значительно отличается от структуры данных). В основе класса лежит абстракция некоторого феномена (такого, как предмет или явление) внешнего мира (точнее — предметной области). Объект реализует эту абстракцию в виде конкретных значений. Однако экземпляр класса отличается от другого объекта программы тем, что он является действующим самостоятельно объектом на основе абстракции поведения этого объекта во внешнем мире. Объект класса может самостоятельно выполнять действия посредством посылки и приема сообщений, в роли которых выступают вызовы его методов (функций).

Класс, являясь описанием, существует только на момент написания программы, иначе говоря, только в ее тексте. Никаких классов в объектном модуле, равно как и в исполняемом модуле, сформированных в результате компиляции и компоновки текста программы, не существует.

Экземпляр класса, наоборот, существует в исходном тексте программы разве что в виде некоторой переменной, однако в объектной программе именно объект является главным действующим лицом, поскольку все действия в полностью объектно-ориентированной программе происходит только от его имени и по его требованию.

Дальнейшие разделы описывают классы и объекты с различных точек зрения. Целью является дать наиболее полное представление о роли классов и объектов в объектно-ориентированной программе.

### *Класс как абстракция*

Любой феномен предметной области программы, такой, как предмет или явление, может быть словесно описан перечислением его пассивных (статических) и активных (динамических) характеристик.

Под пассивными (статическими) характеристиками будем понимать здесь параметры, описывающие состояние, положение, конфигурацию и т.п. Пассивные характеристики часто можно измерить и всегда можно выразить какой-то величиной.

Под активными (динамическими) характеристиками будем понимать вероятную активность феномена в виде действий, которые он предположительно может выполнять (над собой).

Например, лампочку как сущность, можно описать физическими параметрами, такими, как мощность, напряжение, тип, цвет, размер, текущее состояние, степень износа (пассивные, статические характеристики), и возможными действиями, такими, как включить или выключить (активные, динамические, поведенческие характеристики).

Другой пример — точка на плоскости. Пассивными характеристиками точки будут координаты «координата-X» и «координата-Y», а активными — действие перемещения (переместить).

Заметим, что статические характеристики описываются существительными (e.g. мощность), а действия — глаголами (e.g. включить).

Объектно-ориентированное программирование имеет целью описать типы и их взаимодействие. Типы можно подразделить на встроенные, такие, как целочисленный тип *int*, и пользовательские, такие, как структура данных *struct*. В отличие от процедурного или структурного подхода к программированию, объектно-ориентированный подход рассматривает пользовательские типы данных как активные, то есть наделенные не только пассивными характеристиками, но и активными.

С этой целью программист формирует классы. Класс как абстракция есть выделение существенных для предметной области программы статических и динамических характеристик сущностей, предположительно составляющих предметную область.

Рассматривая сущность «лампочка», можно выделить такие существенные характеристики, как «текущее состояние» (статическая характеристика), «включить» и «выключить» (динамические характеристики).

При этом подразумевается, что для данной предметной области несущественны такие характеристики, как мощность, цвет, размер и т.п., поскольку они не влияют на взаимодействие с другими сущностями.

Абстрактную сущность можно изобразить графически. Такой подход используется, например, в языке моделирования объектно-ориентированных систем UML (Unified Modeling Language). На рисунке 2 приведено графическое изображение сущности «Лампочка».

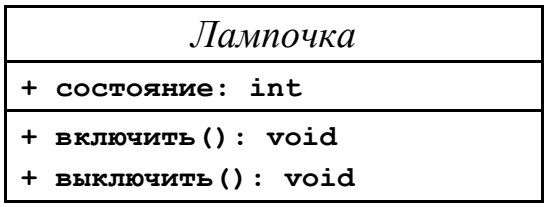


Рисунок 2 — Абстракция сущности «Лампочка»

Графическое изображение состоит из трех секций, расположенных по вертикали. Верхняя секция содержит название сущности, в данном случае «Лампочка». Вторая секция содержит перечень пассивных характеристик сущности. Принято называть эту секцию «Свойства». Наконец, третья секция содержит активные характеристики сущности. Эту секцию называют «Методы».

Таким образом, класс как абстракция может быть описан своим наименованием, перечнем свойств и перечнем методов. Свойства и методы — неотъемлемые характеристики классов и объектов в современном объектно-ориентированном программировании.

Свойства описывают классы и объекты с точки зрения статических характеристик. Каждое свойство обладает некоторым типом и является хранителем некоторого значения этого типа. Свойство можно считывать и изменять (записывать). При считывании мы узнаём текущее значение свойства, при записи мы меняем текущее значение на новое. С этой точки зрения можно сказать, что свойство используется в операторе присваивания, либо в левой его части, либо в правой. Свойство может быть использовано также в выражении (при считывании).

Методы описывают классы и объекты с точки зрения поведения.

С этой точки зрения методы есть функции, которые могут быть выполнены. Выполнение метода обычно ведет к изменению внутреннего состояния объекта, то есть его свойств. Выполнение метода называется «вызовом», поскольку на самом деле это просто вызов функции. Методы используются для управления объектом и взаимодействия объектов между собой: одни объекты могут вызывать методы других. Другие объекты могут также читать и записывать свойства.

На графическом изображении сущности «Лампочка» названиям свойств и методов предшествует знак "+". Он означает, что данное свойство или данный метод является общедоступным (*public*), иначе говоря, программист может свободно читать и записывать свойство или вызывать метод. Свойства и методы могут быть недоступными для программиста (*private*). В этом случае они помечаются знаком "–". Такие свойства и методы могут использоваться только методами самого класса (объекта). Некоторые свойства и методы могут иметь ограниченную доступность (*protected*). В этом случае они помечаются знаком "#".

Абстракция сущности формирует ее концептуальные границы с точки зрения наблюдателя и формирует так называемый *интерфейс* класса. Под интерфейсом в разных контекстах называют разные вещи. Интерфейсом класса было бы удобнее называть совокупность его общедоступных, открытых, свойств и методов, однако на практике, вследствие особенностей языков программирования, под интерфейсом класса часто понимаются все его свойства и методы, видимые программисту.

Если внимательно рассмотреть интерфейс сущности «Лампочка», приведенный на рисунке 2, то можно заметить, что состояние лампочки можно изменить двумя способами: использовать свойство «состояние» или методы «включить» и выключить». Можно было бы вообще удалить методы данного класса и пользоваться только его свойством. При этом, правда, нарушается семантика (смысл) интерфейса, абстракции. Поэтому свойство «состояние» следует пометить атрибутом *read only*, означающим «свойство только для чтения» (рисунок 3).

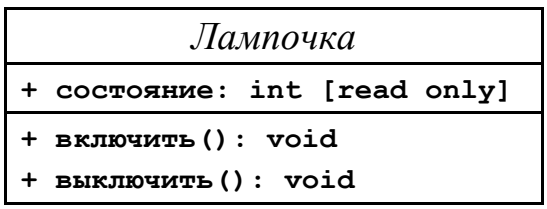


Рисунок 3 — Правильная абстракция сущности «Лампочка»

Теперь соблюдается правильная семантика абстракции: лампочку можно включить или выключить, можно также узнать ее состояние.

К сожалению, далеко не все языки и системы программирования поддерживают концепцию свойства. К таким языкам относится и язык программирования C++. Поэтому интерфейс класса «Лампочка» для этого языка должен быть графически изображен иначе (рисунок 4).

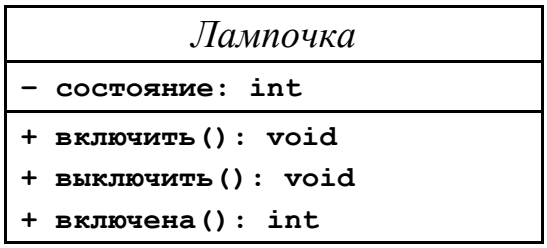


Рисунок 4 — Абстракция сущности «Лампочка» для языка C++

В новой редакции интерфейса класса мы видим, что хранителем состояния является закрытое (*private*) свойство, управлять которым можно только при помощи методов. При этом нам открывается часть реализации класса, которую абстракция должна скрывать. Смысл абстракции в том и состоит, чтобы описать только внешние границы концепции (то есть типа) и скрыть ее внутреннее устройство (то есть реализацию).

Кроме того, в интерфейс класса добавлен метод «включена», который возвращает текущее состояние лампочки и *почти* полностью аналогичен свойству «состояние» с атрибутом «только для чтения».

В качестве дополнительного примера абстракции приведем интерфейс сущности «Точка» (рисунок 5).

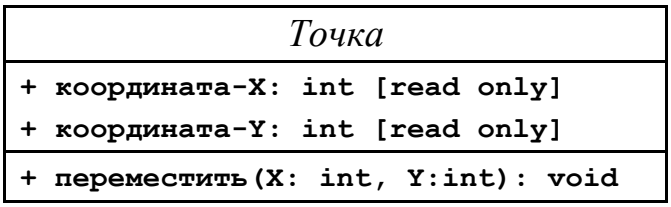


Рисунок 5 — Абстракция сущности «Точка»

Как видим, координаты точки можно только читать, а изменить их можно только посредством метода «переместить». Метод изменяет сразу все координаты. Пример приведен для того, чтобы показать, что методы могут иметь параметры. Заметим, что свойства также могут иметь параметры, однако это не является характерным признаком. Интерфейс этой сущности для языка C++ предлагается определить самостоятельно.

Таким образом, класс как абстракция есть перечисление его существенных свойств и методов, то есть интерфейс, видимый наблюдателю. Абстракция определяет интерфейс класса, но скрывает его реализацию.

## *Класс как тип*

Методология объектно-ориентированного программирования вводит концепцию пользовательских, или абстрактных, типов, и их взаимодействие. Программирование в объектно-ориентированной среде сводится к определению типов, их иерархий, и построению программы в виде модели взаимодействующих объектов — представителей типов.

Определим понятие типа более точно. Тип определяют:

1) множество элементов данных (занимающих память);

2) множество допустимых значений;

3) множество разрешенных операций;

4) отношения с другими типами.

В каждом языке программирования существуют так называемые встроенные типы. При этом язык может и не поддерживать концепцию типов, например, в физической или процедурной модели программирования. Тем не менее, любой встроенный тип обладает некоторыми характеристиками, позволяющими отделить его от других типов.

Прежде всего тип задает множество допустимых значений. Так, встроенный тип *char* языка Си задает целочисленные значения в диапазоне от –128 до +127, а тип *unsigned char* задает диапазон 0..255.

Во-вторых, тип задает также допустимые операции с объектом данного типа. Так, для целочисленных типов языка Си, в том числе для типов *char* и *unsigned char*, определены арифметические, логические и побитовые операции.

Такое толкование типа допустимо для встроенных типов, но недостаточно при определении типов пользователя, описывающих более сложные структуры данных, соответствующих сущностям предметной области. В качестве примера таких структур можно привести понятия положительного или комплексного числа, стека, очереди и других часто используемых в практическом программировании объектов.

Например, при определении концепции рационального числа диапазон допустимых значений не может быть задан одним определением, поскольку структура данных определяет два элемента, а именно — числитель и знаменатель. Множества допустимых значений этих элементов почти совпадают, но не полностью.

Для более сложных структур данных может понадобиться более сложное множество описывающих их элементов, и для каждого элемента потребуется, скорее всего, свое ограничение на допустимые значения.

Структурное программирование вводит понятие структуры данных, но не определяет понятие типа полностью. Так, определение *struct* в языке СИ задает множество определяющих тип элементов, но не определяет допустимые над ним операции. Операции задаются множеством

отдельно стоящих функций, никак не связанных со структурой данных. Это ведет к возможности выполнения ошибочных действий, когда операция вызывается с операндами неподходящих типов.

Концепция класса призвана определить понятие типа, интегрированного в язык (систему) программирования не только на уровне множества допустимых значений, но и на уровне множества допустимых операций. При этом некоторые языки позволяют определить для типа стандартные операции типа сложения или деления для того, чтобы новый тип можно было бы использовать так же, как и встроенный, в смысле синтаксиса операторов. К таким языкам как раз и относится язык C++. Нужно заметить, что языков, позволяющих определить для типа стандартные операции, немного. Язык C++ обладает в этом смысле самыми большими возможностями, в других языках эти возможности либо отсутствуют совсем, либо определены частично.

Однако концепция класса определяет понятие типа несколько шире. Классы (типы) могут образовывать иерархии, в которых одни классы являются базовыми для других, производных классов (типов). Кроме того, объекты или ссылки на классы (типы) могут быть параметрами методов класса, а также его составными частями (свойствами). При этом между классами могут возникать различные отношения или связи.

Согласно спецификации UML, наиболее распространены четыре типа отношений (всего существует семь отношений).

Отношение обобщения (*generalization*) возникает между производным и базовым типами. Оно означает, что базовый класс обобщает совпадающие характеристики базового и производного типов, производный тип является при этом специализацией (специальной разновидностью) базового. Пример отношения обобщения приведен на рисунке 6.

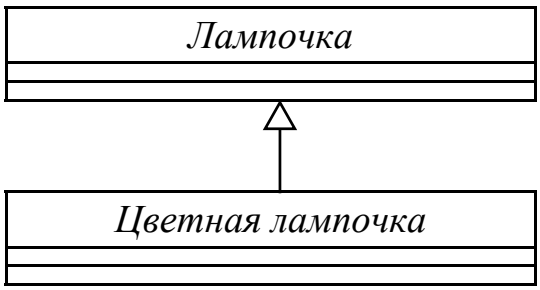


Рисунок 6 — Отношение обобщения

Здесь цветная лампочка является более сложной, более специфичной (по характеристикам) сущностью, чем лампочка просто. Поэтому общие характеристики лампочки как таковой описываются в базовом типе, а производный тип расширяет возможности базового. При этом часть

функциональности производного типа сосредоточена в базовом типе, а часть — в производном. Заметим, что отношение обобщения называют также отношением «*is a*» (является) или «*is like a*» (похож).

Например, «Цветная лампочка» «*is a*» «Лампочка», то есть цветная лампочка является лампочкой или похожа на нее.

Отношение ассоциации (*association*), описывает структурное отношение, в котором объекты одного типа связаны некоторым образом с объектами другого типа. Часто при этом типы находятся в отношении «часть-целое», когда один тип является составной частью другого.

На рисунке 7 приведено отношение ассоциации типа композиции.

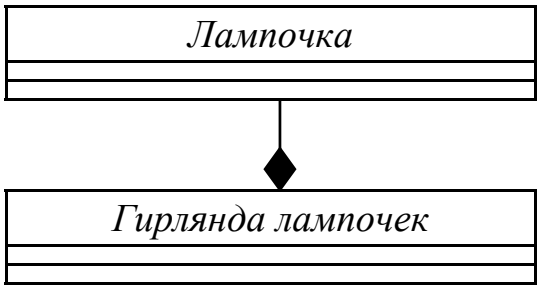


Рисунок 7 — Отношение ассоциации (композиция)

Здесь тип «Лампочка» является составной частью типа «Гирлянда лампочек». Физическая реализация этого отношения может быть различной, например, класс гирлянды лампочек может содержать элемент данных — массив указателей на тип лампочки или массив типов лампочки.

Предполагается, что лампочки физически являются составной частью гирлянды. В таком же отношении находятся, например, автомобиль и двигатель. С другой стороны, отношения ассоциации могут возникать в случае, когда составные части типа не являются его физическими частями. Примером является ассоциация, возникающая между типами «Учебное заведение» и «Преподаватель» и «Студент». Заметим, что в этом случае отношение обозначается прозрачным ромбиком, а отношение относят к типу агрегации [4].

Отношение ассоциации называют также отношением «*is part of*» (является частью). В нашем примере, «Лампочка» «*is part of*» «Гирлянда лампочек».

Следующим видом отношений является зависимость (*dependency*). Это отношение использования, при котором изменение спецификации (интерфейса) одного класса влияет на класс, его использующий. Часто при этом объекты одного класса передаются как аргументы методов другого класса.

Примером может служить отношение между классом приложения, использующего гирлянду лампочек, и классом гирлянды. Приложение содержит ссылку на тип гирлянды и, соответственно, использует его интерфейс. Изменение интерфейса гирлянды затронет также реализацию класса приложения. Рисунок 8 показывает отношение зависимости.

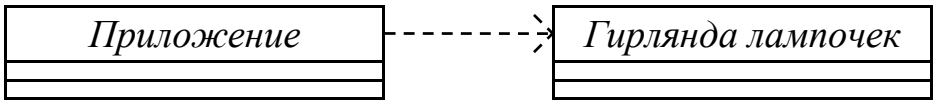


Рисунок 8 — Отношение зависимости

Четвертым отношением между классами (типами) является отношение реализации (*realization*). Это отношение имеет большое значение и применение в современном объектно-ориентированном программировании. Это семантическое отношение, при котором класс (тип) обязывается выполнить (реализовать) некоторый интерфейс, иногда называемый контрактом. Под интерфейсом при этом понимается тип, задающий набор связанных (семантикой, смыслом) методов и свойств и не имеющий их реализации. Иначе говоря, интерфейс — это описание, которое в принципе не может быть использовано, кроме как через реализацию в производных типах. Общепринято именовать типы, описывающие интерфейсы, с начальной буквой "I". Пример отношения реализации приведен на рисунке 9.

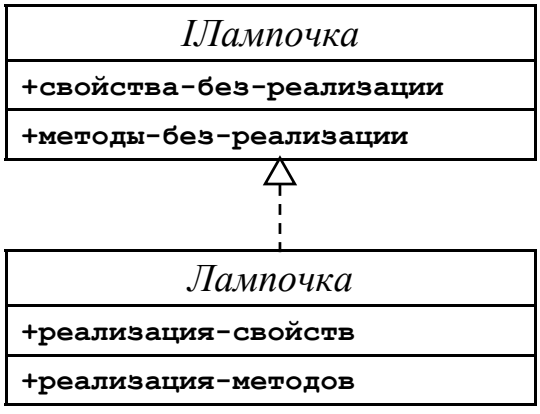


Рисунок 9 — Отношение реализации

Здесь приведен интерфейс некоторой абстрактной лампочки «ILампочка». Производный класс «Лампочка» обязан реализовать все свойства и методы этого интерфейса и, таким образом, гарантировать функциональность, предусматриваемую им. Заметим, что в языке C++ интерфейс не может содержать свойств (не поддерживает их).

## Класс как структура данных

Здесь мы рассмотрим, как описываются классы на языке C++.

Для описания класса используется структура данных типа *struct*, в которую добавлены два важных изменения.

Во-первых, структура данных может содержать не только описание элементов данных, но и функций, называемых функциями-элементами.

Во-вторых, структура данных может быть поделена на секции, описывающие области различного доступа к элементам класса.

Рассмотрим сначала функции-элементы. Для определенности будем описывать тип «Лампочка», по-английски *lamp*. Листинг 1 показывает, как можно описать тип лампочки на языке C++ и протестировать его.

***Листинг 1***

```
// класс - описание лампочки                    // 01
struct lamp {                                   // 02
    // хранит состояние лампочки                // 03
    int m_state;                                // 04
    // включает лампочку                        // 05
    void turn_on() {                            // 06
        m_state = 1;                            // 07
    }                                           // 08
    //  выключает лампочку                      // 09
    void turn_off() {                           // 10
        m_state = 0;                            // 11
    }                                           // 12
    // возвращает текущее состояние лампочки    // 13
    int state() {                               // 14
        return m_state;                         // 15
    }                                           // 16
};                                              // 17
// основная функция                             // 18
void main() {                                   // 19
    lamp k;                                     // 20
    k.m_state = 0;                              // 21
    k.turn_on();                                // 22
    k.turn_off();                               // 23
    int s = k.state();                          // 24
}                                               // 25
```

Приведенное описание соответствует графическому изображению типа лампочки, приведенному на рисунке 2.

В строке 4 структура описывает единственный элемент данных, определяющий текущее состояние объекта — переменную *m_state*. Этот элемент можно использовать обычным для структуры данных образом, что показано в строке 21. При выполнении этой строчки кода переменная *m_state* меняет свое значение с неопределенного на нулевое.

В строчках 5-8 структура описывает функцию-элемент *turn_on*. Она устанавливает элемент данных *m_state* в значение 1.

В строке 22 функция-элемент *turn_on* вызывается. Как видим, вызов функции-элемента синтаксически эквивалентен обращению к элементу данных. Здесь используется операция «точка», означающая обращение к элементу структуры. Поскольку элемент *turn_on* является функцией, круглые скобки обязательны (для элемента данных, являющимся свойством, скобки являются синтаксической ошибкой).

В строчках 9-12 структура описывает функцию-элемент *turn_off*. Она устанавливает элемент данных *m_state* в значение 0. Вызов этой функции-элемента в строке 23 аналогичен вызову функции *turn_on*.

В строчках 13-16 структура описывает функцию-элемент *state*, назначение которой — вернуть текущее состояние. В строке 24 показано, как эта функция может быть использована. Заметим, что свойство *m_state* и функция *state* использованы в операторах присваивания.

Таким образом, описание класса на C++ состоит из элементов данных и функций-элементов. Описание класса может содержать и другие элементы, но эти наиболее важные, поэтому остановимся пока на них.

Как уже упоминалось, данное описание класса является не совсем удачным, хотя и полностью работоспособным. Его работоспособность не уменьшится, если мы удалим из класса все функции-элементы. При этом класс превратится в обычную структуру данных, и ее единственный элемент *m_state* обеспечит необходимую функциональность. Возникает вопрос, — имеет ли смысл выстраивать столь сложную структуру, если более простая обеспечивает нас необходимыми качествами?

Ну, во-первых, перефразируя известное высказывание, заметим, что если классы придумали, значит, это зачем-то нужно. Во-вторых, вопрос о качестве структуры совсем не так однозначен, как может показаться. В-третьих, если рассматривать данный класс как абстракцию, то простая структура данных ей не соответствует — нарушается семантика использования экземпляра класса, то есть объекта. Предполагается, что объект типа «Лампочка» можно включить и выключить. Это действия, которые объект должен «уметь» выполнять (над собой). В структуре без функций эти качества отсутствуют.

Кроме того, нарушается не только внешняя, видимая семантика, но и внутренняя. Используя прямое обращение к свойству *m_state*, мы можем установить его в произвольное значение, не только в значения 0 и 1. Мы даже можем попытаться установить его в недопустимое значение! И в некоторых случаях компилятор этого «не заметит», и свойство примет какое-то произвольное значение. Может быть, в примере с лампочкой это и не имеет особого значения, но ведь это самый простой пример, какой только можно было придумать, чтобы выявить различия между структурами и классами.

Заметим, что функции-элементы, являясь неотъемлемой частью описания класса, *всегда* будут выполнять не только внешнюю, но и внутреннюю семантику класса как абстракции. Это не просто важно, а очень важно и это следует особенно глубоко осознать. Класс как абстракция описывает только внешние границы концепции (феномена, сущности), но не описывает семантику его внутреннего устройства. Класс как тип задает область допустимых значений и операций. Класс как структура определяет реализацию поведения (внешней семантики) в пределах замкнутой области (называемой областью видимости класса). Вследствие этого поведение экземпляров класса определяется не действиями пользователя (того же программиста, заметим), а предписанной реализацией. Это, с одной стороны, гарантирует правильную последовательность операций, выполняемых над элементами данных класса (над элементами данных конкретного экземпляра), а с другой стороны, ограничивает область проектирования и уменьшает сложность разработки.

Рассмотрим теперь второе отличие класса от структуры данных. Оно заключается в возможности поделить описание класса на секции (области), в пределах которых устанавливается определенная степень доступности извне (листинг 2).

***Листинг 2***

```
// класс - описание лампочки                    // 01
struct lamp {                                   // 02
private:                                        // 03
    // хранит состояние лампочки                // 04
    int m_state;                                // 05
public:                                         // 06
    // включает лампочку                        // 07
    void turn_on() {                            // 08
        m_state = 1;                            // 09
    }                                           // 10
    //  выключает лампочку                      // 11
    void turn_off() {                           // 12
        m_state = 0;                            // 13
    }                                           // 14
    // возвращает текущее состояние лампочки    // 15
    int state() {                               // 16
        return m_state;                         // 17
    }                                           // 18
};                                              // 19
// основная функция                             // 20
void main() {                                   // 21
    lamp k;                                     // 22
    //k.m_state = 0;                            // 23
    k.turn_on();                                // 24
    k.turn_off();                               // 25
    int s = k.state();                          // 26
}                                               // 27
```

Для этого описание может содержать спецификаторы доступа:
*public* — открывает общедоступную (открытую) секцию;
*private* — открывает недоступную (закрытую) секцию;
*protected* — открывает секцию ограниченного доступа.

В листинге 2 в закрытую секцию, объявленную в строке 3, попадает переменная *m_state*, поэтому в строке 23 оператор прямого обращения к переменной закомментирован. Доступ к закрытой секции запрещается на уровне компилятора, который фиксирует семантическую ошибку.

В строке 6 начинается открытая секция, в которую попали все функции-элементы. Вызов этих функций ничем не ограничивается, что и демонстрируют строчки кода 24-26. Заметим, что данная реализация класса соответствует графическому изображению абстракции класса, приведенному на рисунке 3.

Спецификатор *protected* описывает защищенную область класса, прямой доступ к которой возможен только из собственно класса и из класса, производного от данного. Иначе говоря, этот спецификатор используется в случае, если класс участвует в отношении обобщения и выступает в качестве базового. В нашем классе его применение возможно, но не имеет особого смысла. Можно заменить спецификатор *private* на спецификатор *protected*, — этот никак не отразится на функциональности данного класса. С точки зрения пользователя объектов класса спецификатор *protected* равнозначен спецификатору *private*, то есть доступ к защищенной секции класса извне запрещен.

В C++ спецификаторы доступа могут быть использованы в произвольных местах описания класса в произвольном порядке и в произвольном количестве. В других языках спецификаторы доступа могут приписываться к каждому элементу класса (то есть являться частью синтаксиса объявлений этих элементов).

Сейчас естественным образом должен возникнуть вопрос о доступе по умолчанию. Иначе говоря, каков будет доступ к элементам класса в случае, если описание класса не содержит никаких спецификаторов. Как мы уже могли видеть в листинге 1, в случае применения описания с ключевым словом *struct*, доступ по умолчанию открытый (*public*). Уместно отметить, что в C++ класс может быть также описан с ключевыми словами *union* и *class*. Доступ по умолчанию для класса типа *union* такой же, как и для описания *struct*. Если же используется ключевое слово *class*, доступ по умолчанию закрытый (*private*).

Использование того или иного ключевого слова (из *struct* или *class*) особого значения не имеет, хотя на практике это часто определяется, исходя из предназначения класса.

В примере (листинг 2) мы можем заменить ключевое слово *struct* на ключевое слово *class*, при этом никаких последствий для класса не наступит, описание класса будет эквивалентным описанию до изменения. Спецификатор *private* при этом потеряет смысл, его можно удалить, но и его присутствие никак не повлияет ни на смысл, ни на результат.

Класс в смысле структуры данных определяет объекты примерно так же, как и обычная структура (*struct*). Напомним, что объект как экземпляр определенного типа занимает некоторую область оперативной памяти. Распределение памяти объекта производится между переменными (элементами данных) структуры в порядке их объявления, независимо от секционирования класса на области различного доступа. Зная устройство класса (в той степени, в какой это показывает описание), в связи с этим всегда можно получить прямой доступ к элементам данных независимо от степени их защищенности (или открытости). В качестве примера рассмотрим листинг 3.

***Листинг 3***

```
class B {                               // 01
    int x, y;                           // 02
public:                                 // 03
    void setxy(int xx, int yy) {        // 04
        x = xx;                         // 05
        y = yy;                         // 06
    }                                   // 07
};                                      // 08
void main() {                           // 09
    B k;                                // 10
    k.setxy(1, -1);                     // 11
    int s = sizeof(k);                  // 12
    int a = *((int*)&k);                // 13
    int b = *(((int*)&k) + 1);          // 14
}                                       // 15
```

Здесь определяется некий класс B, состоящий из двух элементов данных *x* и *y*, и функции *setxy*, которая устанавливает значения переменных класса. Объект класса занимает в памяти область, равную двум размерам типа *int* (строчка 12). Сначала в памяти размещается, в соответствии с порядком объявления, элемент *x*, затем элемент *y*.

В строке 13 в переменную *a* считывается значение элемента *x* объекта *k*, а в строке 14 — в переменную *b* считывается значение элемента *y*, при том, что оба элемента размещены в закрытой секции.

Наверное, сложно сразу сообразить, что написано в строке 13, поэтому поясню: присвоить *a* значение разыменования приведенного к типу «указатель на *int*» адреса объекта *k*. В строке 14 перед разыменованием к адресу объекта прибавляется адресная единица для того, чтобы получить адрес второго элемента.

Как справедливо отмечает Страуструп [3], такое обращение с классом не что иное, как жульничество. Классы защищают нас от ошибок, но не от хакерских приемов. Заметим, что таким образом можно получить доступ к сколь угодно сложной и запутанной защищенной структуре данных, просто придется написать немного больше кода. Кстати, попробуйте вставить строчку 13 из листинга 3 перед строкой 25 листинга 2, и вы получите доступ к переменной *m_state* класса *lamp*.

К сожалению, класс как структура данных не защищает свои элементы во время выполнения кода, что как раз и показывает листинг 3. Защита элементов класса происходит во время компиляции, и об этом нужно знать и помнить. Иначе говоря, классы — конструктивная особенность компилятора, а не внутренняя характеристика типа. Это является одним из недостатков класса, как инструмента реализации объектно-ориентированной модели программирования, нисколько, однако, не умаляющим несомненные достоинства.

Кроме переменных и функций, описание класса может содержать также описание констант, синонимов типов и других, внутренних (вложенных) классов. Константы можно задать с помощью ключевого слова *const*, а также перечислением с помощью ключевого слова *enum*. Листинг 4 показывает, как задаются указанные элементы.

***Листинг 4***

```
class points {                                                    // 01
    static const int max_points = 5;                              // 02
    struct point {                                                // 03
        int x, y, c;                                              // 04
        void move(int a, int b) { x += a; y += b; }               // 05
    };                                                            // 06
    point g[max_points];                                          // 07
public:                                                           // 08
    enum color {RED, GREEN, BLUE};                                // 09
    void move(int a, int b) {                                     // 10
        for (int i = 0; i < max_points; i++) g[i].move(a, b);     // 11
    }                                                             // 12
    void set_color(int n, color c) { g[n].c = c; }                // 13
};                                                                // 14
void main() {                                                     // 15
    points g;                                                     // 16
    g.move(1, 1);                                                 // 17
    g.set_color(0, g.RED);                                        // 18
}                                                                 // 19
```

Здесь константа, объявленная как статическая переменная, задает значение 5, которое нельзя изменить. Вложенный класс *point* можно использовать только внутри класса *points* (независимо от секции доступа). Константы перечисления *color* можно использовать вне класса.

## Класс как область видимости

Под областью видимости понимается область программного текста (модуля), в пределах которой можно обращаться (то есть использовать) ту или иную переменную или функцию, иначе говоря, область, в пределах которой переменная или функция «видна».

Наиболее часто используемыми являются глобальная и локальная видимости. В примере кода, приведенном в листинге 5, приведены две переменные с одинаковым именем *x*, но имеющих разные области видимости.

***Листинг 5***

```
int x = 0;              // 01
void main() {           // 02
    int x = 1;          // 03
    int y = x;          // 04
    int z = ::x;        // 05
}                       // 06
```

В строке 1 определяется переменная *x*, область видимости которой простирается от момента объявления до конца модуля (текущего файла). Говорят, что эта область видимости глобальна (в пределах модуля).

В строке 3, внутри функции *main*, определена переменная с таким же именем *x*, область видимости которой простирается от момента объявления до скобки, закрывающей тело функции. Говорят, что эта область видимости локальна (в пределах функции).

Здесь возникает конфликт имен. В строке 4 переменной *y* присваивается значение переменной *x*, и возникает вопрос, какая переменная *x* имеется в виду, поскольку переменная *y* находится одновременно в двух областях видимости. Все языки программирования решают этот вопрос одинаково, а именно — в пользу локальной области видимости. Иначе говоря, область локальной видимости перекрывает область глобальной видимости (можно сказать, что локальная область видимости имеет б*о*льший приоритет). Таким образом, в нашем примере переменная *y* принимает значение 1, соответствующее значению локальной переменной *x*.

Язык Си (равно как и C++) имеет операцию разрешения глобальной видимости, обозначаемую двойным двоеточием "::". В строке 5 переменной *z* присваивается значение глобальной переменной *x*. Заметим, что не все языки разрешают «видеть» глобальные переменные (равно, впрочем, как и функции).

В языке Си (равно как и в C++), а также в некоторых других, может возникнуть и более сложная ситуация, которую демонстрирует пример кода, приведенный в листинге 6.

***Листинг 6***

```
int x = 0;                  // 01
void main() {               // 02
    int x = 1;              // 03
    int z = ::x;            // 04
    {                       // 05
        int x = 2;          // 06
        int g = x;          // 07
        int h = ::x;        // 08
    }                       // 09
}                           // 10
```

Здесь внутри анонимного блока {} невозможно напрямую определить значение второй, по порядку объявлений, переменой *x*, и переменная *g* принимает значение 2, а переменная *h* — значение 0.

Рассмотрим также пример кода, приведенный в листинге 7.

***Листинг 7***

```
int x = 0;                  // 01
void f(int x) {             // 02
    //int x;                // 03
    int a = x;              // 04
}                           // 05
void main() {               // 06
    f(1);                   // 07
}                           // 08
```

Здесь аргумент функции *f* с именем *x* перекрывает глобальную переменную *x*, то есть обладает локальной областью видимости. В связи с этим, объявление внутри функции *f* переменной с именем *x* недопустимо, поэтому строка 3 закомментирована. В строке 4 переменная *a* принимает значение аргумента функции *x*.

Класс определяет свою собственную область, называемую областью видимости класса. Это область, в пределах которой видны все элементы класса, то есть его переменные и функции.

Рассмотрим листинг 8.

***Листинг 8***

```
int x = 2;                          // 01
struct A {                          // 02
    void set_x(int x) {             // 03
        this->x = x;                // 04
        mod();                      // 05
    }                               // 06
    int get_x() { return x; }       // 07
private:                            // 08
    void mod()  { x %= ::x; }       // 09
    int x;                          // 10
};                                  // 11
```

Класс A определяет два закрытых элемента и два открытых. Область видимости класса заключена между скобками {}, в пределах которой все элементы класса «видны» всем его функциям независимо от расположения и степени доступа. Например, закрытая функция *mod* видна внутри функции *set_x*, а закрытая переменная *x* видна в функции *mod*, хотя объявление *x* расположено ниже по тексту.

Заметим, что в области видимости класса никакой элемент класса невозможно перекрыть глобальными или локальными объектами, расположенными вне класса, в том числе и аргументами функций. Если в каком-то месте класса возникает конфликт имен, элемент класса будет «виден» через специальный указатель *this* (строка 4). Об указателе *this* подробнее будет рассказано ниже.

В случае, если реализация функции-элемента располагается вне описания интерфейса класса (а это возможно и очень часто применяется), используется оператор разрешения видимости "::".

Рассмотрим листинг 9, описывающий тот же класс, в котором реализация функций-элементов вынесена из описания класса.

***Листинг 9***

```
// Описание класса          // 01
int x = 2;                  // 02
struct A {                  // 03
    void set_x(int x);      // 04
    int get_x();            // 05
private:                    // 06
    void mod();             // 07
    int x;                  // 08
};                          // 09
// Реализация функций       // 10
void A::set_x(int x) {      // 11
    this->x = x;            // 12
    mod();                  // 13
}                           // 14
int A::get_x() {            // 15
    return x;               // 16
}                           // 17
void A::mod() {             // 18
    x %= ::x;               // 19
}                           // 20
```

Как видим, функции-элементы, реализация которых находится вне описания класса, должны быть описаны с операцией разрешения видимости, которой предшествует имя класса (на цветном изображении выделено красным цветом, строки 11, 15 и 18). Видимость в этом случае начинается сразу после операции "::", и простирается до скобки, закрывающей тело функции.

Если вне класса определяется функция-элемент вложенного класса, то следует использовать более длинную квалификацию имени. Например, чтобы определить вне класса функцию-элемент *move* класса *point* из листинга 4, нужно применить операцию разрешения видимости сначала для класса *points*, а затем для класса *point* (листинг 10).

***Листинг 10***

```
void points::point::move(int a, int b) {
    x += a;
    y += b;
}
```

## Указатель this

Классы описывают типы данных, а использование классов требует создания объектов, то есть экземпляров классов. Объект состоит только из элементов данных, описанных в классе (на самом деле объект может содержать также дополнительные указатели, но для наших рассуждений сейчас это не важно). Функции-элементы, описанные в классе, не принадлежат объекту, — они принадлежат классу. Иначе говоря, функции-элементы существуют в единственном числе, независимо от количества действующих экземпляров класса. Должен возникнуть естественный вопрос, — каким образом функция-элемент «видит» элементы данных конкретного объекта?

Здесь возникает очень интересная ситуация. Пусть есть некий класс, который описывает один элемент данных. Пусть мы создали несколько объектов этого класса. Тогда каждый объект, по определению обладая свойством идентичности, отличается от других объектов значением своего элемента данных. На самом деле, разумеется, значения элемента данных разных, и даже всех, объектов могут быть одинаковыми, и, когда мы говорим об идентичности объектов, на самом деле мы говорим об их различии в смысле разных областей памяти. Идентичность здесь понимается именно как идентификация сущности (два совершенно одинаковых стакана, будучи представителями класса стаканов, идентичны в смысле существования двух разных физических феноменов, а вовсе не в смысле их кажущейся внешней одинаковости; они различаются, например, конкретными молекулами, из которых, как говорят, они состоят).

Пусть теперь в классе определена функция, которая изменяет состояние элемента данных. В коде функции мы использует оператор присваивания, который задает новое значение переменной класса, которая, как следует из вышеприведенного изложения, является абстракцией! В классе нет никакого указания на то, что данная переменная идентична (!) для конкретного объекта.

Противоречие здесь заключается в том, что абстракция сущности обладает поведением, и на самом деле поведение принадлежит объекту (в смысле, что каждый, например, человек, умеет ходить лично, а не абстрактно). Однако реализация класса отделяет умение сущности выполнять действия от самого этого объекта, и это представляется вполне естественным: человек как абстракция умеет ходить в принципе, вообще, а не в смысле того, что каждый ходит по-своему.

Упрощая ситуацию, зададим следующий вопрос: как конкретная функция «видит» абстрактный элемент данных конкретного объекта?

Прежде давайте посмотрим, как можно «заставить» отдельно стоящую функцию «увидеть» элементы конкретного экземпляра структуры данных в языке, например, Си (листинг 11).

***Листинг 11***

```
struct lamp {                  // 01
    int m_state;               // 02
};                             // 03
void turn_on(lamp * o) {       // 04
    o->m_state = 1;            // 05
}                              // 06
void main() {                  // 07
    lamp x, y;                 // 08
    turn_on(&x);               // 09
    turn_on(&y);               // 10
}                              // 11
```

Здесь для изменения состояния объекта типа структуры *lamp* используется функция *turn_on* (можно, естественно, аналогичным образом определить и функцию *turn_off*).

Как видим, для получения доступа к абстрактному элементу структуры *lamp* в функцию просто-напросто передается указатель на соответствующий объект (объект *x* в строке 9 или объект *y* в строке 10).

В классах все именно так и сделано, на уровне компилятора и на уровне объектного кода. Любая не статическая функция-элемент класса независимо от нашего желания и незаметно для нас (говорят, — неявно) первым своим аргументом (независимо от числа аргументов) получает указатель на тот объект, с которым она синтаксически вызвана.

Этот указатель доступен внутри области видимости класса под именем *this* (в языке C++; в других языках он может именоваться иначе, например, *self* в языке *Turbo Pascal*, *Me* в языке *Visual Basic*) и определяется как константный (то есть его нельзя изменить).

Фактически, когда в описании класса осуществляется доступ к элементу данных класса, на самом деле неявно компилируется доступ через указатель *this* на область памяти объекта.

Рассмотрим листинг 12.

***Листинг 12***

```
class lamp {                        // 01
    int m_state;                    // 02
public:                             // 03
    void turn_on() {                // 04
        this->m_state = 1;          // 05
    }                               // 06
    void turn_off() {               // 07
        this->m_state = 0;          // 08
    }                               // 09
    void state() {                  // 10
        return this->m_state;       // 11
    }                               // 12
};                                  // 13
```

Здесь в операторе присваивания в строке 5 и в строке 8 а также в выражении в строке 11 явным образом используется указатель на объект *this*. Нет особой необходимости делать это, поскольку компилятор подставляет указатель автоматически, однако в некоторых случаях это необходимо (например, в случае конфликта имен переменных или функций класса, как было показано ранее в листинге 8).

Указатель на объект существенно необходим также в случаях, когда семантика функции-элемента подразумевает возвращаемое значение функции в виде самого этого объекта (то есть ссылки на вызывающий объект). В качестве примера рассмотрим класс, описание которого приведено в листинге 13.

***Листинг 13***

```
class positive {            // 01
    int pos;                // 02
public:                     // 03
    void set(int a) {       // 04
        pos = a;            // 05
    }                       // 06
    int get() {             // 07
        return pos;         // 08
    }                       // 09
    void add(int a) {       // 10
        pos += a;           // 11
    }                       // 12
};                          // 13
void main() {               // 14
    positive x;             // 15
    x.set(0);               // 16
    x.add(1);               // 17
    x.add(2);               // 18
    x.add(3);               // 19
    int y = x.get();        // 20
}                           // 21
```

Здесь мы видим функцию *add*, которая добавляет некоторое значение к уже существующему. В основной функции к объекту *x* добавляется какое-то значение три раза подряд. Если изменить семантику функции *add*, то мы можем получить некоторый выигрыш (листинг 14).

***Листинг 14***

```
class positive {                                    // 01
    int pos;                                        // 02
public:                                             // 03
    void set(int p) {                               // 04
        pos = p;                                    // 05
    }                                               // 06
    int get() {                                     // 07
        return pos;                                 // 08
    }                                               // 09
    positive & add(int p) {                         // 10
        pos += p;                                   // 11
        return *this;                               // 12
    }                                               // 13
};                                                  // 14
void main() {                                       // 15
    positive x;                                     // 16
    x.set(0);                                       // 17
    int y = x.add(1).add(2).add(3).get();           // 18
}                                                   // 19
```

Следует обратить внимание на объектную запись в строке 18. Она, как мне кажется, выглядит лаконично и понятно.

## Конструктор

Одна из целей класса как типа — обеспечение целостности типа, то есть его элементов данных. Это означает, что в любой момент существования объекта элементы данных должны принимать только допустимые значения. В качестве примера рассмотрим класс, описывающий рациональное число (листинг 15).

***Листинг 15***

```
class rat {                             // 01
    int num, den;                       // 02
public:                                 // 03
    void set(int n, int d) {            // 04
        num = n;                        // 05
        den = d;                        // 06
    }                                   // 07
    int get_num() { return num; }       // 08
    int get_den() { return den; }       // 09
};                                      // 10
```

Поскольку рациональное число есть дробь, очевидно, что знаменатель, то есть элемент данных *den*, не должен принимать нулевого значе

ния. Однако если мы создадим объект этого класса, то сразу после конструирования объекта значения его элементов данных неопределённы (имеют произвольные значения). Чтобы установить правильные значения, в класс инкапсулирована функция *set*. Однако у этой функции есть два недостатка.

Первый — показанная реализация функции допускает недопустимое значение знаменателя. Это наше упущение, которое легко исправить.

Второй недостаток хуже. Предполагается, что пользователь класса обязан вызвать функцию *set*, чтобы проинициализировать объект. Таким образом, контроль за соблюдением целостности типа в нашей реализации класса возлагается на пользователя и противоречит понятию типа.

С встроенными типами данных все проще. Какое бы произвольное значение не приняла переменная типа *int*, оно будет всегда допустимым в соответствии с определением типа. Поэтому нам не приходится особенно задумываться об инициализации встроенных типов, разве что о присвоении им каких-то начальных значений.

С классами не так. Поскольку в них элемент данных описывает значения, часто не совпадающие с областью существования описывающего элемент стандартного типа, нужен встроенный в класс механизм, позволяющий выполнить начальную инициализацию.

Этот механизм существует и называется конструктором класса. Конструктор — это специальная функция класса, которая вызывается автоматически во время создания экземпляра. В языке C++ конструктор имеет имя, совпадающее с именем класса (с точностью до регистра). В новой реализации класса *rat* имеется конструктор (листинг 16).

***Листинг 16***

```
class rat {                                  // 01
    int num, den;                            // 02
public:                                      // 03
    rat() {                                  // 04
        num = 0;                             // 05
        den = 1;                             // 06
    }                                        // 07
    void set(int n, int d) {                 // 08
        num = n;                             // 09
        (d == 0) ? den = 1 : den = d;        // 10
    }                                        // 11
    int get_num() { return num; }            // 12
    int get_den() { return den; }            // 13
};                                           // 14
```

Конструктор определяется в строчках 4-7. Заметим, что конструктор не имеет и не может иметь возвращаемого значения (даже *void*). У конструктора нет аргументов, и он конструирует объект со значением "0/1" (то есть нулевой объект).

Класс может иметь несколько конструкторов. В этом случае они должны различаться наборами аргументов. Рассмотрим другой вариант класса рациональных чисел (листинг 17).

***Листинг 17***

```
class rat {                                  // 01
    int num, den;                            // 02
public:                                      // 03
    // конструктор по умолчанию              // 04
    rat(int n = 0) {                         // 05
        num = n;                             // 06
        den = 1;                             // 07
    }                                        // 08
    // конструктор с параметрами             // 09
    rat(int n, int d) {                      // 10
        num = n;                             // 11
        (d == 0) ? den = 1 : den = d;        // 12
    }                                        // 13
    // конструктор приведения                // 14
    rat(double n) {                          // 15
        num = (int)n;                        // 16
        den = 1;                             // 17
    }                                        // 18
    void set(int n, int d) {                 // 19
        num = n;                             // 20
        (d == 0) ? den = 1 : den = d;        // 21
    }                                        // 22
    int get_num() { return num; }            // 23
    int get_den() { return den; }            // 24
};                                           // 25
void main() {                                // 26
    rat a, b(2), c(3, 2), d(4.0);            // 27
}                                            // 28
```

В классе три конструктора. Первый похож на конструктор класса в листинге 16, за исключением того, что он имеет аргумент со значением по умолчанию. Иначе говоря, вызов этого конструктора не требует задания какого-либо параметра. Такой конструктор называется *конструктором по умолчанию* (*default ctor*). Конструктор по умолчанию требуется тогда, когда объект конструируется неявно (не по желанию программиста, а компилятором). В основной функции *main* с помощью этого конструктора создается объект *a* со значением 0/1.

Второй конструктор имеет два параметра и называется, соответственно, конструктором с параметрами. Цель этого конструктора — создать объект с определенными (не нулевыми) значениями элементов данных. Можно сказать, что действие конструктора с параметрами равнозначно инициализации переменной стандартного типа при ее объявлении. В основной функции этот конструктор создает объект *c* со значением 3/2.

Третий конструктор имеет один параметр и называется конструктором приведения или конструктором преобразования (его можно назвать также и конструктором с параметром). Назначение этого конструктора заключается в приведении значения типа *double* к типу класса. С его помощью в основной функции создается объект *d* со значением 4/1.

Заметим, что конструктор по умолчанию, имеющий один аргумент со значением по умолчанию, является также конструктором приведения. В нашем случае он конструирует объект из типа *int*, то есть приводит тип *int* к типу *rat*. В основной функции с помощью этого конструктора создается объект *b* со значением 2/1.

Можно было бы добавить в класс *rat* еще несколько конструкторов, в основном выполняющих приведение к типу класса, но особой необходимости в этом нет. Нужно также помнить о том, что компилятор и сам умеет приводить одни типы к другим. Так, если бы в классе не было конструктора приведения типа *double* к типу класса, его функции выполнял бы конструктор по умолчанию, при этом компилятор выполнил бы приведение значения типа *double* к типу аргумента *int*. Кроме того, программист может указать это приведение типа явным образом.

Заметим, что конструктор с количеством параметров более одного можно назвать конструктором преобразования — он преобразует несколько типов данных в тип класса.

Цель конструктора — инициализировать начальные значения элементов данных. Это может быть сделано как в коде конструктора, так и в специальном списке, называемом *списком инициализации*. Рассмотрим листинг 18.

***Листинг 18***

```
class rat {                                  // 01
    int num, den;                            // 02
public:                                      // 03
    rat(int n = 0) : num(n), den(1) {}       // 04
};                                           // 05
```

Список инициализации расположен после двоеточия и перед блоком, описывающим пустое тело конструктора. Он является особенностью языка C++. Список инициализации иногда является единственным способом инициализации объекта в C++.

Заметим, что инициализация в списке инициализации соответствует объявлению переменной с начальным значением, а инициализация в теле конструктора — выполнению операции присваивания.

Как и любая функция, конструктор может быть размещен в любой секции класса. Если конструктор размещается в закрытой или защищенной секции, вызвать его нельзя, то есть нельзя создать объект с его по

мощью. Защищенные конструкторы используются обычно при создании объектов производных классов. Если все конструкторы класса закрытые или защищенные, создавать объект класса запрещено. Иногда конструкторы помещают в закрытую секцию класса, чтобы запретить выполнение некоторых операций, выполняемых с помощью конструкторов.

## Деструктор

Как следует из названия, деструктор служит для разрушения объекта класса. Деструктор — это специальная функция класса, которая вызывается автоматически при уничтожении объекта. Как и конструктор, деструктор не может иметь возвращаемого значения, а в отличие от конструктора, деструктор не может иметь аргументов, и, соответственно, в классе может быть определен только один деструктор.

Имя деструктора в языке C++ совпадает с именем класса, которому предшествует знак "~" (тильда). Поскольку тильда соответствует операции дополнения, имя деструктора подчеркивает, что деструктор является как бы дополнением конструктора.

Рассмотрим пример класса, приведенный в листинге 19.

***Листинг 19***

```
class string {                              // 01
    char * p;                               // 02
public:                                     // 03
    string(char * s) {                      // 04
        p = new char[1 + strlen(s)];        // 05
        strcpy(p, s);                       // 06
    }                                       // 07
    void set(char * s) {                    // 08
        strcpy(p, s);                       // 09
    }                                       // 10
};                                          // 11
void main() {                               // 12
    string a("abc"), b("yz");               // 13
    b = a;                                  // 14
    a.set("qwe");                           // 15
}                                           // 16
```

Здесь элемент данных *p* является указателем. Конструктор приведения выделяет динамическую память и копирует в нее параметр *s*.

В основной функции создается два объекта. Объект *a* выделяет динамическую память для строки "*abc*", объект *b* — для строки "*yz*". В строке 12 объект *a* копируется в объект *b*. Затем в строке 15 изменяется значение объекта *a*.

Рисунок 10 поможет нам понять, что же при этом на самом деле происходит (показано распределение памяти).

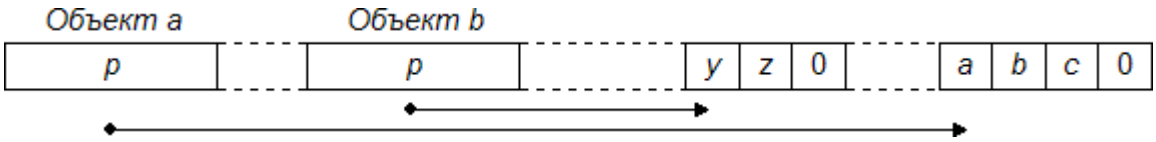


Рисунок 10

При создании объекта *a* его указатель *p* указывает на область динамической памяти, в которой выделяется буфер для строки "*abc*". Аналогично при создании объекта *b* его указатель *p* указывает на область динамической памяти, в которой размещается буфер для строки "*yz*".

Поскольку класс не определяет, как выполнять присваивание одного объекта другому, компилятор выполняет так называемое «побитовое копирование», при котором область памяти одного объекта копируется «как есть» в область памяти другого.

В нашем случае область объекта *a* копируется поверх области объекта *b*. При этом теряется указатель на буфер "*yz*", и происходит «утечка памяти» (указатель на буфер "*yz*" потерян, и вернуть память эту память обратно не удастся). Кроме того, оба объекта указывают теперь на одну и ту же область динамической памяти, то есть на буфер "*abc*". Если каким-либо образом изменить значение буфера (то есть объекта) *a*, то одновременно изменится и значение объекта *b* (строка 15).

Рассмотрим теперь, что происходит в момент завершения основной функции. Переменные *a* и *b* выходят из области видимости и уничтожаются. При этом значения указателей *p* теряются, и происходит утечка памяти для буфера "*qwe*". В итоге оба буфера в динамической памяти утеряны. Вот почему нам нужен деструктор.

Рассмотрим листинг 20, новую редакцию класса *string*.

***Листинг 20***

```
class string {                          // 01
    char * p;                           // 02
public:                                 // 03
    string(char * s) {                  // 04
        p = new char[1 + strlen(s)];    // 05
        strcpy(p, s);                   // 06
    }                                   // 07
    ~string() {                         // 08
        delete[] p;                     // 09
    }                                   // 10
};                                      // 14
void main() {                           // 15
    string a("abc"), b("yz");           // 16
    b = a;                              // 17
}                                       // 18
```

Деструктор описывается в строчках 08-10. Как видим, он возвращает динамическую память при помощи операции *delete*[]. Квадратные скобки здесь указывают на то, что мы освобождаем массив.

Рассмотрим, что же теперь происходит в нашей программе.

Как и раньше, при присвоении объекту *b* значения объекта *a* происходит утечка памяти (теряется буфер "*yz*"). В момент завершения основной функции сначала вызывается деструктор для объекта *b*. Он освобождает память буфера "*abc*". Затем вызывается деструктор для объекта *a*, и он также пытается освободить память для буфера "*abc*", которая уже освобождена. Возникает исключительное состояние, и дальнейшие действия программы зависят от компилятора. Таким образом, получилось даже хуже, чем было. Раньше наша программа плохо, но работала. Теперь же она совсем не работает.

Это вовсе не означает, что деструктор следует убрать из класса. Деструктор в данном классе необходим. Однако при проектировании классов, в которых используются динамически выделяемые области памяти и указатели на них, необходимо очень четко определить все возможные операции, которые допустимо выполнять над объектами. Как мы убедились, операция присваивания одного объекта класса другому вызывает неправильное поведение программы. У нас есть два способа избежать проблем с присваиванием. Первый — описать порядок выполнения операции присваивания. Второй — запретить эту операцию. Как это сделать — тема следующего раздела.

Вернемся к листингу 19 и обратим внимание на описанную в классе операцию установки нового значения *set*. Мы инкапсулировали эту функцию для того, чтобы продемонстрировать факт побитового копирования указателя *p* объекта *a* в объект *b*. При этом мы не задумывались над тем, насколько правильно эта функция работает. Заметим, что в некоторых случаях такая функция будет допустима, однако для класса из листинга 19 — нет.

Предположим, что в строке 15 листинга 19 мы зададим новое значение строки для объекта *a*, равное, например, "*qwerty*". Поскольку длина новой строки превышает размер буфера, выделенного для строки "*abc*", вполне вероятно, что новая строка «перепишет» область динамической памяти, занимаемую каким-то другим объектом программы. При этом в лучшем случае программа перестанет работать правильно, в худшем — компьютер остановится или вообще сломается.

Смысл функции *set* заключается в том, чтобы установить новое значение строки объекта. Можно заметить, что операция присваивания делает то же самое. Иначе говоря, код этой функции должен показывать, как правильно повторно инициализировать объект данного класса.

Рассмотрим листинг 21.

***Листинг 21***

```
class string {                              // 01
    char * p;                               // 02
public:                                     // 03
    string(char * s) {                      // 04
        p = new char[1 + strlen(s)];        // 05
        strcpy(p, s);                       // 06
    }                                       // 07
    void set(char * s ) {                   // 08
        delete[] p;                         // 09
        p = new char[1 + strlen(s)];        // 10
        strcpy(p, s);                       // 11
    }                                       // 12
    char * get() {                          // 13
        return p;                           // 14
    }                                       // 15
    ~string() {                             // 16
        delete[] p;                         // 17
    }                                       // 18
};                                          // 19
void main() {                               // 20
    string a("abc"), b("yz");               // 21
    b.set(a.get());                         // 22
}                                           // 23
```

Здесь операция *set* сначала освобождает уже существующий буфер, а затем формирует его заново и копирует в него новую строку. В класс добавлена также операция *get*, которая возвращает указатель на буфер объекта. В строке 22 мы использовали ее для того, чтобы присвоить объекту *b* значение объекта *a*. Таким образом, с помощью двух дополнительных функций мы добились желаемого результата. Программа работает вполне корректно, хотя конструкция в строке 22 не очень привлекательна. Заметим также, что ничто не запрещает программисту-пользователю класса написать присваивание объекта *a* объекту *b*, и при этом все проблемы с памятью возобновятся.

## Операция присваивания

В языке C++ существует возможность переопределять многие из операций, которые могут выполняться над стандартными типами. Это переопределение называется перегрузкой операций и более подробно рассматривается далее. Здесь же мы остановимся только на операции присваивания, поскольку, как мы убедились, она действительно важна для решения наших проблем.

Перегрузка операции присваивания заключается в описании действий с помощью специальной функции класса с именем *operator*=. Рассмотрим листинг 22.

***Листинг 22***

```
class string {                                  // 01
    char * p;                                   // 02
public:                                         // 03
    string(char * s) {                          // 04
        p = new char[1 + strlen(s)];            // 05
        strcpy(p, s);                           // 06
    }                                           // 07
    ~string() {                                 // 08
        delete[] p;                             // 09
    }                                           // 10
    string & operator=(string & s) {            // 11
        delete[] p;                             // 12
        p = new char[1 + strlen(s.p)];          // 13
        strcpy(p, s.p);                         // 14
        return *this;                           // 15
    }                                           // 16
    string & operator=(char * s) {              // 17
        delete[] p;                             // 18
        p = new char[1 + strlen(s)];            // 19
        strcpy(p, s);                           // 20
        return *this;                           // 21
    }                                           // 22
};                                              // 23
void main() {                                   // 24
    string a("abc"), b("yz"), c(""), d("");     // 25
    c.operator=(b);                             // 26
    b = a;                                      // 27
    d = b = "xyz";                              // 28
}                                               // 29
```

В классе *string* здесь определены целых две реализации операции присваивания. Первая операция позволяет присваивать объекту класса другой объект этого класса, а вторая операция — присваивать объекту класса строковый литерал. Соответственно, операции присваивания имеют аргументы различных типов.

Обратим внимание на строку 26. В ней показано, как можно использовать функцию операции присваивания как функцию. В строке 27 показано, как используется та же самая операция в более наглядном виде. Наконец, в строке 28 показано, как объекту *b* присваивается значение строкового литерала. Кроме того, поскольку операции присваивания возвращают ссылку на класс, можно выполнять многократное присваивание в одном операторе, что и демонстрирует строка 28. Сначала в ней, как было сказано, объекту *b* присваивается значение строкового литерала, а затем значение объекта *b* присваивается объекту *d*.

Заметим, что операции присваивания могут быть описаны как возвращающие тип *void*. Это можно сделать, однако в этом случае многократное присваивание будет недопустимо, и в строке 28 компилятор зафиксирует синтаксическую ошибку.

Несмотря на то, что класс *string* в листинге 22 кажется нам теперь вполне совершенным, в нем есть одна серьезная ошибка. Стоит попробовать выполнить присваивание объекта самому себе, например, *a* = *a*.

Трассирование выполнения этой операции убедит нас, что объект *a* в этом случае будет разрушен, хотя исключительной ситуации может и не возникнуть. Это происходит потому, что первая строчка операции присваивания из объекта класса возвращает буфер, который затем должен быть заново выделен и заполнен. Поэтому операция присваивания из объекта класса должна производить проверку на само-присвоение. Листинг 23 показывает, как это может быть сделано.

***Листинг 23***

```
class string {                              // 01
    char * p;                               // 02
public:                                     // 03
    string(char * s) {                      // 04
        p = new char[1 + strlen(s)];        // 05
        strcpy(p, s);                       // 06
    }                                       // 07
    ~string() {                             // 08
        delete[] p;                         // 09
    }                                       // 10
    string & operator=(string & s) {        // 11
        if (this == & s) return *this;      // 12
        delete[] p;                         // 13
        p = new char[1 + strlen(s.p)];      // 14
        strcpy(p, s.p);                     // 15
        return *this;                       // 16
    }                                       // 17
};                                          // 18
void main() {                               // 19
    string a("abc");                        // 20
    a = a = a;                              // 21
}                                           // 22
```

Проверка на само-присвоение производится в строке 12. В этом случае не выполняется никаких действий и просто возвращается ссылка на объект. Строка 21 призвана продемонстрировать безопасность операции присваивания.

Если теперь строчку 12 из листинга 23 вставить перед строкой 12 из листинга 22, то получится действительно хороший класс, позволяющий выполнять некоторые базовые операции со строками. К недостаткам этого класса можно отнести теперь отсутствие конструктора по умолчанию, который конструировал бы объект, описывающий пустую строку. Это дало бы возможность упростить описание объектов *c* и *d* в строке 25 листинга 22.

Как было сказано, можно также запретить присваивание. Например, мы не хотим разбираться с проблемами, которые при этом возникают.

Чтобы запретить операцию присваивания, достаточно поместить ее описание в закрытую секцию класса (листинг 24).

***Листинг 24***

```
class string {                              // 01
    char * p;                               // 02
    string & operator=(string & s);         // 03
public:                                     // 04
    string(char * s) {                      // 05
        p = new char[1 + strlen(s)];        // 06
        strcpy(p, s);                       // 07
    }                                       // 08
    ~string() {                             // 09
        delete[] p;                         // 10
    }                                       // 11
    char * operator=(char * s) {            // 12
        delete[] p;                         // 13
        p = new char[1 + strlen(s)];        // 14
        strcpy(p, s);                       // 15
        return p;                           // 16
    }                                       // 17
};                                          // 18
void main() {                               // 19
    string a(""), b("");                    // 20
    b = a = "xyz";                          // 21
}                                           // 22
```

Как видим, здесь операция присваивания из объекта класса размещена в закрытой секции класса и реализации этой функции не представлено (она никогда не потребуется). Однако в классе определена операция присваивания из строкового литерала, которая допускает многократное присваивание, для чего возвращаемый тип этой функции заменен на тип строкового литерала и соответственно изменено возвращаемое значение. Получился довольно интересный класс.

В данном классе можно вообще не описывать операцию присваивания из строкового литерала. В этом случае использование операции присваивания было бы запрещено совсем. Проверить данное утверждение вы можете самостоятельно.

## Конструктор копии

Есть еще одна важная специальная функция класса, которая конструирует копию объекта класса в некоторых случаях. Конструктор копии вызывается компилятором автоматически каждый раз, когда требуется копия объекта. Это происходит в трех случаях:

- при инициализации объекта класса другим объектом этого класса;
- при передаче объекта в функцию по значению;
- при возврате объекта из функции.

Сигнатуру конструктора копии иногда обозначают X(X&), что означает, что конструктор копии имеет один аргумент типа ссылки на класс (подразумевается, что X — имя класса).

Рассмотрим листинг 25.

***Листинг 25***

```
class string {                              // 01
    char * p;                               // 02
    string & operator=(string & s);         // 03
public:                                     // 04
    string(char * s) {                      // 05
        p = new char[1 + strlen(s)];        // 06
        strcpy(p, s);                       // 07
    }                                       // 08
    ~string() {                             // 09
        delete[] p;                         // 10
    }                                       // 11
    string(string & s) {                    // 12
        p = new char[1 + strlen(s.p)];      // 13
        strcpy(p, s.p);                     // 14
    }                                       // 15
    string & operator=(char * s) {          // 16
        delete[] p;                         // 17
        p = new char[1 + strlen(s)];        // 18
        strcpy(p, s);                       // 19
        return *this;                       // 20
    }                                       // 21
};                                          // 22
string foo(string e) {                      // 23
    e = "xyz";                              // 24
    return e;                               // 25
}                                           // 26
void main() {                               // 27
    string a("abc");                        // 28
    string b = a;                           // 29
    string c(a);                            // 30
    string d = foo(a);                      // 31
}                                           // 32
```

Конструктор копии здесь определен в строчках 12-15. Как видим, он похож на конструктор приведения из типа *char**, однако использует объект класса в качестве инициализирующего значения.

Приведенный пример достаточно сложный и требует внимательного изучения. Следует понимать, когда вызывается конструктор копии. В одних случаях он вызывается явно (при применении знака присваивания во время инициализации), в других — неявно (компилятором при передаче копии объекта в функцию и при возврате объекта из функции).

Конструктор копии вызывается в данном примере 4 раза. Первый раз он вызывается при конструировании объекта *b* в строке 29. Заметим, что здесь используется не операция присваивания, хотя и использован знак

этой операции. Второй раз конструктор копии вызывается при конструировании объекта *c* в строке 30.

В строке 31 демонстрируется неявное конструирование двух объектов при помощи конструктора копии. Во время вызова функции *foo* (на самом деле непосредственно перед вызовом) в строке 31 сначала конструируется новый объект, который инициализируется объектом, передаваемым функции в качестве параметра. При этом объект *e*, используемый как локальная переменная функции *foo*, отделяется от объекта *a*, который был передан в функцию как параметр.

Внутри функции *foo* локальному объекту *e* присваивается новое значение "*xyz*", после чего снова вызывается конструктор копии, который конструирует новый объект из локального объекта *a*, и этот новый объект возвращается из функции. В основной функции в строке 31 возвращаемый объект присваивается переменной *d*. Перед завершением функции *foo* локальный объект *e* уничтожается деструктором.

Заметим, что изменение программы всего на один знак приведет к совершенно другому результату.

Пусть аргумент функции *foo* имеет тип *string*&. Тогда во время вызова функции *foo* аргумент *e* станет синонимом внешней (по отношению к функции) переменной *a*. При изменении значения объекта *e* одновременно изменится и значение внешнего объекта *a*. Конструктор копии для конструирования объекта *e* не вызывается (так как никакой новый объект не конструируется), и деструктор для его разрушения также не вызывается. После вызова функции *foo* объект *d* является копией объекта *a* и оба объекта имеют значение "*xyz*" (при этом для создания возвращаемого объекта вызывается конструктор копии).