А.А. Богуславский, С.М. Соколов

Основы программирования на языке Си++

Часть III. Объектно-ориентированное программирование на языке Си++

(для студентов физико-математических факультетов педагогических институтов)

Коломна, 2002

ББК 32.97я73
УДК 681.142.2(075.8)
Б 73

Рекомендовано к изданию
редакционно-издательским советом
Коломенского государственного
педагогического института

Богуславский А.А., Соколов С.М.
Б73 Основы программирования на языке Си++: Для студентов физико-математических факультетов педагогических институтов. – Коломна: КГПИ, 2002. – 490 с.

Пособие предназначено для обучения студентов, обладающих навыками пользовательской работы на персональном компьютере, основным понятиям и методам современного практического программирования. Предметом изучения курса является объектно-ориентированное программирование на языке Си++ в среде современных 32-х разрядных операционных систем семейства Windows. Программа курса разбита на 4 части: (1) Введение в программирование на языке Си++; (2) Основы программирования трехмерной графики; (3) Объектно-ориентированное программирование на языке Си++ и (4) Программирование для Microsoft Windows с использованием Visual C++ и библиотеки классов MFC.

После изучения курса студент получает достаточно полное представление о содержании современного объектно-ориентированного программирования, об устройстве современных операционных систем Win32 и о событийно-управляемом программировании. На практических занятиях вырабатываются навыки программирования на Си++ в интегрированной среде разработки Microsoft Visual C++ 5.0.

Рецензенты:

И.П. Гиривенко – к.т.н., доцент, зав. кафедрой информатики и вычислительной техники Рязанского государственного педагогического университета им. С.А. Есенина.

А.А. Шамов – к.х.н., доцент кафедры теоретической физики Коломенского государственного педагогического института.

СОДЕРЖАНИЕ

Введение

Эта часть учебного курса "Основы программирования на языке Си++" предназначена для первоначального знакомства с методами объектно-ориентированного программирования (ООП) на языке Си++. В первых лекциях кратко рассматривается история развития ООП и наиболее важные свойства объектно-ориентированного проектирования (проектирование на основе распределения обязанностей), которое является необходимым условием эффективной разработки программного обеспечения с использованием ООП. Далее основное внимание уделяется введению понятий классов, объектов, пересылки сообщений и наследования и соответствующим синтаксическим особенностям Си++. Множественное наследование не рассматривается ввиду сложности вопроса и отсутствия подходящих примеров, пригодных для использования в процессе начального изучения программирования на Си++.

На практических занятиях используется среда разработки **Microsoft Visual C++** на ПК под управлением **Windows 95/98/NT**. Все программы, рассматриваемые в качестве примеров в лекциях, и ответы к упражнениям написаны на стандартном ANSI Си++ и проверены в среде **Microsoft Visual C++ 5.0** на ПК под управлением **Windows 98**.

Лекция 1. Основные понятия ООП

1. Появление объектно-ориентированных языков программирования

Первые объектно-ориентированные языки программирования появились в конце 60-х гг. (Симула-67, Смоллток), но быстрое развитие технологии ООП, в результате которого эта технология стала общепринятым стандартом, началось лишь в середине 80-х гг. Побуждающим мотивом для развития стала постоянно возрастающая сложность программного обеспечения, с которой все хуже справлялись традиционные средства процедурного программирования.

Большинство программистов знают процедурные языки вроде Си и Паскаля, поэтому особую популярность завоевали не специальные ОО-языки (Смоллток, CLOS), а ОО-расширения уже известных языков – Си++ и Объектный Паскаль. Тот факт, что эти языки основаны на хорошо известных языках, создает иллюзию о простоте усвоения ООП программистами, знающими процедурные языки.

Язык Си++ очень быстро развивался, начиная с середины 80-х гг. (первая версия была разработана Бьярном Страуструпом в 1979 г.). В 1994 г. комитетом по стандартизации ANSI/ISO был принят стандарт языка Си++. В начале 90-х гг. на основе языка Си++ был разработан язык Java, предназначенный для написания программ для Интернет.

Язык Объектный Паскаль был особенно популярен, пока фирма Apple применяла его в качестве основного языка программирования для компьютеров Macintosh, но потом утерял этот статус. Объектный Паскаль продолжает применяться на IBM-совместимых ПК в составе среды быстрой разработки программ Inprise Delphi.

2. Причины популярности ООП

Возлагается надежда, что метод ООП упростит написание более сложных программ. Парадоксальность ООП заключается в том, что:

1) это новая идея о том, что собственно называется вычислениями, а также того, как мы можем структурировать информацию в памяти компьютера;
2) это эволюционный шаг, естественным образом вытекающий из предшествующей истории развития средств абстракции данных.

Можно выделить три основных причины популярности ООП:

1) надежда, что ООП приведет к быстрому росту продуктивности программистов и повышению надежности программ (решение "кризиса программного обеспечения");
2) желание перейти от существующих языков к новой технологии;
3) сходство с методами проектирования, применяющимися в других инженерных областях (сборка изделия из готовых блоков).

Под термином "кризис программного обеспечения" подразумевается то, что в научных, технических и экономических приложениях требуется разрабатывать все более сложные программные системы, требующие усилий большого количества программистов, которых сложно координировать. Затраты на обмен информацией об отдельных частях проекта быстро начинают превышать выигрыш от увеличения количества разработчиков (Ф. Брукс).

На первых компьютерах для написания программ сначала применялись машинные коды, а затем язык ассемблера. Но этот язык не соответствует сегодняшним

стандартам. По мере роста сложности программ оказалось, что разработчики не в состоянии помнить всю информацию, нужную для отладки и совершенствования их программ. Какие значения хранятся в регистрах? Есть ли уже переменная с этим именем? Какие переменные надо инициализировать, перед тем как передать управление следующему коду?

Частично эти проблемы решили первые языки высокого уровня: Фортран, Кобол, Алгол (например, в них было введено автоматическое управление локальными переменными и неявное присваивание значений). Но рост сложности программ продолжался, и появились проекты, в которых ни один программист не мог удержать в голове все детали. Над проектами стали работать команды программистов.

Значительная взаимозависимость частей ПО мешает создавать ПО по типу конструирования материальных объектов. Например, здание, автомобиль и электроприборы обычно собираются из готовых компонент, которые не надо разрабатывать "с нуля". Многократное использование ПО – цель, к которой постоянно стремятся, но и которой редко достигают. Из программной системы тяжело извлечь независимые фрагменты. ООП облегчает эту задачу.

3. Понятие декомпозиции в задачах программирования

Само по себе применение объектно-ориентированного языка не вынуждает к написанию ОО-программ, хотя и упрощает их разработку. Чтобы эффективно использовать ООП, требуется рассматривать задачи иным способом, нежели это принято в процедурном программировании.

Известно утверждение, применимое к естественным языкам, что язык, на котором высказывается идея, направляет мышление (распространенный пример про снег и какой-нибудь профессиональный лексикон, который формируется "по потребности" для решения задач в конкретный предметных областях). Как для компьютерных, так и для естественных языков справедливо: язык направляет мысли, но не предписывает их.

Аналогично, объектно-ориентированная техника не снабжает программиста новой вычислительной мощностью, которая бы позволила решить проблемы, недоступные для других средств. Но ОО-подход делает задачу проще и приводит ее к более естественной форме. Это позволяет обращаться с проблемой таким образом, который благоприятствует управлению большими программными системами.

ООП часто называется новой парадигмой программирования. Другие парадигмы: процедурное (языки Паскаль, Си), логическое (Пролог), функциональное (Лисп) программирование. Парадигма программирования – способ концептуализации, который определяет, как проводить вычисления и как работа, выполняемая компьютером, должна быть структурирована и организована.

Процесс разбиения задачи на отдельные, структурно связанные, части, называется *декомпозицией*. При процедурной декомпозиции в задаче выделяются алгоритмы и обрабатываемые ими структуры данных, при логической – правила, связывающие отдельные понятия. При ОО-декомпозиции в задаче выделяются классы и способы взаимодействия объектов этих классов друг с другом.

Центральная часть ООП – техника организации вычислений и данных. В программировании она явилась совершенно новой, но она базируется на давно известном подходе к классификации, примененном еще Линнеем (XVIII в., выделение родов и видов для классификации животных и растений).

4. Взаимодействие объектов на бытовом примере

Основные свойства ООП проиллюстрируем на примере обыденной ситуации, а затем рассмотрим, как можно наиболее близко смоделировать найденное решение на компьютере.

Допустим, что требуется приобрести компьютер. Поскольку нужна гарантия и нет времени на поездки за комплектующими, вариант самостоятельной сборки отпадает. Тогда простейшим вариантом будет пойти в ближайший компьютерный магазин, найти продавца, сформировать с ним конфигурацию компьютера, оплатить заказ и прийти через определенное время, чтобы забрать собранный компьютер.

В решении описанной задачи явно заметны два агента: покупатель и продавец. Покупатель выполняет поиск продавца, передает ему запрос на получение компьютера и затем забирает этот компьютер. В данной естественной трактовке заметно основное свойство ООП: агент-источник посылает сообщение агенту-приемнику, чтобы он выполнил некоторое действие. В терминологии ООП агенты, обменивающиеся сообщениями, называются объектами.

Агент-источник для посылки сообщения выполняет два необходимых действия:

1) поиск подходящего агента;
2) передача агенту сообщения, содержащего запрос.

Удовлетворение запросов заданного типа является обязанностью выбранного агента-приемника. Например, попытка заказать компьютер у продавца в продуктовом магазине закончится неудачей – соответствующий агент выдаст диагностическое сообщение об ошибке.

Для удовлетворения запроса у агента есть некоторый *метод* – алгоритм, или последовательность операций, которая используется агентом для выполнения запроса. Агента, пославшего запрос, детали метода принципиально не интересуют. Например, продавец может взять готовый компьютер со склада, может передать заказ технику на сборку из имеющихся частей, может заказать комплектующие в другой фирме и собрать компьютер позже, или может купить компьютер в другой фирме и затем перепродать его.

Т.е. в реализации метода может быть предусмотрена переадресация сообщения другому агенту. Запрос может быть удовлетворен в результате выполнения последовательности запросов, посылаемых различными агентами друг другу. Но агенты не могут во всех случаях реагировать на сообщения только переадресацией этих сообщений. На некоторой стадии по крайней мере некоторые агенты должны выполнять какую-то работу перед пересылкой запроса другим агентам.

5. Основные принципы ООП

5.1 Сообщения

Первый принцип ООП: *действия задаются в форме сообщений, посылаемых одними объектами другим*.

Действие в ООП инициируется посредством передачи сообщения объекту, ответственному за выполнение действия. Сообщение содержит запрос на осуществление действия и сопровождается дополнительной информацией (параметрами), необходимой для выполнения действия (например, при покупке компьютера параметрами является описание конфигурации и деньги).

Получатель (receiver) – это объект, которому посылается сообщение от объекта-клиента (client или sender). Если получатель принимает сообщение, то на него автоматически возлагается ответственность за выполнение указанного действия. В качестве реакции на сообщение получатель запустит некоторый метод, чтобы удовлетворить принятый запрос.

Понятие обязанности или ответственности за выполнение действия является фундаментальной концепцией ООП. Запрос выражает только стремление получить желаемый результат, а не способ его достижения. Полный набор обязанностей, связанных с определенным объектом, часто определяется с помощью термина "протокол".

5.2 Скрытие информации

При пересылке сообщений действует важный *принцип скрытия информации*: клиенту, посылающему запрос, ничего не требуется знать о способе его выполнения. Если уже существует объект, который может выполнить запрос, то получатель может переадресовать запрос ему. Т.о., ООП поощряет разработку повторно используемых компонент программного обеспечения.

Скрытие информации является важным принципом и в традиционных языках программирования. Пересылка сообщений отличается от вызова процедуры, хотя в обоих случаях имеется последовательность точно определенных действий, выполняемых в ответ на запрос. Выделяются два отличия. Во-первых, у сообщения имеется вполне конкретный получатель – объект, которому послано сообщение (хотя на уровне реализации вполне может быть так, что получатель передается в качестве первого параметра процедуры). Во-вторых, интерпретация сообщения (вызываемый метод) зависит от получателя и является различной для различных получателей (например, в разных компьютерных магазинах сборка компьютера может выполняться по-разному).

Часто конкретный получатель неизвестен вплоть до выполнения программы. В таком случае говорят, что имеет место позднее связывание между сообщением (именем процедуры или функции) и фрагментом кода (методом), используемым в ответ на сообщение. Эта ситуация противопоставляется раннему связыванию (на этапе компиляции и компоновки программы) имени с фрагментом кода, что происходит при традиционных вызовах процедур.

5.3 Наследование

Все объекты являются представителями, или экземплярами, классов. Понятие "класс" обозначает категорию объектов, имеющих общие черты. Классы будем обозначать именами, начинающимися с заглавной буквы 'C' (от слова class).

Класс включает в себя (инкапсулирует) набор свойств (переменных), определяющих состояние объекта данного класса, и набор действий (методов), определяющих поведение объектов данного класса.

Метод, выполняемый объектом-получателем в ответ на сообщение, определяется классом, к которому принадлежит получатель сообщения. Все объекты одного класса используют одни и те же методы в ответ на одинаковые сообщения.

Организация знаний о классах представляется в виде иерархии. Классы представляются в виде иерархической древовидной структуры, в которой более абстракт-

ные (т.е. более общие) классы располагаются в корне дерева, а более специализированные классы располагаются на его концах, в ветвях. Древовидные структуры в программировании принято изображать в направлении "от корня вниз" (рис. 1.1).

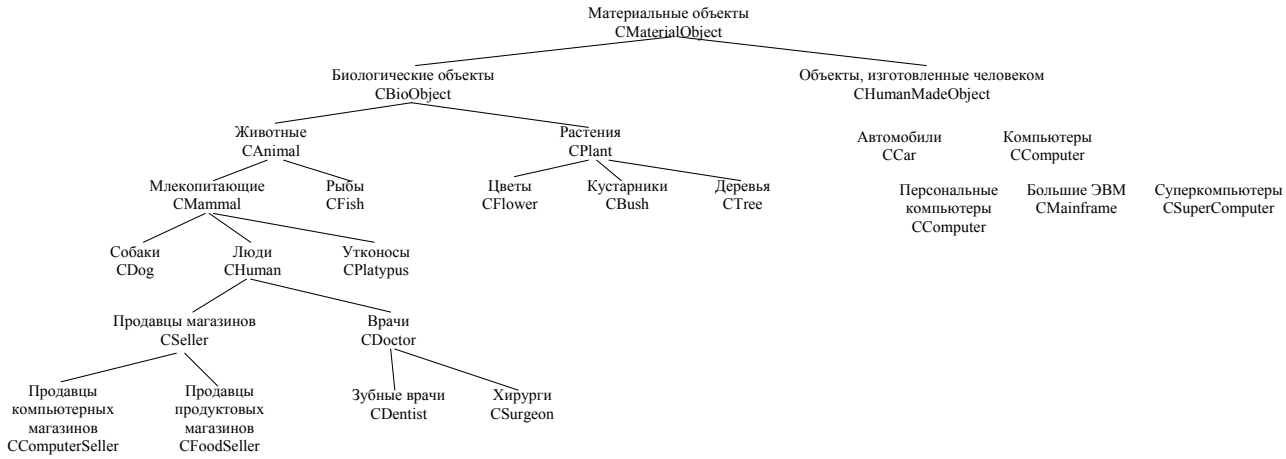

Рис. 1.1. Иерархическое дерево классов для некоторых объектов материального мира

Идея наследования состоит в том, что классы могут быть организованы в иерархическую структуру с наследованием свойств. Дочерний класс (или подкласс) наследует свойства родительского класса (или надкласса), расположенного выше в иерархическом дереве. Абстрактный родительский класс – это класс, не имеющий экземпляров. Он используется только для порождения подклассов.

5.4 Полиморфизм

Информация, содержащаяся в подклассе, может переопределять информацию, наследуемую из родительского класса (например, на рис. 1.1 утконосы отнесены к млекопитающим, но они являются яйцекладущими). Очень часто при реализации такого подхода метод, соответствующий подклассу, имеет то же имя, что и соответствующий метод в родительском классе. При поиске метода, подходящего для обработки сообщения, используется следующее правило: поиск метода начинается с методов, принадлежащих классу получателя. Если подходящий метод не найден, то поиск продолжается для родительского класса.

Применение различных методов разными объектами для обработки одного сообщения является примером полиморфизма.

6. Формулировка характеристик ООП

Фундаментальные характеристики ООП (в формулировке Алана Кея, одного из основоположников ООП и разработчика языка Смолльток):

1) Все является объектом.
2) Вычисления осуществляются путем взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некое действие. Объекты взаимодействуют, посылая и получая сообщения. Сообщение – это запрос на выполнение действия, дополненный набором параметров, которые могут понадобиться для выполнения действия.

3) Каждый объект имеет независимую память, которая состоит из других объектов.
4) Каждый объект является представителем класса, который выражает общие свойства объектов (например, таких, как целые числа или списки).
5) В классе задается поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.
6) Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение, связанное с экземплярами определенного класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

Традиционная модель, описывающая выполнение программы на компьютере, базируется на дуализме процесс–состояние. С этой точки зрения компьютер является администратором данных, следующим некоторому набору инструкций. Он перемещается по пространству памяти, изымает значения из ее ячеек (адресов памяти), некоторым образом преобразует полученные величины, а затем помещает их в другие ячейки. Проверяя значения, находящиеся в различных ячейках, мы определяем состояние машины или же результат вычислений. Хотя эта модель может рассматриваться как более или менее точный образ хранения предметов в почтовых ящиках или значений в ячейках памяти, но мало что из житейского опыта может подсказать, как структурировать задачу.

Антропоморфные описания являются отражением огромной выразительной силы метафор. Декомпозиция задачи на набор взаимодействующих объектов позволяет в ряде случаев применять естественные аналогии с повседневным опытом и способами моделирования, принятыми в конкретных предметных областях, для которых разрабатывается программное обеспечение

7. Развитие средств абстрагирования в программировании

Важность ООП-подхода можно понять, рассмотрев разнообразные механизмы, которые использовались программистами для контроля над сложностью. Сложность программного обеспечения в крупных проектах проявляется в том, что начиная с некоторого момента добавление новых разработчиков удлиняет, а не сокращает расписание работ над проектом.

Сложность порождается не только большим объемом задач, а вследствие уникального свойства программных систем, разработанных с использованием традиционных подходов – в этих системах существует большое количество перекрестных ссылок между компонентами. Перекрестные ссылки в данном случае обозначают зависимость одного фрагмента кода от другого. Действительно, каждый фрагмент программной системы должен выполнять некоторую работу, иначе он оказывается не нужен. Если эта деятельность нужна другим частям программы, то этот фрагмент должен обмениваться с ними данными. По этой причине, чтобы понять код фрагмента, надо знать и тот код, который им пользуется. Получается, что даже относительно независимый фрагмент программы нельзя полностью понять в изоляции от других фрагментов.

Главный способ борьбы со сложностью ПО – абстрагирование, те. способность отделить логический смысл фрагмента программы от проблемы его реализации. В не-

котором смысле ОО–подход не является революционным и может рассматриваться как естественный результат эволюции от процедур к модулям, далее к абстрактным типам данных и, наконец, к объектам.

7.1 Подпрограммы

Подпрограммы (процедуры и функции являются разновидностями подпрограмм) – это первые механизмы абстрагирования в языках программирования. Они позволяют сконцентрировать в одном месте работу, выполняемую многократно и затем многократно использовать этот код, вместо того чтобы писать его снова и снова. Они впервые обеспечили возможность скрытия информации – пользователи процедур могли не знать деталей реализованного алгоритма, а только интерфейс программы.

Недостатки подпрограмм: нет эффективного механизма скрытия данных, проблема использования одинаковых имен полностью не снимается.

В качестве примера применения подпрограмм можно привести стек, реализованный с помощью глобальных переменных.

```
int datastack[100];
int datatop = 0;

void init()
{
  datatop = 0;
}

void push( int val )
{
  if ( datatop < 100 )
    datastack[datatop++] = val;
}

int top()
{
  if ( datatop > 0 )
    return datastack[datatop – 1];
  return 0;
}

int pop()
{
  if ( datatop > 0 )
    return datastack[--datatop];
  return 0;
}
```

7.2 Модули

Модули – улучшенный метод создания и управления совокупностями имен и связанными с ними значениями. Пример со стеком: есть информация (интерфейсные процедуры), которая должна быть широко доступной, и есть некоторые данные (собственно данные стека), доступ к которым должен быть ограничен. Суть модуля состоит в разбиении пространства имен на две части: открытая (public) часть является доступной извне модуля, закрытая (private) часть доступна только внутри модуля.

Типы, данные (переменные) и процедуры могут быть отнесены к любой из двух частей.

Были сформулированы (Д. Парнас, 1972) два принципа использования модулей:

1) Пользователя модуля надо снабдить *минимально* необходимой для его использования информацией.
2) Разработчика надо снабдить *минимально* необходимой информацией для создания модуля.

Достоинства модулей: эффективный механизм скрытия данных.

Недостатки: нет способа размножения экземпляров областей данных. Для преодоления этого недостатка была разработана следующая концепция.

7.3 Абстрактные типы данных

АТД создаются программистом, но с этими типами можно работать так же, как и со встроенными типами данных. Каждому АТД соответствует набор допустимых значений (м.б. бесконечный) и ряд элементарных операций, которые могут быть выполнены над данными. Например, стек можно определить как АТД и служебные функции – как единственные действия, которые можно производить над отдельными экземплярами стеков.

Модули часто используются для реализации АТД, но непосредственной логической связи между этими идеями нет, хотя они и близки. Чтобы построить АТД, надо уметь:

1) Экспортировать определения типа данных.
2) Делать доступным набор операций, использующихся для манипулирования экземплярами типа данных.
3) Защищать данные, связанные с типом данных, чтобы с ними можно было работать только через указанные подпрограммы.
4) Создавать несколько экземпляров АТД.

В этом определении модули служат только как механизм скрытия информации для выполнения шагов 2) и 3). Остальные шаги могут быть выполнены с помощью соответствующей техники программирования.

7.4 Объекты. Сообщения, наследование и полиморфизм.

Правда, что объекты являются АТД, но понятия ООП, хотя и строятся на идеях АТД, добавляют к ним важные новшества по части разделения и совместного использования программного кода.

Главная идея: пересылка сообщений. Действие инициируется по запросу, обращенному к конкретному объекту, а не через вызов функции (хотя это м.б. только способ интерпретации, а не реализации). Неявная идея в пересылке сообщений – то, что интерпретация сообщения может меняться для различных объектов. Например, `push` для стека и робота-манипулятора означают разные действия. Имена операций не обязаны быть уникальными, что приводит к более читаемому и понятному исходному тексту.

Механизм наследования: позволяет различным типам данных совместно использовать один и тот же код, приводя к уменьшению его размера и повышению функциональности.

Полиморфизм: перекраивает этот общий код так, чтобы удовлетворить конкретным особенностям отдельных типов данных.

Упор на независимость индивидуальных компонент позволяет использовать процесс пошаговой сборки, когда отдельные блоки ПО разрабатываются, программируются и отлаживаются до того, как они объединяются в большую систему.

Структурный и ОО-подходы различаются тем, что в первом случае программист думает, как обрабатывать структуры данных, а во втором, что именно эти структуры данных "могут сделать".

8. Резюме

ООП – не просто набор новых свойств, добавленных в существующие языки. Это новый шаг в осмыслении процессов декомпозиции задач и разработки ПО.

ООП рассматривает программы как совокупность гибко связанных между собой агентов, называемых объектами. Каждый из них отвечает за конкретные задачи. Вычисление осуществляется посредством взаимодействия объектов. Программирование превращается в процесс моделирования в какой-либо предметной области.

Объект получается в результате инкапсуляции состояния (данных) и поведения (операций). Тем самым объект во многих отношениях аналогичен модулю или абстрактному типу данных.

Поведение объекта диктуется его классом. Каждый объект является экземпляром некоторого класса. Все экземпляры одного класса будут вести себя одинаковым образом (то есть вызывать те же методы) в ответ на одинаковые запросы.

Объект проявляет свое поведение путем вызова метода в ответ на сообщение. Интерпретация сообщения (то есть конкретный используемый метод) зависит от объекта и может быть различной для различных классов объектов.

Объекты и классы расширяют понятие АТД путем введения наследования. Классы могут быть организованы в виде иерархического дерева наследования. Данные и поведение, связанные с классами, которые расположены выше в иерархическом дереве, доступны для нижележащих классов. Происходит наследование поведения от родительских классов.

С помощью уменьшения взаимозависимости между компонентами программного обеспечения ООП позволяет разрабатывать системы, пригодные для многократного использования. Такие компоненты могут быть созданы и отлажены как независимые программные единицы, в изоляции от других частей прикладной программы.

Многократно используемые программные компоненты позволяют разработчику иметь дело с проблемами на более высокой ступени абстрагирования. Мы можем определять и манипулировать объектами просто в терминах сообщений, которые они распознают, и работы, которую они выполняют, игнорируя детали реализации.

Лекция 2. Объектно-ориентированное проектирование

Работа на ОО-языке (т.е. на языке с поддержкой наследования, пересылки сообщений и классов) не является ни необходимым, ни достаточным условием для того, чтобы заниматься ООП. Наиболее важный аспект в ООП – техника проектирования, основанная на выделении и распределении обязанностей (responsibility-driven design).

В примере из предыдущей лекции о заказе компьютера в магазине можно отметить, что после того, как заказ сделан, покупатель не задумывается о том, как именно заказ будет выполнен. Для покупателя важно только то, что после выдачи запроса будет получен желаемый результат.

Традиционное процедурное программирование основывается в основном на приказаниях чему-либо сделать что-то – к примеру, модифицировать запись или обновить массив данных. Каждый фрагмент программы оказывается сильно связанным со многими другими фрагментами. Проектирование, основанное на распределении обязанностей, старается отсекать эти связи или по крайней мере сделать их максимально слабыми. Поэтому в ООП жизненно важным оказывается принцип скрытия информации.

1. Масштаб проектов разработки программного обеспечения

Одно из основных преимуществ ООП наблюдается, когда программные компоненты многократно используются в разных проектах. Это особенно важно при разработке "больших проектов". Программные проекты можно условно разделить на "малые" и "большие".

Для "малых" проектов характерно:

- Код разрабатывается единственным программистом, или, возможно, небольшой группой программистов. Отдельно взятый индивидуум может понять все аспекты проекта.
- Основная проблема при разработке состоит в проектировании программы и написании алгоритмов для решения поставленной задачи.

С другой стороны, у "больших" проектов можно отметить следующее:

- Программная система разрабатывается большой командой программистов. При этом одна группа может заниматься проектированием (или спецификацией) системы, другая – осуществлять написание кода отдельных компонент, а третья – объединять компоненты в конечный продукт. Нет единственного человека, который бы знал о проекте все.
- Основная проблема в процессе разработки – управление проектом и обмен информацией между группами и внутри групп.

Особенности многих ОО-языков наилучшим образом проявляются при программировании "больших" проектов. ООП получило широкое развитие как раз благодаря упрощению разработки подобных проектов.

2. Учебный пример: электронный ежедневник. Общее описание программы

Программа "Электронный ежедневник" предназначена для персональных компьютеров. Она должна заменить бумажную записную книжку. Ежедневник предназначен для ведения базы данных дел, запланированных на разные даты и время. Предполагается, что значительная часть дел является встречами и телефонными пере-

говорами с другими людьми, поэтому ежедневник должен также позволять работать с базой данных людей.

Ежедневник должен позволять планировать расписание встреч и других дел на длительный период. Пользователь программы может просматривать встречи по дням, искать дела по теме и по людям, заполнять записи о новых встречах на заданные дни или подбирать свободное время для новых дел, а также просматривать базу данных людей и пополнять ее в диалоговом режиме.

Как обычно и бывает, первоначальное описание системы двусмысленно и не слишком полно. На данном примере рассмотрим, как будет выполняться уточнение проекта и разбиение его на компоненты, которые можно поручить различным разработчикам. Основой объектно-ориентированного проектирования является характеристика программного обеспечения в терминах *поведения*, т.е. в терминах действий, которые должны быть выполнены.

Сначала поведение характеризуется на очень абстрактном уровне, т.е. поведение программы в целом. Затем описывается поведение различных компонент. Затем, только тогда, когда все аспекты поведения будут выделены и описаны, программисты-разработчики приступят к написанию исходного текста.

3. Основные этапы проектирования программной системы

Сначала надо выполнить анализ функционирования (поведения) системы. Это объясняется тем, что поведение системы обычно известно задолго до остальных ее свойств.

Предшествовавшие методы разработки ПО концентрировались на таких идеях, как характеристики основных данных или же общая структура вызова функций. Но структурные элементы программы могут быть определены только после интенсивного анализа задачи. Поэтому формирование формальной спецификации на первом этапе часто заканчивается созданием документа, который не понимают ни программисты, ни клиенты.

Но поведение – это нечто, что может быть описано в момент возникновения идеи программы и выражено в терминах, имеющих значение как для программиста, так и для клиента.

В разработке системы с использованием объектно-ориентированного проектирования можно выделить несколько основных этапов, на каждом из которых главную роль играет все более детальный анализ поведения системы.

1) Уточнение спецификации (постановки задачи)

Исходные спецификации обычно двусмысленны и непонятны во всем, кроме наиболее общих положений. На этом этапе надо уточнить, чем будет конечный продукт и обсудить структуру будущей программной системы. Уточненная спецификация передается для дальнейшего обсуждения клиенту.

2) Идентификация компонент

Создание сложной системы, вроде здания или автомобиля, упрощается с помощью разбиения проекта на структурные единицы. Аналогично, разработка программ облегчается после выделения в них отдельных компонент. *Компонента – это просто абстрактная единица, которая может выполнять определенную работу (т.е. иметь определенные обязанности).* На этом этапе нет необходимости знать в точности то, как задается компонента и как именно она будет выполнять свою работу.

В конечном счете компонента может быть преобразована в отдельную функцию, структуру или класс, или же в совокупность других компонент (шаблон).

3) Разработка документации

Разработку документации следует начинать уже на первых этапах разработки проекта. Документация включает в себя два основных документа: руководство пользователя и проектную документацию. Эти документы начинают разрабатываться задолго до написания исходного текста. В руководстве пользователя описывается взаимодействие с системой с точки зрения пользователя. Это руководство может служить для проверки того, как концепция разработчиков соответствует мнению клиента.

В проектной документации протоколируются основные решения, принятые при планировании программы. Сначала в ней приводится глобальное описание системы, а затем совершается переход к уровню отдельных компонент. Чересчур детальное описание внутреннего устройства компонент может затруднить понимание системы в целом.

4) Выбор представления данных

На данном этапе команда разработчиков разделяется на группы, отвечающие за конкретные компоненты программы. Теперь надо решить, как перейти от описания компоненты к конкретному коду. Главное здесь – проектирование структур данных, которые будут использоваться каждой компонентой для хранения внутренней информации, а также преобразование описания поведения компонент в алгоритмы.

5) Реализация компонент

Если предыдущие этапы выполнены корректно, то каждая обязанность или поведение будут кратко охарактеризованы, выбраны структуры данных и сформированы алгоритмы. Теперь надо записать их на языке программирования. На этом этапе детально разрабатываются конкретные компоненты. Для программистов, работающих над проектом, крайне важно понимать, как отдельный фрагмент кода подключается к более высокому уровню, и уметь работать в составе группы. При реализации каждой компоненты надо проверить, правильно ли она работает, если вызвать ее с корректными входными значениями.

6) Интеграция компонент

Когда индивидуальные компоненты разработаны и протестированы, они должны быть интегрированы в конечный продукт. Это делается поэтапно, начиная с элементарной основы (макета системы), к которой постепенно добавляются новые элементы. Для еще не реализованных частей применяются заглушки. Постепенно заглушки заменяются настоящим кодом и проводится тестирование. Этот процесс называется *тестированием системы в целом*.

Если ошибка, проявляющаяся в одной из компонент, оказывается вызвана некорректным кодом в другой, то эта ошибка исправляется и тестирование повторяется. Этот процесс называется *регрессионным тестированием*.

7) Сопровождение и развитие

С передачей продукта пользователю работа разработчиков не завершается. Практически всегда требуется дополнительное сопровождение программного обеспечение. Это вызвано необходимостью исправлять ошибки, изменением требований к системе в связи с появлением новых технических и государственных стандартов, переходом на новую аппаратную платформу, изменением запросов пользователей (м.б., в связи с появлением конкурирующих продуктов).

4. Методика выделения компонент на основе CRC-карточек

После уточнения спецификации системы команда программистов прорабатывает сценарий системы, чтобы выявить отдельные компоненты и определить их обязанности. Т.е. воспроизводится запуск программы, как если бы она была уже готова. Любое действие, которое может произойти, приписывается некоторой компоненте в качестве ее обязанности. В целом, компонента должна удовлетворять двум важным требованиям:

- компонента должна иметь небольшой набор четко определенных обязанностей;
- компонента должна взаимодействовать с другими компонентами настолько слабо, насколько это возможно.

Выделение компонент производится во время мысленного представления работы системы. Часто это происходит как цикл вопросов "что/кто". Разработчики определяют: что требуется делать? Это немедленно приводит к вопросу: кто будет выполнять действие? Действия, которые должны быть выполнены, приписываются некоторой компоненте в качестве ее обязанностей. В ОО-проекте для каждого действия обязательно должна быть установлена выполняющая это действие компонента.

Для выделения обязанностей компонент удобно изображать компоненты с помощью небольших карточек. На лицевой стороне карточки написаны имя компоненты, ее обязанности и имена других компонент, с которыми она должна взаимодействовать (рис. 2.1). Такие карточки обычно называются *CRC-карточками* от слов Component, Responsibility, Collaborator (компонента, обязанность, сотрудники).

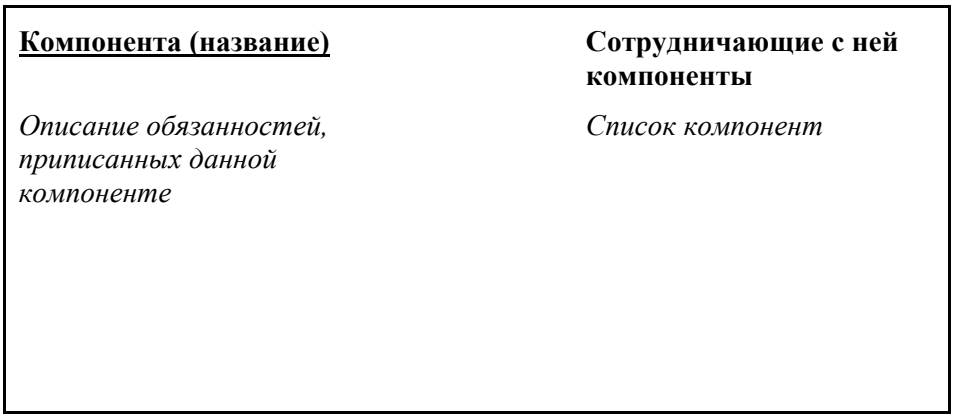

Рис. 2.1. Структура CRC-карточки

CRC-карточки недороги и доступны, поэтому с минимальными затратами можно изучить несколько альтернативных проектов. Физическое разделение карточек стимулирует понимание важности логического разделения компонент. Небольшой размер карточки служит оценкой примерной сложности отдельного фрагмента – компонента, которой приписывается больше задач, чем может поместиться на карточке, вероятно, является излишне сложной, и должно быть найдено более простое решение (например, можно разбить компоненту на две или пересмотреть разделение обязанностей между различными компонентами).

5. Учебный пример: проектирование электронного ежедневника

5.1 Идентификация компонент

Рассмотрим начало проектирования ежедневника. Будем считать, что после запуска система показывает привлекательное информационное окно. Ответственность за его отображение приписана компоненте, названной `MainWin`. Некоторым образом (с помощью меню, клавиатуры, или мыши) пользователь выбирает одно из нескольких действий:

1) Просмотр дел на заданную дату.
2) Создание нового дела.
3) Редактирование существующего дела.
4) Поиск времени, на которое назначена встреча с заданным человеком.
5) Просмотр базы данных о людях.
6) Добавление сведений о новом человеке.
7) Редактирование данных о человеке.

Эти действия разбиваются на три группы. Первые три связаны с базой данных дел, одно – с базой данных дел и с базой данных людей, последние три – с базой данных людей. В результате принимается решение: нужны по крайней мере две компоненты – база данных дел и база данных людей.

Компоненты, работающие с базами данных, должны обеспечивать просмотр существующих данных, редактирование и добавление новых записей. Т.е. их назначение – поддержка однотипных записей. Но создание нового дела осложняется тем, что надо следить, чтобы не было пересекающихся дел и при необходимости выполнять поиск свободных временных промежутков. Поэтому для создания нового дела требуется отдельная компонента – менеджер планирования. Таким образом, учитывая наличие трех компонент, можно сформировать первый вариант CRC-карточки для компоненты `MainWin` (рис. 2.2).

MainWin (главное окно программы)	**Сотрудничающие компоненты**
Вывести на экран заставку	База данных дел
Предложить пользователю выбрать параметры	База данных людей
Передать управление другой компоненте:	Менеджер планирования
базе данных дел	
базе данных людей	
менеджеру планирования	

Рис. 2.2. CRC-карточка компоненты MainWin.

На данном этапе часть решений, касающихся отдельных компонент, можно отложить. Например, как пользователь станет просматривать базу данных дел? Возможны несколько вариантов: по дням, или задавая имя человека, или по теме встречи. В каком виде отображать перечень дел на один день? Стоит ли имитировать страницы ежедневника? Все эти решения влияют только на отдельные компоненты и не затрагивают функционирование остальных частей системы.

Однако при разработке компонент следует учитывать, что в будущем неизбежны изменения. Они связаны как с неточностью исходной спецификации, так и с изменением желаний и потребностей пользователей. Потому разработчики должны по возможности планировать свои действия с учетом нескольких соображений:

- Главная цель состоит в том, что изменения должны затрагивать как можно меньше компонент.
- Старайтесь предсказывать наиболее вероятные источники изменений и по возможности изолируйте их от других компонент (обычно это пользовательский интерфейс, форматы обмена информацией, вид выходных данных).
- Уменьшайте зависимость программы от аппаратуры. Например, ежедневник впоследствии может потребоваться переработать для карманного компьютера.

5.2 Взаимосвязь компонент

Каждое назначенное дело будет связано с конкретной программной компонентой `Deal`. При просмотре дел удобен режим, когда они показываются по дням. Поэтому дела, относящиеся к одному дню, будут связаны с отдельной компонентой (`Date`). При выборе конкретного дела управление передается объекту, связанному с этим делом.

Дело содержит некоторую информацию: это время, на которое назначено дело, примерная продолжительность, с кем назначена встреча и краткое содержание дела. Компонента `Deal` умеет отображать себя на экране.

При выборе дела для редактирования управление передается этой компоненте: возможно изменение времени, содержания дела и других данных. С другой стороны, пользователь может попросить распечатать дела, относящиеся к определенному дню. Это является обязанностью компоненты `Date`. Пока мы рассматриваем компоненты `Date` и `Deal` как отдельно взятые объекты, хотя на самом деле это прототипы многочисленных объектов.

Предположим, пользователь хочет добавить в базу данных людей нового человека. В блоке управления БД некоторым образом (пока не конкретизируется, как именно) определяется, в какой раздел поместить человека (деловые контакты, личные и т.п.), запрашивается его имя, характеристики (например, день рождения и место работы), телефон и выводится окно для набора текста комментария. Эту задачу естественно отнести к компоненте `Person`, которая отвечает за хранение и редактирование данных о людях.

При просмотре дел должна быть возможность просмотра информации о человеке, с которым назначена встреча.

Теперь рассмотрим, как выполняется планирование дел. Редактирование сведения о деле выполняется компонентой `Deal`. Но при создании нового дела или изменении времени существующего дела надо проверить, свободно ли первоначально предполагаемое время или выбрать подходящее время. Это выполняет менеджер планирования – `Plan Manager`. Он подбирает корректную дату для дела и передает ему управление для редактирования.

Вообще, компонента `Plan Manager` может запускаться или для просмотра существующих дел по дням, или при создании нового дела.

Каждая дата ассоциируется с компонентой типа `Date`. Она может показывать и распечатывать список дел, относящихся к этому дню. В компоненте `Date` хранится

значение даты, некоторый комментарий к этому дню (какие-либо напоминания, информация о днях рождения и др.).

В конечном счете можно сказать, что все действия можно надлежащим образом распределить между семью компонентами (рис. 2.3). На рис. 2.3 отрезками показано, какие компоненты с какими взаимодействуют. Например, компонента `Date` обращается к компонентам `Deal` только через посредство базы данных `Deal Database`.

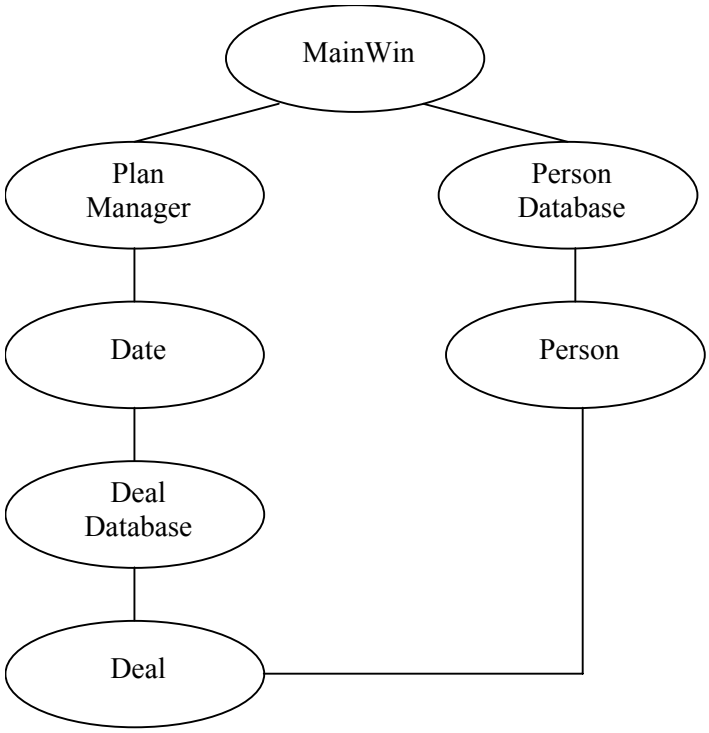

Рис. 2.3. Взаимосвязь между компонентами электронного ежедневника по отношению "доступ".

5.3 Диаграммы взаимодействия

Схема, показанная на рис. 2.3, хорошо подходит для отображения статических связей между компонентами. Она не годится для описания динамического взаимодействия во время выполнения программы. Для этого применяются диаграммы взаимодействия (рис. 2.4). На рис. 2.4 по вертикальной оси в направлении сверху-вниз откладывается время. Каждая компонента представлена вертикальной линией. Сообщение от одной компоненты другой изображается горизонтальной стрелкой между вертикальными линиями. Возврат управления (и, возможно, результата) в компоненту представлен пунктирной стрелкой.

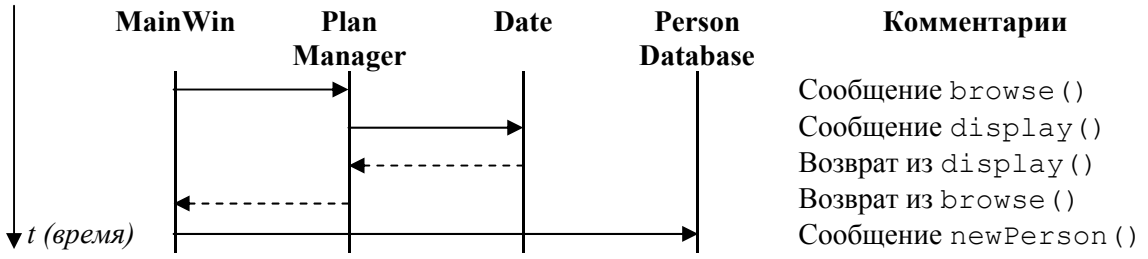

Рис. 2.4. Фрагмент диаграммы взаимодействия для программы электронного ежедневника.

6. Некоторые характеристики компонент программ

6.1 Поведение и состояние

Компоненты характеризуются поведением и состоянием. *Поведение* – это то, что должна делать компонента (обязанности компоненты), а *состояние* – это внутренняя информация компоненты, необходимая ей для выполнения предписанных обязанностей. Полное описание поведения компоненты иногда называется *протоколом*.

Например, в протоколе компоненты `Deal` значится, что она осуществляет редактирование параметров назначенных дел, отображает их на экране, печатает на принтере. Состояние компоненты `Deal` включает в себя значение времени, на которое назначена встреча, примерная длительность встречи, ее краткое содержание. Состояние не является статическим и может изменяться с течением времени. Например, пользователь может перенести встречу или изменить ее содержание.

Не все компоненты обязаны иметь состояние. Например, у компоненты `MainWin` скорее всего не будет никаких внутренних данных. Но большинство компонент характеризуется и поведением, и состоянием.

6.2 Экземпляры и классы

Выше были разделены понятия поведения и состояния. В электронном ежедневнике будет храниться много встреч и людей. Все компоненты `Deal` будут вести себя одинаково. Отличается только их состояние: время встречи, фамилия человека, с кем назначена встреча, ее содержание. На ранних стадиях разработки нас интересует поведение, общее для всех компонент `Deal`. Детали, специфические для отдельной встречи, не важны.

Термин "*класс*" используется для описания множества сущностей с похожим поведением. Конкретные представители класса называются *объектами*. Очень важно заметить, что поведение ассоциировано с классом, а не с индивидуальными объектами. Т.е. все объекты данного класса воспринимают одни и те же сообщения и выполняют их обработку одинаково. С другой стороны, состояние является уникальной характеристикой объекта. Это видно на примере различных объектов `Deal`. Все они могут выполнять одинаковые действия (редактирование, вывод на экран, печать), но используют различные данные.

6.3 Скрытие информации

Идея характеризации компонент через их поведение имеет одно чрезвычайно важное следствие. Программист может знать, как использовать компоненту, разработанную другим программистом, не вникая в детали ее реализации.

Допустим, семь компонент ежедневника разрабатываются разными программистами. Разработчик компоненты `Date` должен обеспечить просмотр дел, назначенных на заданную дату и выбор отдельного дела. Для этого компонента `Date` просто вызывает функцию `browse()`, привязанную к компоненте `Deal Database`. Эта функция возвращает из базы данных набор дел на заданную дату. Это справедливо независимо от того, как внутри `Deal Database` реализована работа с базой данных.

Разделение интерфейса и реализации является, возможно, наиболее важной идеей в программировании. Ее непросто понять, т.к. скрытие информации имеет зна-

чение в основном только при разработке больших проектов, над которыми работают много программистов.

Напомним принципы, сформулированные Дэвидом Парнасом:

- разработчик компоненты должен предоставить пользователю компоненты минимум информации, позволяющий ее использовать.
- разработчик компоненты должен знать только требуемое поведение компоненты и ничего кроме этого.

Принцип отделения интерфейса от реализации облегчает программисту экспериментирование с различными алгоритмами, не затрагивая при этом остальных компонент программы.

6.4 Типы компонент

После того, как компоненты выделены и определено их поведение, можно решить, как они будут реализованы. Компонента, характеризуемая только поведением (не имеющая внутреннего состояния), может быть оформлена в виде функции. Например, это компонента, заменяющая все заглавные буквы в символьной строке на строчные. Компоненты с многими функциями лучше реализовать в виде классов.

Каждой обязанности, записанной на CRC-карточке компоненты, присваивается имя. Эти имена станут затем названиями функций или методов. Вместе с именами определяются типы параметров, передаваемых функциям. Затем описывается вся информация, содержащаяся внутри компоненты. Если компоненте требуются некие данные для выполнения конкретного задания, их источник (параметр функции, глобальная или внутренняя переменная) должен быть явно описан.

6.5 Имена компонент

Имена, связанные с различными действиями, должны тщательно выбираться. Они должны быть внутренне совместимы, значимы, коротки и содержательны.

Можно привести несколько положений общего характера, регулирующих выбор имен:

- Используйте имена, которые легко произнести вслух;
- Чтобы отметить начало слова в составном имени, надо применять заглавные буквы или символы подчеркивания (напр., `CardReader` или `Card_Reader` вместо `cardreader`).
- Тщательно проверяйте сокращения (напр., непонятно, что значит `TermProcess` – процесс, связанный с терминалом компьютера (terminal process) или нечто, что прекращает выполнение процесса (terminate process)).
- Избегайте многозначности имен (напр., функция `empty()` – выполняет проверку того, что некоторый объект пуст, или же удаляет все содержимое объекта и делает его пустым?).
- Не используйте цифры в именах (легко перепутать `0` и `O`, `1` и `l`, `2` и `Z`, `5` и `S`).
- Логическим функциям присваивайте такие имена, чтобы было ясно, как интерпретировать `true` и `false` (напр., `PrinterIsReady` ясно показывает, что значение `true` соответствует принтеру в рабочем состоянии, а `PrinterStatus` является гораздо менее точным).

После того, как для всех действий выбраны имена, CRC-карточка каждой компоненты переписывается заново с указанием имен функций и списка параметров

(рис. 2.5). Но пока остается не установленным, как именно каждая компонента будет выполнять указанные действия.

После переписывания CRC-карточек необходимо еще раз детально проанализировать сценарий работы программы, чтобы гарантировать, что все действия учтены и вся необходимая информация имеется и доступна для соответствующих компонент.

Компонента Date (один день ежедневника)	**Сотрудничающие компоненты**
Содержит информацию о делах, запланированных на конкретный день Date(year, month, day) Создает новый экземпляр типа Date Display() Выводит информацию обо всех встречах на данный день Edit() Редактирует информацию общего характера о заданном дне ViewPersons() Показывает более подробную ин формацию о людях, с которыми назначены встречи на этот день	Менеджер планирования База данных дел

Рис. 2.5. CRC-карточка для компоненты Date

7. Упражнения.

1) Выполните идентификацию компонент и описание их обязанностей для одного из перечисленных технических устройств: видеомагнитофон, телевизор, автомат по продаже газированных напитков, лифт.
2) Выполните идентификацию компонент применительно к игре в морской бой (примеры компонент – игровое поле и корабли) или для игры "объемный тетрис".
3) С помощью CRC-карточек опишите компоненты программы "музыкальный проигрыватель", в которой предусмотрено ведение базы данных музыкальных файлов и их воспроизведение.

Лекция 3. Объявление классов в Си++

В данной лекции рассматривается объявление классов в языке Си++, которое включает в себя описание переменных и методов, содержащихся в классах. Необходимо понимать, что объявление класса и создание объекта класса – это различные действия. В первом случае просто указывается новый тип данных. Объявление характеризует общие свойства и общее поведение множества объектов, но само по себе объявление класса не создает новых данных. Это происходит при создании объекта класса. Данную операцию можно рассматривать как разновидность объявления переменной. Создание объектов описывается в следующей лекции.

1. Инкапсуляция поведения и состояния

Инкапсуляция (encapsulation) – это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования.

Классы можно рассматривать с нескольких точек зрения. Применительно к операции объявления удобно рассматривать классы как абстрактные типы данных. При программировании, основанном на абстракции данных, программный код, реализующий типы данных, разбивается на две части. Интерфейсная часть, доступная пользователю типа, представляет собой совокупность операций, которые определяют поведение абстракций. Вторая часть, часть реализации, видна программисту, выполняющему реализацию этого типа данных. В части реализации видны значения переменных, которые нужны для поддержания внутреннего состояния объекта.

В качестве примера рассмотрим АТД "стек". Пользователь видит описание допустимых операций: `push`, `pop`, `top` и т.д. С другой стороны, программисту, реализующему стек, необходимо работать с конкретными структурами данных, где хранятся значения стека, например, для целочисленного стека емкостью до 100 чисел это может быть массив `int n[100]`. Т.о. оказывается. что конкретные детали скрыты (инкапсулированы) внутри более абстрактного объекта.

Для обозначения представителя класса используется термин "*объект*". Для обозначения внутренней переменной объекта будет применяться термин *"переменная объекта"* или *"атрибут"*. Каждый объект имеет свою собственную совокупность атрибутов. Обычно они не изменяются клиентами напрямую, а только с помощью специально предназначенных для этой цели методов классов.

Объект инкапсулирует внутри себя состояние и поведение. Состояние описывается атрибутами объекта, а поведение характеризуется методами. Снаружи клиенты могут узнать только о поведении объектов. Изнутри доступна полная информация о том, как методы обеспечивают необходимое поведение, изменяют состояние и взаимодействуют другими объектами.

2. Разновидности классов

Большинство классов, использующихся в ООП, можно разделить на 4 категории (хотя, конечно, существуют и другие, более редкие категории классов):

- управление данными (менеджеры данных, Data Managers);
- источники и приемники данных (Data Sources и Data Sinks);
- классы для просмотра данных (Views);

- вспомогательные классы, упрощающие проектирование программ (Facilitators).

В некоторых случаях оказывается, что класс относится сразу к двум категориям. Тогда, возможно, его следует разбить на два класса.

Основная обязанность классов для управления данными – хранение и организация доступа к информации о состоянии чего-либо. Например, для абстрактной модели карточной игры основная задача класса `CCard` – хранение масти и ранга (достоинства) карты. Классы-менеджеры данных являются фундаментальными блоками программы. В спецификации проекта прототипами таких классов обычно являются существительные.

Классы-источники данных служат для генерации данных по запросам других классов. Пример подобного класса – генератор случайных чисел. Приемники данных предназначены для приема и дальнейшей передачи данных (например, это класс для записи данных в файл). В отличие от менеджеров данных, источники и приемники не хранят данные внутри себя неопределенно долго, а генерируют (источники данных) или обрабатывают данные (посредники данных) по запросу.

Классы для просмотра данных используются практически во всех программах, осуществляющих вывод информации на экран. Исходный текст для реализации этих классов обычно является сложным, часто модифицируется и в значительной степени не зависит от содержания отображаемых данных.

При проектировании программ принято отделять классы, содержащие специфические данные программы (модель данных, например, сведения о элементах чертежа в чертежном редакторе) от классов отображения этих данных (которые выполняют непосредственное рисование чертежа на экране). Это упрощает многократное использование кода, поскольку одни и те же классы отображения можно применять во многих программах. Часто модель может иметь несколько визуальных представлений. Например, финансовую информацию о курсе валют можно представить в виде гистограмм, или круговых диаграмм, или таблиц.

К категории вспомогательных классов относятся те классы, которые не содержат важной для программы информации, но облегчают выполнение сложных операций. Например, при отображении игральной карты на экране может использоваться вспомогательный класс, рисующий линии и текст на экране. Другой служебный класс может, например, обслуживать связный список для хранения карт (колоду).

3. Учебный пример: класс "карта" для карточной игры

Рассмотрим, какие переменные для представления состояния и методы для реализации поведения необходимы в абстрактном классе "карта" `CCard`, который может использоваться в произвольной карточной игре.

На рис. 3.1 показана CRC-карточка, описывающая состояние и поведение игральной карты. Поскольку пока не уточняется структура конкретной программы, в которой используется класс `CCard`, нельзя указать сотрудничающие с классом компоненты. Обязанности класса `CCard` очень ограничены, он является менеджером данных, описывающих состояние игральной карты – масть, ранг, расположение на столе. Поведение класса состоит из методов, связанных с отображением карты на экране и изменением состояния.

Класс CCard	**Сотрудничающие компоненты**
Хранит масть и ранг карты	
Возвращает цвет карты	
Хранит состояние "картинка вверх" или "картинка вниз"	
Рисует карту на экране	
Удаляет карту с экрана	

Рис. 3.1. CRC-карточка класса `CCard`.

CRC-карточки многократно уточняются и переписываются, постепенно эволюционируя от естественного языка к тексту программы. После определения обязанностей класса надо выбрать имена для его методов и указать для них список параметров. Описание может не поместиться на одну карточку, тогда придется скреплять скрепками несколько карточек (или заменить их на обычные листы бумаги).

На рис. 3.2 показана CRC-карточка класса `CCard` с записанными именами методов. Обратите внимание, что даже если обязанность состоит всего лишь в возврате значения (например, признака "картинка вверх"), все равно для этого определяется специальный метод. Более подробно необходимость введения посредников для доступа к атрибутам класса будет обоснована позже.

На оборотной стороне CRC-карточки можно записать имена и типы атрибутов данного класса. Затем надо переходить от CRC-карточки к описанию поведения и состояния на языке программирования.

Класс CCard **(описание методов)**	**Сотрудничающие компоненты**
Suit – возвращает масть карты	
Rank – возвращает ранг карты	
Color – возвращает цвет карты	
Draw, Erase – рисует или удаляет образ карты с экрана	
IsFaceUp, Flip – проверяет или изменяет состояние "картинка вверх"	

Рис. 3.2. Уточненная CRC-карточка класса `CCard`.

4. Две части описания класса: интерфейс и реализация

ООП является развитием идей модульности и скрытия информации. Принципы Парнаса ("ничего лишнего") применимы к ООП в той же мере, что и к модульному подходу. Их можно записать в терминах объектов следующим образом:

- объявление класса должно обеспечивать клиента информацией, минимально необходимой для использования класса.
- методам класса должна быть доступна вся информация, минимально необходимая для выполнения обязанностей класса.

Принципы Парнаса делят класс на две части: *интерфейс* (interface) и *реализацию* (implementation). Интерфейсная часть видна пользователю объекта. Она описывает, как объект взаимодействует с внешним миром. Пользователю разрешен доступ только к тем атрибутам и методам класса, которые описаны в интерфейсной части. Реализация определяет, как именно достигается выполнение обязанностей, объявленных в интерфейсной части.

5. Объявление класса в языке Си++

5.1 Синтаксис объявления класса

Интерфейсная часть класса располагается в заголовочном файле с расширением "`*.h`", а реализация – в файле с расширением "`*.cpp`". В одном заголовочном файле могут находиться описания нескольких классов, но обычно так бывает только если классы тесно связаны.

Объявление класса начинается со служебного слова `class` (см. фрагмент программы 3.1). Описание класса очень похоже на описание структуры, только вместе с полями данных стоят заголовки функций. Служебное слово `private:` предшествует фрагментам кода, доступ к которым разрешен только изнутри самого класса (закрытая часть описания класса). Служебное слово `public:` обозначает открытую интерфейсную часть класса, доступную его пользователям. То, что в интерфейсном файле содержится описание закрытой части класса и оно видно пользователям, является нарушением принципа Парнаса. Так приходится поступать, поскольку компилятору требуется знать, сколько памяти требуется для объекта класса, а это можно определить, только если известен размер всех переменных-атрибутов – и общедоступных, и закрытых.

В терминологии Си++ атрибуты класса называются *переменными-членами класса*, а методы – *функциями-членами класса*.

```
// Константы для обозначения карточной масти
enum Suit { Heart /* черви */, Club /* трефы */,
            Diamond /* бубновая масть */, Spade /* пики */ };
// Константы для обозначения цвета карт
enum Color { Red, Black };

class CCard {
  public:
    // Конструктор
    CCard( Suit, int );
    // Доступ к атрибутам карты
    Color GetColor();
    bool  IsFaceUp();
    int   GetRank();
    Suit  GetSuit();
    // Выполняемые картой действия
    void  Draw( CWindow& wnd, int x0, int y0 );
    void  Erase( CWindow& wnd );
    void  Flip();

  private:
    bool isFaceUp;      // Состояние "картинка вверх"/"картинка вниз"
    int  rank;          // Ранг карты
    Suit suit;          // Масть карты
};
```

Фрагмент программы 3.1. Описание класса `CCard` (хранится в файле `card.h`).

Пользователи чаще интересуются открытой областью класса, поэтому в описании она должна идти первой. Чтобы сделать описание более читаемым, надо использовать комментарии, табуляцию, группирование атрибутов и функций-членов по смыслу.

Метод `CCard(Suit, int)` является в нескольких отношениях особенным. У этого метода имя совпадает с именем класса, и у него нет возвращаемого значения. Этот метод называется *конструктором* класса. Он используется для инициализации создаваемых объектов класса (конструкторы будут рассматриваться позже).

Методу `Draw()` в качестве параметра передается ссылка на некий класс `CWindow`, описывающий окно, в котором надо выполнить рисование карты. Классы обычно передаются по ссылке, а не по значению, т.к. имеют значительный размер.

Атрибуты и функции-члены не могут иметь одинаковых имен. Поэтому переменная для хранения масти карты имеет имя `suit`, а функция-член для получения этого значения – имя `GetSuit`. Аналогично, имена `rank` и `GetRank` нужны для хранения и получения ранга карты.

Файл реализации для класса `CCard` должен обеспечить работу методов, описанных в интерфейсном файле. Начало файла реализации для класса `CCard` показано во фрагменте программы 3.2.

```
#include "card.h"

CCard::CCard( Suit sv, int rv )
{
  suit = sv;
  rank = rv;
  isFaceUp = true;
}

int CCard::GetRank()
{
  return rank;
}

Color CCard::GetColor()
{
  return (Color)( suit % 2 );   // Очень плохой способ реализации
}
```

Фрагмент программы 3.2. Файл реализации класса `CCard` (`card.cpp`).

Тело функции-члена записывается как обычная функция Си++, но имени функции-члена предшествует имя класса и два двоеточия. На атрибуты класса внутри функций-членов можно ссылаться как на обычные переменные. Комбинация имени класса и имени функции-члена образует полное имя функции-члена. Его можно рассматривать как аналоги имени и фамилии при идентификации личности.

5.2 Встраиваемые функции

Очень короткие функции, в которых нет ни условных операторов, ни циклов, в Си++ можно определить как *встраиваемые*. Синтаксически обращение к встраиваемой функции выглядит совершенно аналогично обращению к обычной функции. Единственная разница состоит в том, что компилятор имеет право (но не обязан) пре-

образовать вызов встраиваемой функции непосредственно в код в точке ее вызова, сокращая тем самым расходы на обращение к функции и возврат управления. Например, обращение к функции, состоящей из единственного оператора `return`, может занять больше времени, чем выполнение этого оператора. Встраиваемые функции позволяют избежать подобной проблемы. Для определения встраиваемой функции или функции-члена применяется служебное слово `inline`:

```
inline int CCard::GetRank()
{
  return rank;
}
```

Встраиваемые функции-члены можно записывать непосредственно в описании класса, как показано во фрагменте 3.3. При интенсивном использовании встраиваемых функций вполне реально, что файл реализации окажется короче файла с интерфейсом. Однако при таком подходе определение класса становится более трудным для чтения и поэтому должно использоваться только тогда, когда функций-членов немного, а их код очень короткий.

```
class CCard {
  public:
    // Конструкторы
    CCard( Suit, int );
    CCard();
    CCard( const CCard& );

    // Доступ к атрибутам карты
    Color GetColor();
    bool  IsFaceUp()  { return isFaceUp; }
    int   GetRank()   { return rank; }
    Suit  GetSuit()   { return suit; }
    // Выполняемые картой действия
    void  Draw( CWindow& wnd, int x0, int y0 );
    void  Erase( CWindow& wnd );
    void  Flip()      { isFaceUp = !isFaceUp; }

  private:
    bool isFaceUp;    // Состояние "картинка вверх"/"картинка вниз"
    int  rank;        // Ранг карты
    Suit suit;        // Масть карты
};
```

Фрагмент программы 3.3. Описание класса `CCard` с применением встраиваемых функций.

5.3 Полиморфизм методов

Полиморфизм (polymorphism) – это свойство, которое позволяет одно и то же имя использовать для решения нескольких схожих, но технически (на уровне реализации) разных задач. В Си++ можно использовать одно имя функции для множества различных действий. Это называется *перегрузкой функций*. Перегруженные функции отличаются друг от друга количеством и типом параметров. Компилятор организует вызов соответствующей функции после анализа количества и типов переданных параметров.

Например, в классе `CCard` можно предусмотреть две функции для рисования карты, которые различаются способом указания левого верхнего угла карты: или двумя отдельными значениями координат, или с помощью объекта класса `CPoint`:

```
class CCard {
  public:
    ...
    void Draw( CWindow& wnd, int x0, int y0 );
    void Draw( CWindow& wnd, CPoint& topLeft );
    ...
};
```

Перегруженные имена функций – это, главным образом, удобство записи. Оно имеет большое значение для функций с общепринятыми именами вроде `print` и `open`. При вызове функции `f` компилятор должен понять, к какой из функций с именем `f` следует обратиться. Для этого типы всех фактических параметров сравниваются с типами формальных параметров всех функций с именем `f`. В результате вызывается функция, которая лучше всего совпадает с параметрами, или выдается ошибка во время компиляции, если никакая из функций не подходит.

5.4 Перегруженные операторы

Перегрузка операторов, наряду с перегрузкой функций, является проявлением полиморфизма в Си++. Фактически полиморфизм в арифметических операторах применяется почти во всех языках программирования. Например, в языке Си символ `+` используется для сложения целых, длинных целых, символьных переменных и чисел с плавающей точкой. В Си++ можно применять эту концепцию к любым типам данных, созданным программистом. Кроме арифметических, можно определять еще и логические операции, операции сравнения, присваивания `=`, вызова `()`, индексирования `[]` и разыменования `->`.

То, что можно определять, как действуют операции на объекты классов, помогает программисту организовать работу с объектами классов в программе более удобным способом по сравнению с тем, чего можно было бы достичь с использованием лишь функций.

Например, простой класс комплексного числа с операциями сложения и умножения можно объявить следующим образом:

```
class CComplex {
  public:
    CComplex( double r, double i )    { re = r; im = i; }
    CComplex operator+(CComplex);
    CComplex operator*(CComplex);
  private:
    double re, im;
};
```

Программист задает смысл операций `+` и `*` путем определения функций-членов класса с именами `operator+` и `operator*`. Если, например, даны объекты `b` и `c` класса `CComplex`, то выражение `b+c` означает (по определению) вызов функции-члена `b.operator+(c)`. Т.е. теперь есть возможность приблизиться в тексте программы к общепринятой интерпретации комплексных выражений.

В файле реализации описание оператора сложения будет выглядеть так:

```
CComplex CComplex::operator+(CComplex c1)
{
  return CComplex( re + c1.re, im + c1.im );
}
```

6. Упражнения

1) Цифровой счетчик – это целочисленная переменная с ограниченным диапазоном значений, которая сбрасывается при достижении максимального значения. Примеры использования: цифровые часы, счетчик метров в индикаторе пробега. Опишите класс для такого счетчика. Предусмотрите возможность установления минимальных и максимальных значений, увеличения значения счетчика на единицу, возвращения текущего значения.
2) Определите класс для дробей – рациональных чисел, являющихся отношением двух целых чисел. Напишите функции-члены для сложения, вычитания, умножения и деления дробей. Определите вариант класса, в котором эти действия оформлены в виде перегруженных операторов.
3) В классе `CCard` (см. фрагмент программы 3.2) для определения цвета карты по ее масти используется целочисленное деление. Опишите достоинства и недостатки этого приема. Перепишите соответствующий метод так, чтобы убрать зависимость от конкретных значений, приписанных мастям карт.
4) Рассмотрите две показанные ниже комбинации класса и функции. Объясните разницу в применении функции `addi()` с точки зрения пользователя. Можно ли заменить эту функцию перегруженным оператором?

```
class example1 {
  public :
    int i;
};

int addi( example1& x, int j)
{
  x.i += j;
  return x.i;
}
```

```
class example2 {
  public :
    int i;

    int addi( int j )
      { i += j; return i; }
}
```

Лекция 4. Создание объектов и пересылка сообщений

1. Синтаксис пересылки сообщений

Термин *"пересылка сообщения"* обозначает процесс обращения к объекту с требованием выполнить определенное действие. В 1-й лекции были отмечены основные отличия пересылки сообщения от обычного вызова процедуры:

- Сообщение всегда обращено к некоторому объекту, который называется *получателем* или *адресатом*.
- Действие, выполняемое в ответ на сообщение, не является фиксированным и может варьироваться в зависимости от класса получателя. Различные объекты, принимая одно и то же сообщение, выполняют различные действия (это свойство ООП называется полиморфизмом).

В процессе пересылки сообщения выделяются три компоненты: получатель (объект, которому посылается сообщение), название сообщения и список параметров, которые сопровождают сообщение и используются при его обработке.

В Си++ пересылка сообщения называется *вызовом функции-члена*. Описание класса похоже на описание структуры. Синтаксис вызова функции-члена аналогичен обращению к полям данных (атрибутам объекта):

```
<имя объекта-получателя>.<название сообщения>(<список параметров>);
```

Например, если есть объект `theCard` класса `CCard`, то для отображения этой игральной карты в окне `win` в точке с координатами (50, 45), надо вызвать функцию-член так:

```
theCard.Draw( win, 50, 45 );
```

Как и при вызове обычных функций в Си++, если у функции-члена нет параметров, все равно требуется указывать круглые скобки. Благодаря такой записи компилятор отличает вызов функций-членов от обращения к полям данных. Например, проверка расположения карты может быть выполнена следующим образом:

```
if ( theCard.IsFaceUp() )
  {
    ...
  }
else
  {
    ...
  }
```

Внутри каждой функции-члена доступна предопределенная переменная-указатель `this`, которая указывает на объект-получатель (т.е. объект, у которого была вызвана функция-член). Например, функцию-член `CCard.GetColor()` можно реализовать так:

```
Color CCard::GetColor()
{
  if ( this->GetSuit() == Heart || this->GetSuit() == Diamond )
    return Red;
  return Black;
}
```

Но в приведенном примере по правилам Си++ использование `this` является избыточным, т.к. вызов функции-члена без явного указания получателя или обраще-

ние к атрибуту интерпретируются как обращение к текущему получателю. Поэтому обычная форма записи для функции-члена `GetColor()` следующая:

```
Color CCard::GetColor()
{
  if ( GetSuit() == Heart || GetSuit() == Diamond )
    return Red;
  return Black;
}
```

Тем не менее, переменная `this` часто используется, если внутри функции-члена надо передать получателя сообщения в качестве параметра другой функции, например:

```
void COneClass::aMessage( CAnotherClass& ca, int x )
{
  // Передать текущий объект в качестве параметра некоторой
  // функции-члену объекта ca класса CAnotherClass
  ca.doSomething( this, x );
}
```

2. Создание, инициализация и удаление объектов

В Си++ объекты, как и переменные других типов, могут создаваться автоматически или динамически. Автоматическая переменная создается компилятором пир входе в ее область видимости, а при выходе из нее удаляется (т.е. освобождается занятая переменной память).

При автоматическом создании объектов Си++ обеспечивает неявную инициализацию. Это делается с помощью *конструкторов*. Конструктор – это функция-член, имя которой совпадает с именем класса. У конструктора нет возвращаемого значения. Конструктор неявно вызывается каждый раз, когда создается объект данного класса. Это происходит или при объявлении автоматической переменной-объекта, или при вызове оператора `new` для динамического создания объекта.

Рассмотрим следующее описание класса "комплексное число" (приведена часть этого описания):

```
class CComplex {
  public:
    CComplex()
      { re = im = 0.0; }
    CComplex( double re_val )
      { re = re_val; im = 0; }
    CComplex( double re_val, double im_val )
      { re = re_val; im = im_val; }
    ...
  private:
    double re, im;
};
```

Как и обычные функции и функции-члены, конструкторы в Си++ могут быть перегруженными. В приведенном примере задаются три конструктора класса, которые оформлены как встраиваемые функции. Эти конструкторы позволяют инициализировать комплексное число, задавая только вещественную часть или обе части числа. Конструктор по умолчанию (без параметров) приравнивает обе части числа нулю.

Ниже приведены примеры создания автоматических переменных-объектов класса `CComplex` с помощью различных конструкторов:

```
CComplex pi = 3.14159;   // Конструктор с одним параметром
CComplex e( 2.71 );      // Конструктор с одним параметром
CComplex i(0, 1);        // Конструктор с двумя параметрами
CComplex c;              // Конструктор без параметров
```

При динамическом создании объектов используется оператор `new`, за которым следует имя класса и параметры конструктора, например:

```
CComplex* pc = new CComplex( 3.14159, -1.0 );
```

Можно создать массив объектов, если указать его размер в квадратных скобках. Инициализация элементов массива производится с помощью конструктора "по умолчанию" (конструктора без параметров):

```
CComplex c_arr1[25];
CComplex* c_arr2 = new CComplex[25];
```

Динамически созданные объекты, как и обычные динамические переменные, надо удалять с помощью оператора `delete` (или `delete[]` в случае массива объектов). В классе можно определить специальную *функцию-деструктор*, которая будет автоматически вызываться при удалении объекта. Имя деструктора совпадает с именем класса, но ему предшествует символ "тильда" (~). У деструктора нет ни параметров, ни возвращаемого значения.

В следующем примере показано применение конструктора и деструктора в служебном трассировочном классе (который выдает в стандартный поток вывода сообщения, сигнализирующие о порядке выполнения программы). Класс `CTrace` позволяет отследить, в каком порядке вызываются функции и какие блоки операторов выполняются внутри них. Конструктор класса получает указатель на строку с названием трассировочной точки. В конструкторе создается внутренняя копия сообщения, которая хранится внутри объекта. Печать сообщения выполняется дважды: в конструкторе, когда создается объект `CTrace`, и в деструкторе, когда этот объект удаляется. Удаление объекта происходит при выходе из области видимости трассировочного объекта. В деструкторе также производится удаление внутренней для объекта копии сообщения.

```
class CTrace {
  public:
    CTrace( char* );
    ~CTrace();
  private:
    char* msg;
};

CTrace::CTrace( char* s )
{
  msg = new char[ strlen(s) + 1 ];
  strcpy( msg, s );
  cout << "Начало фрагмента " << msg << "\n";
}

CTrace::~CTrace()
{
  cout <<"Конец фрагмента " << msg << "\n";
  delete msg;
}
```

В отладочных целях программист размещает в каждой функции, подлежащей трассировке, описание объекта класса `CTrace` с соответствующим сообщением. Этому объекту не посылается никаких сообщений, т.к. его назначение – только обозначить в трассировочном протоколе вход и выход в данную область видимости. Рассмотрим две функции:

```
void functionA()
{
  CTrace t( "functionA" );
  functionB( 10 );
}

void functionB( int x )
{
  CTrace t( "functionB" );
  if ( x < 5 )
    {
    CTrace t( "functionB: ветвь для обработки x < 5" );
    ...
    }
  else
    {
    CTrace t( "functionB: ветвь для обработки x >= 5" );
    ...
    }
  ...
}
```

Если в программе сначала вызывается функция `functionA()`, то в трассировочном протоколе объекты класса `CTrace` покажут порядок выполнения функций `functionA()` и `functionB()` следующим образом:

```
Начало фрагмента functionA
Начало фрагмента functionB
Начало фрагмента functionB: ветвь для обработки x >= 5
Конец фрагмента functionB: ветвь для обработки x >= 5
Конец фрагмента functionB
Конец фрагмента functionA
```

3. Учебный пример: задача о восьми ферзях

В шахматах ферзь может бить любую фигуру, стоящую с ним на одной строке, столбце или диагонали. Задача о восьми ферзях ставится следующим образом: надо расставить восемь ферзей на шахматной доске, чтобы ни один ферзь не угрожал другому. Эта задача является хорошо известным примером использования метода проб и ошибок и алгоритмов с возвратом. У нее 92 решения, но принципиально различных решений только 12, остальные решения оказываются симметричными с ними относительно поворотов. Одно из решений показано на рис. 4.1.

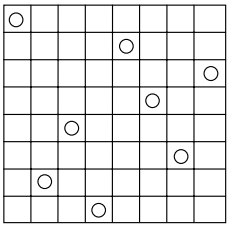

Рис. 4.1. Одно из решений задачи о восьми ферзях.

При традиционном программировании для описания позиций фигур использовалась бы какая-нибудь структура данных, и программа решала бы задачу, перебирая значения в этой структуре и проверяя каждую позицию: не является ли она решением?

При объектно-ориентированном подходе нахождение решения надо запрограммировать как динамическую модель, состоящую из набора однотипных агентов – фигур, обладающих вполне определенным поведением и взаимодействующих друг с другом. Когда состояние модели стабилизируется, решение будет найдено.

3.1 Идентификация компонент

В данной задаче выделяются объекты одного класса – ферзи, которые совместно должны прийти к решению задачи. В любом случае никакие два ферзя не могут занимать один столбец, следовательно, все столбцы заняты. Поэтому каждому ферзю можно сразу присвоить определенный столбец и свести задачу к более простой – поиску соответствующих строк.

Относительно взаимодействия ферзей можно сделать важное упрощающее замечание: каждый ферзь должен знать только о своем соседе слева. Т.о., данные о ферзе состоят из трех значений: столбец, который остается неизменным, строка, которую надо подобрать в процессе решения, и сосед слева.

Будем считать приемлемым решением для столбца N такую конфигурацию столбцов с 1 по N, в которой ни один ферзь из этих столбцов не бьет другого. Каждому ферзю будет поручено найти приемлемое решение для себя и своих соседей слева. Решение всей задачи будет получено, когда самый правый ферзь отыщет приемлемое решение.

Класс CQueen: методы	**Класс CQueen: атрибуты**
`Initialize` – инициализировать строку, затем найти первое приемлемое для себя и соседа решение `Advance` – перейти к следующей строке и найти приемлемое решение `CanAttack` – проверить, может ли заданная позиция быть атакована данным ферзем или его соседями	`row` – номер текущей строки (изменяется в процессе решения) `column` – номер столбца (постоянен) `neighbor` – сосед слева (постоянен)

Рис. 4.2. Две стороны CRC-карточки для класса `CQueen`.

Учитывая все перечисленные соображения, CRC-карточку для класса `CQueen` (Ферзь) можно записать так, как показано на рис. 4.2. На лицевой стороне описаны методы класса, на оборотной – атрибуты класса.

3.2 Описание класса CQueen

Ниже приведено описание класса `CQueen`. Каждый объект этого класса хранит указатель `pNeighbor` на другого ферзя-соседа слева (у крайнего слева ферзя `pNeighbor=0`). Метод `Initialize` реализован в виде конструктора.

```
class CQueen {
  public:
    CQueen( int, CQueen* );

    CQueen* GetNeighbor()  { return pNeighbor; }
    bool    FindSolution();
    bool    Advance();
    void    Print();

  private:
    const int column;      // Номер столбца (от 1 до 8)
    int row;               // Номер строки (от 1 до 8)
    CQueen* pNeighbor;     // Указатель на ферзя-соседа слева

    bool CanAttack( int r, int c );
};
```

Переменная-член `column` (номер столбца) описана как константа. Ее значение задается в конструкторе и больше не изменяется. Т.к. константу нельзя использовать в левой части оператора присваивания, то для инициализации `column` надо воспользоваться инициализирующей записью в заголовке конструктора. При такой записи в конструкторе после заголовка функции ставится двоеточие и через запятую перечисляются атрибуты класса со своими начальными значениями в круглых скобках.

```
CQueen::CQueen( int col, CQueen* pNgb ) :
        column( col ),
        pNeighbor( pNgb ),
        row( 1 )
{
}
```

3.3 Функции-члены класса CQueen

Для поиска решения каждый ферзь, в порядке справа-налево, опрашивает своих соседей, могут ли они его атаковать. Если да, то ферзь продвигается вниз (или возвращается признак "нет решения", если дальнейшее перемещение невозможно). Если соседи сообщают, что они атаковать не могут, то решение найдено. Это решение – "локальное", верное только для текущего ферзя. Общее решение компонуется из локальных решений для всех ферзей. Поиск локального решения выполняет функция-член `FindSolution()`:

```
bool CQueen::FindSolution()
{
  if ( pNeighbor )
    if ( pNeighbor->CanAttack( row, column ) )
      return Advance();

  return true;
}
```

Проверка потенциального решения выполняется с помощью функции-члена `CanAttack()`, которая определяет, будет ли заданный ферзь или один из его соседей слева бить заданную клетку. В реализации используется тот факт, что при движении по диагонали смещение по строкам равно смещению по столбцам.

```
bool CQueen::CanAttack( int r, int c )
{
  // Проверка на ту же строку
  if ( row == r )
    return true;

  // Проверка диагонали
  int deltaCol = c - column;
  if ( ( row + deltaCol == r ) || ( row - deltaCol == r ) )
    return true;

  // Проверка, могут ли бить клетку (r, c) соседи
  if ( pNeighbor )
    return pNeighbor->CanAttack( r, c );

  return false;
}
```

Перемещение ферзя на следующую позицию выполняет функция-член `Advance()`. В ней можно выделить две части. Если ферзь не достиг последней строки, то эта функция просто увеличивает номер строки и выполняет поиск решения. Если ферзь достиг последней строки, то он пытается попросить соседа слева сместиться на одну строку и затем снова найти решение, начиная с 1-й строки.

```
bool CQueen::Advance()
{
  // Пробуем переместиться на следующую строку
  if ( row < 8 )
  {
    row++;
    return FindSolution();
  }

  // Если ферзь стоит на последней строке, то надо
  // попытаться сдвинуть соседа к следующему решению
  if ( pNeighbor )
    if ( pNeighbor->Advance() )
      {
      // Начинаем снова с 1-й строки
      row = 1;
      return FindSolution();
      }

  return false;
}
```

После того, как решение найдено, его надо напечатать. Это делается путем прохода по связному списку ферзей в функции `Print()`:

```
void CQueen::Print()
{
  if ( pNeighbor )
    pNeighbor->Print();
  cout << "Строка: " << row << "; Столбец: " << column << "\n";
}
```

3.4 Главная программа

Т.к. инициализация выполняется в конструкторе, то основная программа может просто создать восемь объектов-ферзей, по одному в каждом столбце, и потом напечатать решение. В показанной далее функции `main()` переменная `pLastQueen` указывает на последнего созданного ферзя (крайнего справа). За счет того, что внутри `CQueen` есть указатель на соседа, ферзи образуют связный список. Явное обращение к этому списку производится при удалении динамически созданных объектов в конце функции `main()`.

```
void main()
{
  CQueen* pLastQueen = NULL;

  for ( int i = 1; i <= 8; i++ )
  {
    // Создать и инициализировать нового ферзя в i-м столбце
    pLastQueen = new CQueen( i, pLastQueen );
    if ( !pLastQueen->FindSolution() )
      cout << "Нет решения.\n";
  }

  // Печать решения
  pLastQueen->Print();

  // Удаление объектов-ферзей
  while ( pLastQueen )
  {
    CQueen* pPrevQueen = pLastQueen->GetNeighbor();
    delete pLastQueen;
    pLastQueen = pPrevQueen;
  }
}
```

4. Упражнения

1) Создайте в MS Visual C++ проект консольного приложения и добавьте в этот проект файлы `queen.h` (объявление класса `CQueen`), `queen.cpp` (реализация класса `CQueen`) и `testprg.cpp` (главная программа). Распределите по этим файлам компоненты программы, приведенные в лекции. Запустите программу и убедитесь, что она находит одно из решений задачи о восьми ферзях.
2) Измените программу так, чтобы она выдавала все возможные решения задачи о восьми ферзях. Как можно отбросить решения, которые являются поворотами других решений?
3) Как изменится программа, если обобщить задачу на случай N ферзей? Т.е. как найти возможные расположения N ферзей на шахматной доске размерами NxN? Существуют некоторые N, для которых вообще нет решений (например, N=2 и N=3). Что будет выдавать программа в подобных случаях?
4) Измените программу так, чтобы она динамически изображала на шахматной доске расположение каждого ферзя во время своей работы. Для отображения ферзей используйте библиотеку **OpenGL**. Какие фрагменты программы оказываются ответственными за отображение?

   Рекомендации: *ферзей можно изображать условно, в виде пирамид или параллелепипедов. Для организации задержки между перемещениями ферзей можно использовать функцию* `Sleep(int dt)` *из Win32 API (параметр* `dt` *задает задержку выполнения программы в миллисекундах).*

Лекция 5. Учебный пример: игра "Бильярд"

В данной лекции рассматривается программа, имитирующая движение шаров по бильярдному столу со стенками и лузами. Чтобы разобраться в этой программе, надо изучить лекции 1-4 по ООП и знать основные методы работы с библиотекой **OpenGL**. **OpenGL** применяется в несколько необычном для ОС **Windows** качестве: как библиотека двумерной графики. Это позволяет существенно уменьшить исходный текст программы по сравнению с вариантом программы, который для рисования пользуется функциями **Windows API**.

Основное внимание при разработке программы "Бильярд" уделяется созданию автономных агентов, взаимодействующих между собой для достижения желаемого результата – имитации движения шаров по бильярдному столу.

1. Описание модели бильярда

Программа отображает на экране окно, в котором в виде прямоугольника изображен бильярдный стол. В его 4-х углах находятся лузы. На столе расположены 15 синих шаров и 1 белый шар (рис. 5.1).

Щелчком левой кнопки мыши в произвольной точке стола пользователь имитирует удар кием по белому шару, сообщая ему некоторую начальную скорость в направлении указателя мыши. Движущийся шар упруго соударяется со стенками и другими шарами. При движении по столу шары теряют энергию за счет трения и со временем останавливаются. При попадании в лузу синие шары удаляются со стола, а белый помещается в начальную позицию (рис. 5.1).

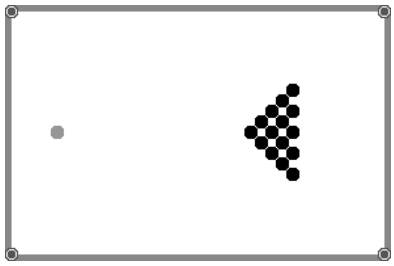

Рис. 5.1. Изображение бильярдного стола с 4-мя угловыми лузами.

2. Основные классы модели

Основными классами в данной задаче являются стенки, лузы и шары. Объектам этих классов приписана некоторая экранная область и они умеют изображать себя внутри этой области. Будем называть эти объекты *графическими объектами*. Графические объекты каждого типа хранятся в программе в виде связных списков. Для этого в каждом классе есть указатель на следующий объект списка.

Размещение ссылок внутри объектов для объединения этих объектов в некоторый абстрактный тип данных (например, связный список) считается плохим стилем программирования. Лучше отделить реализацию АТД от объектов, которые в нем хранятся. Но решение этой задачи содержит некоторые нетривиальные аспекты, которые усложнили бы рассматриваемую программу.

Примем упрощающее предположение, что все графические объекты занимают прямоугольную область. Конечно, для круглых объектов (например, шаров и луз) это неверно. Но учет реальной формы объектов значительно усложнил бы программу.

Главные цели в ее изучении – рассмотрение способа, которым объекты наделяются собственным поведением и как организовано взаимодействие объектов. Каждый графический объект не только умеет изображать себя, но и может взаимодействовать с другими объектами модели бильярда.

2.1 Класс CRect (прямоугольник)

Класс `CRect` является вспомогательным классом, который предназначен для хранения координат прямоугольника и выполнения некоторых операций с этими координатами. Операции с координатами прямоугольников применяются при реализации поведения графических объектов. Далее приведено объявление класса, в котором большая часть функций-членов сделаны встраиваемыми.

```
class CRect {
  public :
    // Конструкторы
    CRect()
        { x1 = y1 = x2 = y2 = 0; }
    CRect( int nx1, int ny1, int nx2, int ny2 )
        { x1 = nx1; y1 = ny1; x2 = nx2; y2 = ny2; }

    // Инициализация координат прямоугольника
    void SetRect( int left, int top, int right, int bottom )
        { x1 = left; y1 = top; x2 = right; y2 = bottom; }
    // Смещение прямоугольника
    void OffsetRect( int dx, int dy )
        { x1 += dx; y1 += dy; x2 += dx; y2 += dy; }
    // Получение координат центра прямоугольника
    void CenterPoint( int* x, int* y )
        { *x = (x1 + x2)/2; *y = (y1 + y2)/2; }
    // Проверка на совпадение координат углов прямоугольника
    bool IsRectEmpty()
        { return ( x1 == x2 && y1 == y2 ); }
    // Получение ширины прямоугольника
    int Width()       { return x2 - x1 + 1; }
    // Получение высоты прямоугольника
    int Height()      { return y2 - y1 + 1; }
    // Функция вычисляет пересечение объекта с прямоугольником another
    CRect IntersectRect( CRect& another );
    // Нормализация координат прямоугольника
    void NormalizeRect();

    // Общедоступные атрибуты
    int x1, y1, x2, y2;
};
```

Атрибуты прямоугольника (координаты) сделаны общедоступными, т.к. этот класс очень прост, не обладает собственным поведением и нет существенной опасности от несанкционированного изменения координат.

Будем называть прямоугольник нормализованным, если его координаты (x1, y1) и (x2, y2) соответствуют левому верхнему и правому нижнему углу. Такое упорядочение координат упрощает расчет пересечения прямоугольников. Эта операция выполняется функцией-членом `IntersectRect()` и требуется для отслеживания столкновения графических объектов.

2.2 Класс CWall (стенка бильярдного стола)

Класс `CWall` объявляется следующим образом:

```
class CWall {
  public:
    CWall( int left, int top, int right, int bottom,
           double cf, CWall* pNextWall );

    // Рисование стенки
    void Draw();
    // Извещение стенки о том, что в нее попал шар
    void HitBy( CBall* pBall );

    CRect  GetRegion()       { return region; }
    CWall* GetLink()         { return pLink; }

  private:
    CWall* pLink;
    CRect region;            // Экранные координаты стенки
    double convertFactor;    // Значение, из которого вычитается
                             // направление ударившегося шара, чтобы
                             // получилось зеркальное отражение
};
```

Атрибут `pLink` служит для организации связного списка объектов `CWall`. Значение этого атрибута, а также координаты области для рисования стенки и параметр отскока инициализируются в конструкторе:

```
CWall::CWall( int left, int top, int right, int bottom,
              double cf, CWall* pNextWall )
{
  convertFactor = cf;
  region.SetRect( left, top, right, bottom );
  pLink = pNextWall;
}
```

Стенка рисуется на экране как сплошной зеленый прямоугольник. Это делается с помощью функций **OpenGL**:

```
void CWall::Draw()
{
  glColor3ub( 0, 255, 0 );
  glRecti( region.x1, region.y1, region.x2, region.y2 );
}
```

Когда о стенку ударяется шар, то направление его движения изменяется в соответствии со значением атрибута `convertFactor`. Для горизонтальных стенок `convertFactor=0`, а для вертикальных `convertFactor=Pi`. В результате шар зеркально отражается от стенки:

```
void CWall::HitBy( CBall* pBall )
{
  pBall->SetDirection( convertFactor - pBall->GetDirection() );
}
```

2.3 Класс CHole (луза бильярдного стола)

Ниже приведено объявление класса `CHole`:

```
class CHole {
  public:
    CHole( int x, int y, CHole* pNextHole );

    // Рисование лузы
    void Draw();
    // Извещение лузы о том, что в нее попал шар
    void HitBy( CBall* pBall );

    CRect GetRegion() { return region; }
    CHole* GetLink()  { return pLink; }

  private:
    CHole* pLink;     // Указатель на следующую лузу для образования
                      // связного списка
    CRect region;     // Экранные координаты области лузы
};
```

Конструктор и функция-член для рисования лузы устроены очень просто:

```
CHole::CHole( int x, int y, CHole* pNextHole )
{
    // Описывающий прямоугольник для лузы с центром в точке (x, y)
    region.SetRect( x - 5, y - 5, x + 5, y + 5 );
    pLink = pNextHole;
}

void CHole::Draw()
{
    // Луза рисуется в виде желтого круга, вписанного в область region
    glColor3ub( 255, 255, 0 );
    glPointSize( (float)region.Width() );
    glEnable( GL_POINT_SMOOTH );
    glBegin( GL_POINTS );
      int cx, cy;
      region.CenterPoint( &cx, &cy );
      glVertex2i( cx, cy );
    glEnd();
    glDisable( GL_POINT_SMOOTH );
}
```

При попадании шара в лузу возможны два варианта. Если шар оказался белым, то он возвращается на исходную позицию. В остальных случаях шар останавливается и рисуется за пределами стола в строке, предназначенной для отображения выбитых шаров.

```
void CHole::HitBy( CBall* pBall )
{
    // Энергия шара обнуляется
    pBall->SetEnergy( 0.0 );

    if ( pBall->IsCue() )
      pBall->SetCenter( 50, 108 );
    else
      {
```

```
      pBall->SetCenter( 10 + saveRack*15, 250 );
      saveRack++;     // Увеличение глобального счетчика забитых шаров
      }
}
```

2.4 Класс CBall (бильярдный шар)

Последним графическим объектом является шар, определяемый следующим описанием класса:

```
class CBall {
  public:
    CBall( int x, int y, bool fc, CBall* pNextBall );

    // Рисование шара в текущем положении
    void Draw();
    // Изменение положения шара в предположении, что с момента
    // предыдущего изменения прошел единичный промежуток времени
    void Update();
    // Извещение шара о том, что в него попал другой шар
    void HitBy( CBall* pBall );
    // Расчет угла между осью OX и направлением от центра шара до точки,
    // смещенной от центра шара на (dx, dy)
    double HitAngle( double dx, double dy );

    // Функции-члены для доступа к переменным класса
    void   SetEnergy( double v )            { energy = v; }
    void   SetCenter( int newx, int newy );
    void   SetDirection( double newDir )    { direction = newDir; };
    CRect  GetRegion()                      { return region; }
    CBall* GetLink()                        { return pLink; }
    double GetEnergy()                      { return energy; }
    void   GetCenter( int* x, int* y );
    double GetDirection()                   { return direction; }
    bool   IsCue()                          { return fCue; }

  private:
    CBall* pLink;     // Указатель на следующий шар связного списка
    CRect  region;    // Экранная область, в которую вписан шар
    double direction; // Направление движения шара (угол в радианах
                      // относительно оси OX)
    double energy;    // Энергия шара
    bool   fCue;      // Признак белого шара
};
```

Специфическими для шара атрибутами являются `direction` (направление движения), `energy` (кинетическая энергия шара) и флаг `fCue`, который равен `true` для единственного в модели белого шара. Шары инициализируются координатами центра (подобно лузам), признаком белого шара и указателем на следующий шар в списке. Энергия и направление шара первоначально равны нулю.

```
CBall::CBall( int x, int y, bool fc, CBall* pNextBall )
{
  SetCenter( x, y );
  SetDirection( 0 );
  SetEnergy( 0.0 );
  pLink = pNextBall;
  fCue = fc;
}
```

На экране шар рисуется в виде круга белого или синего цвета. Отображение выполняет функция-член `Draw()`.

```
void CBall::Draw()
{
  if ( IsCue() )
    glColor3ub( 255, 255, 255 );     // Белый цвет
  else
    glColor3ub( 0, 0, 255 );         // Синий цвет

  glPointSize( (float)region.Width() );
  glEnable( GL_POINT_SMOOTH );
  glBegin( GL_POINTS );
    int cx, cy;
    region.CenterPoint( &cx, &cy );
    glVertex2i( cx, cy );
  glEnd();
  glDisable( GL_POINT_SMOOTH );
}
```

Функция-член `Update()` используется для изменения положения шара через единичный промежуток времени, прошедший с момента предыдущего вызова этой функции. Если шар обладает достаточной энергией, то он перемещается и затем проверяет, не задел ли он другой объект. Если хотя бы один шар на столе сдвинулся, то глобальная переменная-флаг `fBallMoved` устанавливается равной `true`. Если шар задел другой объект, то шар сообщает об этом объекту. Сообщения бывают трех типов: они соответствуют ударам по лузе, стенке и другому шару.

```
void CBall::Update()
{
  // Для движения у шара должна быть некоторая кинетическая энергия
  if ( energy <= 0.5 )
    return;

  fBallMoved = true;

  // На каждом шаге энергия шара уменьшается за счет трения
  energy -= 0.05;

  // Смещение шара вычисляется с учетом квадратичной зависимости
  // кинетической энергии от скорости. Константа 2.0 выбрана для
  // обеспечения более-менее реалистичного движения шаров
  int dx = (int)( 2.0*sqrt(energy)*cos(direction) );
  int dy = (int)( 2.0*sqrt(energy)*sin(direction) );
  region.OffsetRect( dx, dy );

  // Проверка на попадание в лузу
  CHole* hptr = listOfHoles;
  while ( hptr )
    {
    CRect is = region.IntersectRect( hptr->GetRegion() );
    if ( !is.IsRectEmpty() )
      {
      hptr->HitBy( this );
      hptr = NULL;
      }
    else
```

```
      hptr = hptr->GetLink();
    }

  // Проверка на попадание в стенку
  CWall* wptr = listOfWalls;
  while ( wptr )
    {
    CRect is = region.IntersectRect( wptr->GetRegion() );
    if ( !is.IsRectEmpty() )
      {
      wptr->HitBy( this );
      wptr = NULL;
      }
    else
      wptr = wptr->GetLink();
    }

  // Проверка на попадание в другой шар
  CBall* bptr = listOfBalls;
  while ( bptr )
    {
    if ( bptr != this )
      {
      CRect is = region.IntersectRect( bptr->GetRegion() );
      if ( !is.IsRectEmpty() )
        {
        bptr->HitBy( this);
        break;
        }
      }
    bptr = bptr->GetLink();
    }
}
```

При соударении шаров считается, что шар, по которому производится удар, неподвижен и энергия ударившего шара делится между ними пополам. При ударе также меняются направления движения обоих шаров.

```
void CBall::HitBy( CBall* pBall )
{
  // Уменьшаем энергию ударившегося шара вдвое
  pBall->SetEnergy( pBall->GetEnergy()/2.0 );
  // и прибавляем ее к собственной энергии
  energy += pBall->GetEnergy();

  // Расчет нового направления для текущего шара
  int cx1, cy1, cx2, cy2;
  GetCenter( &cx1, &cy1 );
  pBall->GetCenter( &cx2, &cy2 );
  SetDirection( HitAngle( cx1 - cx2, cy1 - cy2 ) );

  // Модификация направления ударившегося шара
  double da = pBall->GetDirection() - GetDirection();
  pBall->SetDirection( pBall->GetDirection() + da );
}

// Расчет угла между осью OX и вектором (dx, dy). Функция возвращает
// значение угла радианах в диапазоне (0,PI) или (-PI,0)
double CBall::HitAngle( double dx, double dy )
```

```
{
  double na;

  if ( fabs(dx) < 0.05 )
    na = PI/2;
  else
    na = atan( fabs(dy/dx) );

  if ( dx < 0 )
    na = PI - na;
  if ( dy < 0 )
    na = -na;

  return na;
}
```

3. Реализация динамического поведения модели

Динамическое поведение модели бильярдного стола обеспечивается с помощью трех функций обратной связи **OpenGL**: обработчика щелчка левой кнопки мыши, фоновой функции и функции отображения сцены.

В обработчике события мыши `MouseButtonDown()` имитируется удар кием по белому шару. Этому шару приписывается некоторая начальная энергия и направление движение "на указатель мыши".

```
void CALLBACK MouseButtonDown( AUX_EVENTREC* event )
{
  // Запоминание координат указателя мыши в правой системе координат,
  // связанной с нижним левым углом окна (т.к. в структуре event
  // они хранятся в левой системе координат, связанной с левым
  // верхним углом окна)
  int mouse_x = event->data[0];
  int mouse_y = WINDOW_HEIGHT - event->data[1];

  // Белому шару передается некоторая начальная энергия
  pCueBall->SetEnergy( 20.0 );
  // и присваивается направление движения "на указатель мыши"
  int cx, cy;
  pCueBall->GetCenter( &cx, &cy );
  pCueBall->SetDirection( pCueBall->HitAngle( mouse_x-cx, mouse_y-cy ) );

  // Флаг наличия хотя бы одного движущегося шара
  fBallMoved = true;
}
```

Фоновая функция `Idle()` при наличии хотя бы одного движущегося шара (на это указывает глобальная переменная-флаг `fBallMoved`) обновляет положение шаров и целиком перерисовывает сцену. Чтобы шары двигались не слишком быстро, в конце фоновой функции организована небольшая задержка, в течение которой выполнение программы приостанавливается.

```
// Фоновая функция
void CALLBACK Idle()
{
  // Положение шаров обновляется, только если есть хотя бы
  // один движущийся шар
  if ( !fBallMoved )
```

```
    return;

  // Обновление положения шаров
  fBallMoved = false;
  CBall* bptr = listOfBalls;
  while ( bptr )
    {
    bptr->Update();
    bptr = bptr->GetLink();
    }

  Display();     // Отображение сцены
  Sleep( 25 );  // Задержка на 25 мс
}
```

Функция отображения сцены реализована довольно прямолинейно: в режиме двойной буферизации рисуются все графические объекты.

```
void CALLBACK Display()
{
  glClear( GL_COLOR_BUFFER_BIT );

  // Рисование стенок стола
  CWall* pWall = listOfWalls;
  while ( pWall )
    {
    pWall->Draw();
    pWall = pWall->GetLink();
    }

  // Рисование луз
  CHole* pHole = listOfHoles;
  while ( pHole )
    {
    pHole->Draw();
    pHole = pHole->GetLink();
    }

  // Рисование шаров
  CBall* pBall = listOfBalls;
  while ( pBall )
    {
    pBall->Draw();
    pBall = pBall->GetLink();
    }

  auxSwapBuffers();
}
```

4. Упражнения

1) Создайте проект консольного приложения **OpenGL** и добавьте в него файл с исходным текстом программы "Бильярд" (`prg5_1.cpp`).
   Разберитесь в исходном тексте программы. Главное – надо понять, как было децентрализовано управление и как сами объекты были наделены возможностями влиять на ход выполнения программы. Все, что происходит при нажатии кнопки мыши, – это передача некоторой кинетической энергии белому шару. В дальнейшем модель работает исключительно за счет взаимодействия шаров, стенок и луз.

В среде **Visual C++** обратите внимание на содержимое закладки `ClassView` (просмотр классов) в окне проекта. Разберитесь, как в этом списке (рис. 5.2) отображаются классы, функции-члены и переменные-члены, а также глобальные переменные и функции.

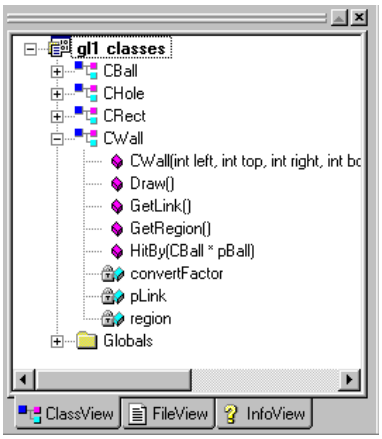

Рис. 5.2. Закладка `ClassView` (просмотр классов) в окне проекта в среде **MS Visual C++**. Показаны имена четырех классов, используемых в программе "Бильярд". Описание класса `CWall` раскрыто для просмотра.

2) Какие части программы надо изменить, чтобы сделать шары цветными? Вместо 15-ти синих шаров сделайте 5 зеленых, 5 синих и 5 красных.

3) Как и на обычном бильярдном столе, добавьте лузы в середине длинных боковых стенок.

Лекция 6. Одиночное наследование

В предыдущих лекциях объяснялось, что при ОО-подходе важно представить задачу в виде взаимодействия программных компонент. В ООП важнейший тип компонент – это объекты классов. В 1-й лекции отмечалось, что классы можно организовать в виде иерархической структуры, основанной на принципе наследования.

Наследование означает, что поведение и данные подкласса являются расширением свойств родительских классов. По существу, подклассы предназначены для того, чтобы программист мог легко выразить сходство классов. Подкласс имеет все свойства родительского класса, и, кроме того, дополнительные свойства. С другой стороны, поскольку подкласс является более специализированной формой родительского класса, он в определенном смысле будет сужением родительского класса. Поэтому наследование является достаточно тонким вопросом: по отношению к родительскому классу в нем присутствует и аспект "расширения", и аспект "сужения".

1. Примеры наследования

Находящиеся в полете космические зонды посылают на наземные станции информацию о состоянии своих основных систем (например, солнечных батарей и двигателей) и измерения датчиков (таких как датчики радиации, масс-спектрометры, телекамеры, датчики столкновений с микрометеоритами и т.д.). Вся совокупность передаваемой информации называется телеметрическими данными. Как правило, они передаются в виде потока данных, состоящего из заголовка (включающего временные метки и коды для идентификации последующих данных) и нескольких пакетов данных от подсистем и датчиков. Все это выглядит как простой набор разнотипных данных, поэтому для описания каждого типа данных телеметрии можно предложить использовать обычные структуры:

```
class CTime ... // Некоторый класс для хранения значения времени
// Структура с параметрами двух солнечных батарей
struct ElectricalData {
  CTime timeStamp;
  int id;
  double fuelCell1Voltage, fuelCell2Voltage;
  double fuelCell1Amperes, fuelCell2Amperes;
  double currentPower;
};
```

У приведенного описания есть ряд очевидных недостатков:

1) Структура `ElectricalData` не защищена, то есть клиент может вызвать изменение такой важной информации, как `timeStamp` или `currentPower` (мощность, развиваемая обеими батареями, которую можно вычислить по значениям тока и напряжения).
2) Т.к. структура является полностью открытой, то любая ее модификация (добавление новых полей в структуру или изменение типа существующих полей) влияет на клиентов. Как минимум, приходится заново компилировать все описания, связанные каким-либо образом с этой структурой. Еще важнее, что внесение в структуру изменений может нарушить логику отношений с клиентами, а следовательно, логику всей программы.

3) Приведенное описание структуры трудно для восприятия. По отношению к такой структуре можно выполнить множество различных действий (пересылка данных, вычисление контрольной суммы для определения ошибок и т.д.), но все они не будут связаны с приведенной структурой логически.

Предположим, что анализ требований к системе обусловил наличие нескольких сотен разновидностей телеметрических данных, включающих показанную выше структуру и другие электрические параметры в разных контрольных точках системы. Очевидно, что описание такого количества дополнительных структур будет избыточным как из-за повторяемости структур, так и из-за наличия общих функций обработки.

Лучше было бы создать для каждого вида телеметрических данных отдельный класс. Это позволит защитить данные в каждом классе и увязать их с выполняемыми операциями. Но этот подход не решает проблему избыточности.

Значительно лучше построить иерархию классов, в которой от общих классов с помощью наследования образуются более специализированные; например, следующим образом:

```
// Родительский класс "телеметрические данные"
class CTelemetryData {
  public:
    CTelemetryData();
    virtual ~CTelemetryData();
    virtual void Transmit();
    CTime CurrentTime();
  protected:
    int id;
    CTime timeStamp;
};
```

Служебное слово `virtual` применяется для создания виртуальных функций-членов, реализация которых может быть изменена в дочерних классах.

У класса `CTelemetryData` есть конструктор, деструктор (который наследники могут переопределить) и функции `Transmit()` и `CurrentTime()`, видимые для всех клиентов. Защищенные элементы `id` и `timeStamp` инкапсулированы – они доступны только классу и его подклассам. Функция `CurrentTime()` сделана открытой, благодаря чему значение `timeStamp` можно читать, но нельзя изменять.

При рассмотрении этого описания надо учитывать, что в Си++ интерфейсная часть описания класса может быть разделена на три составные части:

- открытую (`public`) – видимую всем клиентам;
- защищенную (`protected`) – видимую самому классу и его подклассам;
- закрытую (`private`) – видимую только самому классу.

Теперь рассмотрим, как построить класс `CElectricalData`:

```
class CElectricalData : public CTelemetryData {
  public:
    CElectricalData( double v1, double v2, double a1, double a2);
    virtual ~CElectricalData();
    virtual void Transmit();
    double CurrentPower();
  protected:
```

```
    double fuelCell1Voltage, fuelCell2Voltage;
    double fuelCell1Amperes, fuelCell2Amperes;
};
```

Этот класс – наследник класса `CTelemetryData`, в котором исходный класс дополнен четырьмя новыми переменными, а его поведение переопределено (изменена реализация виртуальной функции `Transmit()` и добавлена функция `CurrentPower()`).

2. Одиночное наследование

Наследование – это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или нескольких других (множественное наследование) классов. Класс, структура и поведение которого наследуются, называется *родительским классом* или *суперклассом*.

Например, `CTelemetryData` является родительским классом для `CElectricalData`. Производный от родительского класса класс называется *подклассом*. Наследование устанавливает между классами иерархию "общего" и "частного". В этом смысле `CElectricalData` является более специализированной разновидностью более общего `CTelemetryData`. В подклассе структура и поведение родительского класса дополняются и переопределяются.

Отношения одиночного наследования от родительского класса `CTelemetryData` показаны на рис. 6.1. Стрелки обозначают отношения общего к частному. Например, `CCameraData` – это разновидность класса `CSensorData`, который в свою очередь является разновидностью класса `CTelemetryData`.

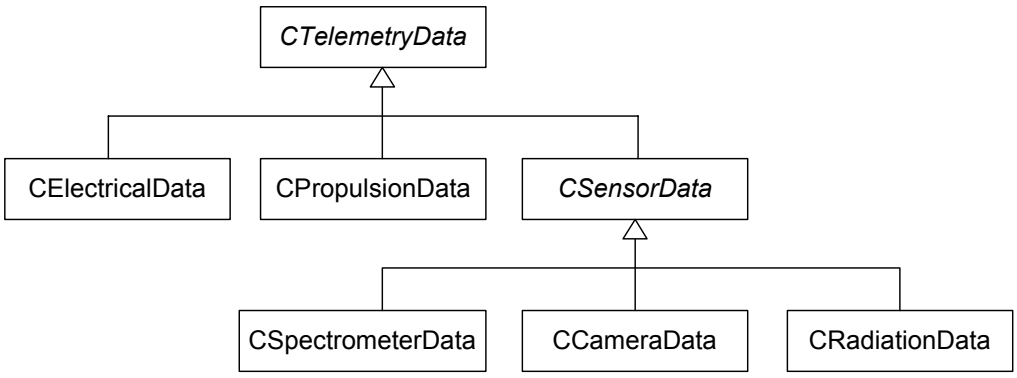

Рис. 6.1. Одиночное наследование.

Можно ожидать, что для некоторых классов на рис. 6.1 будут созданы экземпляры, а для других – нет. Наиболее вероятно образование объектов самых *специализированных классов* `CElectricalData` и `CSpectrometerData`. Образование объектов из классов, занимающих промежуточное положение (`CSensorData` или тем более `CTelemetryData`), менее вероятно. Классы, экземпляры которых не создаются, называются *абстрактными*. Ожидается, что подклассы абстрактных классов доопределят их до жизнеспособной абстракции, наполняя класс содержанием. На диаграммах классов имена абстрактных классов записываются курсивом (рис. 6.1).

Самый общий класс в иерархии классов называется *базовым*. В большинстве программ базовых классов бывает несколько, и они отражают наиболее общие абст-

ракции предметной области. На самом деле, особенно в Си++, хорошо сделанная структура классов – это скорее лес из деревьев наследования, чем одна многоэтажная структура наследования с единственным корнем.

При инициализации дочерних классов для обеспечения наследования конструктор должен вызывать конструктор родительского класса. Если у конструктора родительского класса нет параметров, то он будет вызван автоматически. Если параметры есть, то конструктор надо вызывать явно. Например, реализация конструктора класса `CBall`, унаследованного от класса `CGraphicalObject`, может выглядеть следующим образом (если в качестве параметров конструктору базового класса надо передать экранные координаты центра объекта):

```
CBall::CBall( int x, int y, int c ) :
       CGraphicalObject( x, y )
{
  // Выполняем инициализацию дочернего класса
  ...
}
```

У класса обычно бывает два вида клиентов:

- объекты (экземпляры классов);
- подклассы.

Часто полезно иметь для них разные интерфейсы. В частности, мы хотим показать только внешне видимое поведение для клиентов-объектов, но нам нужно открыть служебные функции и представления клиентам-подклассам. Этим объясняется наличие открытой, защищенной и закрытой частей описания класса в языке Си++: разработчик может четко разделить, какие элементы класса доступны для объектов, а какие для подклассов.

Есть серьезные противоречия между потребностями наследования и инкапсуляции. В значительной мере наследование открывает наследующему классу некоторые "секреты". На практике, чтобы понять, как работает какой-то класс, часто надо изучить все его родительские классы в их внутренних деталях.

Наследование подразумевает, что подклассы повторяют структуры их родительских классов. В предыдущем примере объекты класса `CElectricalData` содержат элементы структуры родительского класса (`id` и `timeStamp`) и более специализированные элементы (переменные `fuelCell1Voltage`, `fuelCell2Voltage`, `fuelCell1Amperes`, `fuelCell2Amperes`).

Поведение родительских классов также наследуется. Применительно к объектам класса `CElectricalData` можно использовать операции `CurrentTime()` (унаследована от родительского класса), `CurrentPower()` (определена в классе) и `Transmit()` (переопределена в подклассе).

В Си++ и в большинстве языков допускается не только наследование методов родительского класса, но также добавление новых и переопределение существующих методов. Функция, объявленная виртуальной (функция `Transmit()` в предыдущем примере), может быть в подклассе переопределена, а остальные (`CurrentTime()`) – нет.

3. Одиночный полиморфизм

Пусть функция для передачи телеметрических данных (например, по радиоканалу) `Transmit()` класса `CTelemetryData` реализована следующим образом:

```
void CTelemetryData::Transmit()
{
  // передать id
  // передать timeStamp
}
```

В классе `CElectricalData` эта виртуальная функция переопределяется для передачи данных, характеризующих именно солнечные батареи:

```
void CElectricalData::Transmit()
{
  CTelemetryData::Transmit();
  // передать значение напряжения
  // передать значение силы тока
}
```

Эта функция сначала вызывает одноименную функцию родительского класса с помощью ее явно указанного имени `CTelemetryData::Transmit()`. Та передаст заголовок пакета (`id` и `timeStamp`), после чего в подклассе передаются его собственные данные.

Определим теперь экземпляры двух описанных выше классов:

```
CTelemetryData telemetry;
CElectricalData electrical(5.0, -5.0, 3.0, 7.0);
```

Допустим, в программе есть глобальная функция:

```
void transmitFreshData( CTelemetryData& d, const Time& t)
{
  if ( d.CurrentTime() >= t )
    d.Transmit();
)
```

Что произойдет, если выполнить следующие два оператора?

```
transmitFreshData( telemetry, Time(60) );
transmitFreshData( electrical, Time(120) );
```

В первом операторе будет передан уже известный нам заголовок. Во втором будет передан он же, плюс четыре числа в формате с плавающей точкой, содержащие результаты измерений электрических параметров. Почему это так? Ведь функция `transmitFreshData()` ничего не знает о классе объекта, она просто выполняет `d.Transmit()`! Это был пример полиморфизма. Переменная `d` может обозначать объекты разных классов. У этих классов есть общий родительский класс и они, хотя и по разному, могут реагировать на одно и то же сообщение, одинаково понимая его смысл.

Традиционные типизированные языки (напр., Паскаль) основаны на той идее, что функции и процедуры, а следовательно, и операнды, должны иметь определенный тип. Это свойство называется *мономорфизмом*, то есть каждая переменная и каждое значение относятся к одному определенному типу. В противоположность мономорфизму *полиморфизм* допускает отнесение значений и переменных к нескольким ти-

пам. Впервые идея полиморфизма была описана применительно к возможности переопределять смысл символов, таких, как "+", сообразно потребности. В современном программировании это называется *перегрузкой*. Например, в Си++ можно определить несколько функций с одним и тем же именем, и они будут автоматически различаться по количеству и типам своих аргументов. Т.о., в Си++ есть два проявления полиморфизма:

1) перегрузка операторов и функций;
2) использование виртуальных функций.

При отсутствии полиморфизма код программы вынужденно содержит операторы множественного выбора `switch`. Допустим, на языке Си для хранения телеметрических данных используются не классы, а структуры, у которых первым полем является переменная `kind`, обозначающая конкретный тип данных. Для выбора варианта нужно проверить значения этой переменной. Поэтому функцию передачи данных можно написать следующим образом:

```
enum TelemetryType { Electrical, Propulsion, Spectrometer };

void transmitFreshData( TelemetryData* pData, Time t )
{
  if ( pData->currentTime >= t )
    switch ( pData->kind )
      {
      case Electrical :
        transmitElectricalData( (ElectricalData*)pData );
        break;
      case Propulsion :
        transmitPropulsionData( (PropulsionData*)pData );
        break;
      }
}
```

Чтобы ввести новую разновидность телеметрических данных, нужно добавить еще одну константу в перечисление `TelemetryType`, создать новое описание структуры (в котором поле `kind` обязательно должно быть первым) и изменить каждый оператор `switch`. В такой ситуации вероятность ошибок сильно увеличивается.

Наследование позволяет различать разновидности абстракций. Полиморфизм наиболее целесообразен в тех случаях, когда несколько классов имеют одинаковые протоколы. Полиморфизм позволяет обойтись без операторов выбора, поскольку объекты "сами знают свой тип".

4. Наследование и типизация

4.1 Вызов виртуальных функций-членов родительского класса

Рассмотрим еще раз переопределение функции `Transmit()`:

```
void CElectricalData::Transmit()
{
  CTelemetryData::Transmit();
  // передать значение напряжения
  // передать значение силы тока
}
```

В Си++, как и в большинстве ОО-языков программирования, при реализации метода подкласса разрешается вызывать напрямую метод какого-либо родительского класса. Как видно из примера, это допускается и в том случае, если функция-член является виртуальной (т.е. ее имя совпадает с именем функции-члена родительского класса и фактически переопределяет ее). В Си++ функцию-член любого достижимого вышестоящего класса можно вызывать, добавляя имя класса в качестве префикса, формируя полное имя функции-члена (например, `CTelemetryData::Transmit()`).

На практике функция-член родительского класса обычно вызывается до или после дополнительных действий. Функция-член подкласса уточняет или дополняет поведение родительского класса.

4.2 Присваивание объектов

Все подклассы на рис. 6.1 являются также *подтипами* вышестоящего класса. В частности, `CElectricalData` является подтипом `CTelemetryData`. Рассмотрим следующие объявления двух объектов:

```
CTelemetryData telemetry;
CElectricalData electrical( 5.0, -5.0, 3.0, 7.0 );
```

Следующий оператор присваивания правомочен:

```
telemetry = electrical; //electrical - это подтип telemetry
```

Хотя он формально правилен, он опасен: любые дополнения в состоянии подкласса по сравнению с состоянием родительского класса просто срезаются. Таким образом, дополнительные четыре параметра, имеющиеся в объекте `electrical`, будут потеряны при копировании, поскольку их просто некуда записать в объект `telemetry` класса `CTelemetryData`.

Следующий оператор неправилен и вызовет ошибку компиляции, т.к. `CTelemetryData` не является подтипом `CElectricalData`:

```
electrical = telemetry;
```

Можно сделать заключение, что присвоение объекту `y` значения объекта `x` допустимо, только если тип объекта `x` совпадает с типом объекта `y` или является его подтипом.

4.3 Преобразование типов при наследовании

Как отмечено в п.4.2, нельзя присваивать значение типа родительского класса переменной, объявленной как объект подкласса. Тем не менее в некоторых редких случаях желательно нарушить это правило. Чаще всего такая ситуация возникает, когда программист дополнительно знает, что значение, хотя и содержится в переменной типа родительского класса, на самом деле является объектом дочернего класса. Тогда можно (хотя это и не приветствуется) "перехитрить" систему проверки типов данных.

В Си++ это можно сделать с помощью конструкции языка, называемой приведением типа данных. Приведение типа заставляет компилятор преобразовать значение из одного типа данных в другой. Наиболее часто этот подход используется в случае указателей, когда осуществляется только логическая замена, а не физическое преобразование.

Проиллюстрируем приведение типа в варианте игры в бильярд, в которой бильярдные шары хранятся в связном списке. Вместо того, чтобы заставлять каждый

экземпляр шара CBall содержать указатель, используемый в связном списке, определим обобщенный класс CLink ("Элемент связного списка") следующим образом:

```
class CLink {
  protected:
    CLink* pLink;
  public:
    CLink* GetNext();
    void SetLink(CLink* pElem);
};
void CLink::SetLink(CLink* pElem)
{
  pLink = pElem;
}
CLink* CLink::GetNext()
{
  return pLink;
}
```

Класс CBall сделаем подклассом класса CLink. Поскольку функция-член SetLink() наследуется от родительского класса, его нет необходимости повторять. Однако есть проблема с наследуемой функцией GetNext(). Он утверждает, что возвращает экземпляр класса CLink, а не CBall. Тем не менее известно, что объект на самом деле принадлежит классу CBall, поскольку это единственные объекты, которые мы помещаем в список. Поэтому можно переписать класс CBall так, чтобы переопределить функцию GetNext(). В реализации этой функции применим приведение типа для того, чтобы изменить тип возвращаемого значения:

```
class CBall : public CLink {
  ...
  public:
    ...
    CBall* GetNext();
    ...
};
CBall* CBall::GetNext()
{
  return dynamic_cast<CBall*>(CLink::GetNext());
}
```

Заметьте, что функция-член GetNext() не описана как виртуальная, т.к. нельзя изменить тип возвращаемого значения виртуальной функции. Важно помнить, что в языке Си++ это приведение типа будет законным только для указателей, а не для собственно объектов. Оператор dynamic_cast() является частью встроенной в Си++ системы **RTTI** (Run-Time Typing Information – идентификация типа во время выполнения программы). Приведение типов через RTTI должно использоваться вместо более ранней синтаксической формы (CBall*), поскольку неправильное приведение типов данных является стандартным источником ошибок в программах.

5. Упражнения

Упражнение 5.1

Имеется описание класса:

```
class CBase {
  public :
```

```
    virtual void Iam()     { cout << "CBase\n"; }
};
```

Постройте два производных от `CBase` класса и для каждого определите функцию `Iam()` ("Я есть"), которая выводит имя класса в стандартный поток вывода. Создайте объекты этих классов и вызовите для них `Iam()`.

Присвойте адреса объектов производных классов указателям `CBase*` и вызовите `Iam()` через эти указатели.

Упражнение 5.2

На рис. 6.2 показана иерархия классов, представляющих графические фигуры разной формы. От базового класса `CShape` унаследованы классы "Сфера", "Треугольная пирамида" и "Параллелепипед". У класса " Параллелепипед" (`CBox`) есть подкласс "Каркасный параллелепипед" (`CWireBox`).

Изучите приведенное ниже описание и разработайте программу, в которой эти классы используются для рисования сцены из нескольких параллелепипедов, сфер и пирамид. Для рисования фигур используйте функции из библиотеки **GLAUX**.

В базовом классе `CShape` хранятся координаты центра фигуры. У базового класса есть следующие функции-члены:

- `Draw` – нарисовать фигуру;
- `Move` – передвинуть фигуру;
- `GetCenter` – вернуть координаты центра фигуры.

Функции-члены `GetCenter` и `Move` являются общими для всех подклассов и не требуют обязательного переопределения. Однако, поскольку только подклассы могут знать, как их изображать, то функция `Draw()` должна быть виртуальной и переопределяться в подклассах.

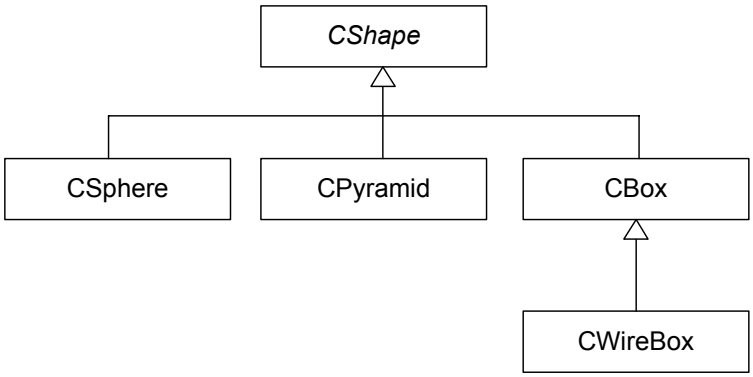

Рис. 6.2. Иерархия классов "графические фигуры".

Класс `CSphere` имеет переменную `radius` и соответствующие функции-члены для задания (`SetRadius`) и чтения значения этой переменной. Функция-член `Draw` у этого класса формирует изображение сферы заданного радиуса с центром в точке `center`. В классе `CBox` есть переменные `height` (высота), `width` (ширина) и `depth` (высота) и соответствующие функции для задания и чтения их значений. Функция `Draw` в данном классе формирует изображение параллелепипеда заданных размеров с центром в заданной точке `center`. Подкласс `CWireBox` наследует все особенности класса `CBox`, но операция `Draw` в этом подклассе переопределена.

Рассмотрим следующий фрагмент программы:

```
CShape* shapes[10];
// Здесь объекты создаются и инициализируются
...
for ( int i = 0; i < 10; i++ )
  shapes[i]->Draw();
```

В приведенном фрагменте программы есть разнородный массив объектов, содержащий указатели на любые разновидности подклассов `CShape`. Подобный способ рисования можно применить для рисования сцены, состоящей из нескольких графических объектов.

Упражнение 5.3

Примените наследование в программе для имитации бильярда, которая описана в 5-й лекции. Классы "Шар", "Луза" и "Стенка" унаследуйте от базового класса "Графический объект". Этот класс определяется следующим образом:

```
class CGraphicalObject {
  public:
    CGraphicalObject( int left, int top, int right, int bottom );
    // Операции, выполняемые графическими объектами
    virtual void Draw()      {;}
    virtual void Update()    {;}
    virtual void HitBy( CGraphicalObject* pObj )   {;}

    // Функции-члены для доступа к переменным класса
    CRect GetRegion()               { return region; }
    CGraphicalObject* GetLink()     { return pLink; }

  protected :
    CRect region;
    CGraphicalObject* pLink;
};
```

В конструкторе просто запоминаются координаты экранной области, в которую вписано изображение объекта. Функции-члены `Draw()`, `Update()` и `HitBy()` ничего не делают, так как их фактическое поведение определено в подклассах.

```
CGraphicalObject::CGraphicalObject( int left, int top,
                                    int right, int bottom )
{
  region.SetRect( left, top, right, bottom );
}
```

Теперь классы `CBall`, `CWall` и `CHole` объявляются как подклассы класса `CGraphicalObject`, и внутри них ни к чему объявлять данные или функции, если только они не переопределяются:

```
class CHole : public CGraphicalObject {
  public:
    CHole( int x, int y, CGraphicalObject* pNextObj );
    // Рисование лузы
    virtual void Draw();
    // Извещение лузы о том, что в нее попал шар
    virtual void HitBy( CGraphicalObject* pObj );
};
```

Функция-член `HitBy` должна преобразовать тип параметра в `CBall*`. Для этого применяется оператор приведения типа `dynamic_cast`. (Чтобы можно было пользоваться этим оператором в **MS Visual C++**, надо разрешить работу средств RTTI командой ***Project⇒Settings⇒C/C++⇒C++ Language⇒Enable Run-Time Type Information (RTTI)***)

```
void CHole::HitBy( CGraphicalObject* pObj )
{
  CBall* pBall = dynamic_cast<CBall*>( pObj );

  if ( pBall )
    {
    pBall->SetEnergy( 0.0 );
    if ( pBall->IsCue() )
      pBall->SetCenter( 50, 108 );
    else
      {
      pBall->SetCenter( 10 + saveRack*15, 250 );
      saveRack++;            // Увеличение счетчика забитых шаров
      }
    }
}
```

Наибольшее упрощение в программе достигается за счет того, что теперь можно держать все графические объекты в одном списке. Функция, рисующая всю сцену, записывается так:

```
void CALLBACK Display()
{
  glClear( GL_COLOR_BUFFER_BIT );
  CGraphicalObject* pObj = listOfObjects;
  while ( pObj )
    {
    pObj->Draw();
    pObj = pObj->GetLink();
    }
  auxSwapBuffers();
}
```

Наиболее важным местом этой функции является вызов функции-члена `Draw()` внутри цикла. Несмотря на то что вызов написан один, иногда будет вызываться функция класса `CBall`, а в других случаях – класса `CWall` или `CHole`.

Часть функции `CBall::Update()`, проверяющая, ударился ли движущийся шар обо что-нибудь, также упрощается аналогичным образом.

Лекция 7. Отношения между классами

1. Типы отношений между классами

Рассмотрим сходства и различия между следующими классами: цветы, маргаритки, красные розы, желтые розы, лепестки и божьи коровки. Мы можем заметить следующее:

- Маргаритка – цветок.
- Роза – (другой) цветок.
- Красная и желтая розы – розы.
- Лепесток является частью обоих видов цветов.
- Божьи коровки питаются вредителями, поражающими некоторые цветы.

Из этого простого примера следует, что классы, как и объекты, не существуют изолированно. В каждой предметной области ключевые абстракции взаимодействуют различными способами, которые надо отразить при разработке программы.

Отношения между классами могут означать одно из двух. Во-первых, у них может быть что-то общее. Например, и маргаритки, и розы – это разновидности цветов: и те, и другие имеют ярко окрашенные лепестки, испускают аромат и так далее. Во-вторых, может быть какая-то смысловая связь. Например, красные розы больше похожи на желтые розы, чем на маргаритки. Но между розами и маргаритками больше общего, чем между цветами и лепестками. Также существует симбиотическая связь между цветами и божьими коровками: божьи коровки защищают цветы от вредителей, которые, в свою очередь, служат пищей божьим коровкам.

Известны три основных типа отношений между классами. Во-первых, это отношение *"обобщение/специализация" (общее и частное)*, известное как "is-a". Розы суть цветы, что значит: розы являются специализированным частным случаем, подклассом более общего класса "цветы". Во вторых, это отношение *"целое/часть"*, известное как "part of". Так, лепестки являются частью цветов. В-третьих, это *смысловые отношения* (ассоциации). Например, божьи коровки ассоциируются с цветами – хотя, казалось бы, что у них общего. Или вот: розы и свечи – и то, и другое можно использовать для украшения стола.

Языки программирования выработали несколько общих подходов к выражению отношений этих трех типов. В частности, большинство ОО- языков непосредственно поддерживают разные комбинации следующих видов отношений:

- ассоциация;
- наследование;
- агрегация;
- использование;
- параметризация.

Из перечисленных видов отношений наиболее общим и неопределенным является *ассоциация*. Обычно при разработке программы проектировщик констатирует наличие ассоциации и, постепенно уточняя проект, превращает ее в какую-то более специализированную связь.

Наследование выражает отношение общего и частного. Однако одного наследования недостаточно, чтобы выразить все многообразие явлений и отношений жизни. Полезна также *агрегация* (*композиция*), отражающая отношения целого и части между экземплярами классов.

Отношение *использования* означает наличие связи между объектами классов. В языке Си++ (и в некоторых других) наряду с наследованием есть специфическая разновидность обобщения, которая называется *параметризацией*.

2. Выбор между агрегацией и наследованием

Среди видов отношений между классами наиболее часто встречаются *наследование* и *агрегация*.

Наследование применяется в тех случаях, когда один класс является уточненной, более специализированной формой другого. Чтобы определить, является ли понятие *X* уточненным вариантом понятия *Y*, можно составить предложение "*X* является экземпляром *Y*". Если утверждение звучит корректно (т.е. есть соответствует вашим представлениям о решаемой задаче), то вы можете заключить, что *X* и *Y* связаны отношением наследования.

Отношение агрегации ("*включать как часть*") имеет место, когда второе понятие является составной частью первого, но оба эти понятия не совпадают ни в каком смысле независимо от уровня общности абстракции. Например, автомобиль `Car` *имеет* двигатель `Engine`, хотя ясно, что это не тот случай, когда `Car` *является экземпляром* `Engine` или `Engine` *является экземпляром* `Car`. Класс `CCar` тем не менее *является экземпляром* класса автомобилей `CVehicle`, который в свою очередь *является экземпляром* класса транспортных средств `CMeansOfTransportation`. Аналогично проверке для наследования, чтобы проверить отношение агрегации, можно составить предложение "*X* включает *Y* как часть" и оценить его корректность.

В большинстве случаев различие между агрегацией и наследованием очевидно. Но иногда оно может быть сомнительно или зависеть от обстоятельств. Далее, в п. 3, рассмотрим решение одной и той же задачи двумя способами, с использованием разных отношений между классами.

3. Демонстрация агрегации и наследования

Чтобы проиллюстрировать агрегацию и наследование, рассмотрим построение абстрактного типа данных "Множество" (класс `CSet`) на основе класса "Связный список" (`CList`). Объекты класса `CList` содержат списки целочисленных величин. Допустим, что имеется класс `CList` со следующим интерфейсом:

```
class CList {
  public:
    CList();

    void AddToFront(int);
    int  FirstElement();
    int  Length();
    bool IsIncludes(int);
    int  Remove(int);
    ...
};
```

Класс "Список" позволяет добавлять новый узел в начало списка, получать значение первого узла, вычислять количество узлов, проверять, содержится ли значение в списке, и удалять узел с заданным значением из списка.

Предположим, что необходимо создать класс "Множество", который позволяет выполнять такие операции, как добавление элемента в множество, определение количества элементов, проверка принадлежности к множеству.

3.1. Использование агрегации

Сначала рассмотрим, как построить класс "Множество" с помощью агрегации. Известно, что класс инкапсулирует внутри себя состояние и поведение. Когда для создания нового класса используется агрегация, то существующий класс включается в новый класс в виде переменной-члена. Ниже показано, что внутри класса `CSet` заведена переменная-член `data` –объект класса `CList`.

```
class CSet {
  public:
    CSet();

    void Add(int);
    int  Size();
    bool IsIncludes(int)

  private:
    CList data;
};
```

Поскольку объект класса `CList` является переменной-членом класса "Множество", то этот объект надо инициализировать в конструкторе класса `CSet`. Если у конструктора объекта нет параметров, то он вызывается автоматически (как в данном случае), иначе его приходится вызывать явно.

Функции-члены класса `CSet` реализованы с использованием функций-членов класса `CList`. Например, функции-члены `IsIncludes` и `Size` для множества просто вызывают соответствующие функции-члены у списка:

```
int CSet::Size()
{
  return data.Length();
}

int CSet::IsIncludes( int newValue )
{
  return data.IsIncludes( newValue );
}
```

Функция-член `Add()` для добавления нового элемента в множество оказывается более сложной, т.к. сначала нужно убедиться, что в множестве данный элемент отсутствует (в множестве м.б. единственный элемент с заданным значением):

```
void CSet::Add( int newValue )
{
  if ( !IsIncludes( newValue ) )
    data.AddToFront( newValue );
}
```

Приведенный пример показывает, как агрегация помогает повторному использованию компонент в программах. Большая часть работы, связанной с хранением данных, в классе `CSet` проделывается существовавшим ранее классом `CList`. Графическое изображение отношения агрегации показано на рис. 7.1.

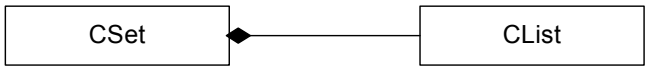

Рис. 7.1. Агрегация.

Следует понимать, что при создании нового класса с помощью агрегации классы `CList` и `CSet` будут абсолютно различны, и ни один из них не является уточнением другого.

3.2. Использование наследования

При использовании наследования новый класс может быть объявлен как подкласс существующего класса. В этом случае все области данных и функции, связанные с исходным классом, автоматически переносятся в подкласс. Подкласс может определять дополнительные переменные и функции. Он переопределяет некоторые функции исходного класса, которые были объявлены в нем как виртуальные.

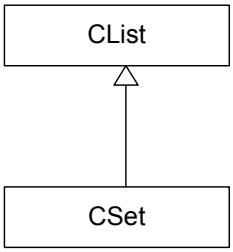

Рис. 7.2. Наследование.

Наследование подробно рассматривалось в 5-й лекции. Графическое изображение наследования приведено на рис. 7.2. Ниже показано, как можно применить наследование для создания класса `CSet` в форме подкласса класса `CList`. Подкласс является расширением существующего класса `CList`, поэтому все функции-члены списка оказываются применимы и к множеству.

```
class CSet : public CList {
  public:
    CSet();
    void Add(int);
    int  Size();
};
```

Обратите внимание, что в подклассе не определено никаких новых переменных. Вместо этого переменные класса `CList` будут использоваться для хранения элементов множества.

Аналогично функции-члены родительского класса можно использовать в подклассе, поэтому не надо объявлять функцию-член `IsIncludes` – в классе `CList` есть функция-член с таким же именем и подходящим поведением. Функция-член для добавления в множество нового элемента выполняет вызовы двух функций-членов класса `CList`:

```
void CSet::Add( int newValue )
{
  if ( !IsIncludes( newValue ) )
    AddToFront( newValue );
}
```

Сравните этот вариант функции `Add` с вариантом из п. 3.1. Оба вида отношений – агрегация и наследование – являются мощными механизмами для многократного использования кода. В отличие от агрегации, наследование содержит неявное предположение, что подклассы на самом деле являются подтипами. Это значит, что объекты подкласса должны вести себя так же, как и объекты родительского класса.

3.3. Сравнение агрегации и наследования

В п. 3.1-3.2 показано, что оба вида отношений между классами – агрегацию и наследование – можно применить для реализации класса "Множество". На рассмотренном примере перечислим некоторые недостатки и преимущества двух подходов.

- Агрегация более проста. Ее преимущество заключается в том, что она ясно показывает, какие точно функции-члены будут содержаться в классе. Из описания класса `CSet` становится очевидно, что для объектов-множеств предусмотрены только операции добавления элемента, проверки на наличие элемента и определение количества элементов в множестве. Это справедливо независимо от того, какие функции-члены определены в классе `CList`.
- При наследовании функции-члены нового класса дополняют и, возможно, переопределяют набор функций-членов родительского класса. Таким образом, чтобы точно знать, какие функции-члены есть у подкласса, программист должен изучить описание родительского класса. Например, из описания класса `CSet` не видно, что у множеств можно выполнять проверку на наличие элемента в множестве (функция-член `IsIncludes`). Это можно понять только после изучения описания родительского класса `CList`. Т.о., у наследования есть неприятная особенность: чтобы полностью понять поведение и свойства подкласса, программист должен изучать описание одного или нескольких родительских классов.
- С другой стороны, указанную выше особенность наследования можно считать преимуществом: описание класса получается более кратким, чем в случае агрегации. Применяя наследование, оказывается ненужным писать все функции для доступа к переменным-членам родительского класса. Наследование также часто обеспечивает большую функциональность. Например, применение наследования в нашем случае делает доступным для множеств не только проверку `IsIncludes`, но и функцию `Remove`.
- Наследование не запрещает пользователям манипулировать новыми классами через вызовы функций-членов родительского класса, даже если эти функции не вполне подходят под идеологию потомка. Например, при использовании наследования для класса `CSet` пользователи смогут добавлять элементы в множество с помощью унаследованной от класса `CList` функции `AddToFront`.
- При агрегации тот факт, что класс `CList` используется для хранения элементов множеств, – просто скрытая деталь реализации. Т.о., можно легко переписать класс `CSet` более эффективным способом (например, на основе массива) с минимальным воздействием на пользователей класса `CSet`. Если же пользователи рассчитывают на то, что класс `CSet` – это более специализированный вариант класса `CList`, то такие изменения было бы трудно реализовать.

Как в конкретном случае выбрать один из двух механизмов реализации? Надо воспользоваться правилом подстановки "*X* является экземпляром *Y*". Корректно ли звучит утверждение "Множество является экземпляром списка"? Т.е. можно ли в программе, где используется класс `CList`, подставлять вместо него класс `CSet`? Исходя из смысла понятий "множество" и "список", можно сказать, что нельзя. Поэтому в *данном случае* агрегация подходит лучше.

4. Отношение ассоциации

Желая автоматизировать розничную торговую точку, мы обнаруживаем две абстракции – товары и продажи. На рис. 7.3 показана ассоциация, которую мы усматриваем между соответствующими классами. Класс `CProduct` – это то, что было продано в некоторой сделке, а класс `CSale` – сама сделка, в которой продано несколько товаров. Эта ассоциация работает в обе стороны: задавшись товаром, можно выйти на сделку, в которой он был продан, а пойдя от сделки, найти, что было продано.

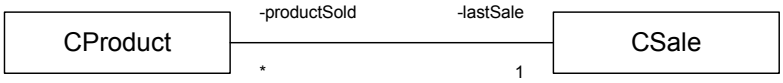

Рис. 7.3. Ассоциация.

В Си++ подобную связь можно выразить при помощи "погребенных указателей", например, следующим образом:

```
class CProduct;
class CSale;
class CProduct {
  public:
    ...
  protected:
    CSale* lastSale;
};
class CSale {
  public:
    ...
  protected:
    CProduct** productSold;
};
```

Это ассоциация вида "один-ко-многим": каждый экземпляр товара относится только к одной последней продаже, в то время как каждый объект `CSale` может указывать на совокупность проданных товаров.

Как показывает этот пример, ассоциация – смысловая (семантическая) связь. По умолчанию, она не имеет направления (если не оговорено противное, ассоциация, как в данном примере, подразумевает двухстороннюю связь) и не объясняет, как классы общаются друг с другом (мы можем только отметить семантическую зависимость, указав, какие роли классы играют друг для друга). Однако именно это нам требуется на ранней стадии анализа. Итак, мы фиксируем участников, их роли и *мощность отношения*.

В предыдущем примере мы имели ассоциацию "один ко многим". Тем самым мы обозначили ее *мощность* (т.е., грубо говоря, количество участников). На практике важно различать три случая мощности ассоциации:

- "один-к-одному"

- "один-ко-многим"
- "многие-ко-многим".

Отношение "один-к-одному" обозначает очень узкую ассоциацию. Например, в розничной системе продаж примером могла бы быть связь между классом `CSale` и классом `CCreditCardTransaction`: каждая продажа соответствует ровно одному снятию денег с данной кредитной карточки.

Отношения "многие-ко-многим" тоже нередки. Например, каждый объект класса `CCustomer` (покупатель) может инициировать транзакцию с несколькими объектами класса `CSalePerson` (торговый агент), и каждый торговый агент может взаимодействовать с несколькими покупателями.

5. Отношение использования

Во многих промышленных процессах требуется непрерывное изменение температуры. Необходимо поднять температуру до заданного значения, выдержать заданное время и понизить до нормы. Профиль изменения температуры у разных процессов разный; зеркало телескопа надо охлаждать очень медленно, а закаляемую сталь очень быстро.

Абстракция нагрева имеет достаточно четкое поведение, что дает нам право на описание такого класса – `CTemperatureRamp`, который по смыслу задает функцию времени от температуры:

```
class CTemperatureRamp {
  public:
    CTemperatureRamp();
    virtual ~CTemperatureRamp();
    virtual void Clear();
    virtual void Bind( int temperature, int minute );
    int TemperatureAt( int minute );
  protected:
    ...
};
```

Управления нагревателем, поддерживающего требуемый профиль, мы от этого класса не требуем. Мы предпочитаем разделение понятий, при котором нужное поведение достигается взаимодействием трех объектов: объекта `CTemperatureRamp`, нагревателя `CHeater` и контроллера `CTemperatureController`. Класс `CTemperatureController` можно определить так:

```
class CTemperatureController {
  public:
    CTemperatureController( int location );
    ~CTemperatureController();
    void Process( const CTemperatureRamp& );
    int Schedule( const CTemperatureRamp& );
  private:
    CHeater h;
};
```

Класс `CTemperatureController` и `CHeater` связаны агрегацией. Класс `CTemperatureRamp` упомянут в заголовке двух функций-членов класса

CTemperatureController. Поэтому можно сказать, что класс CTemperatureController пользуется услугами класса CTemperatureRamp.

Отношение использования между классами соответствует равноправной связи между их объектами. Это то, во что превращается ассоциация, если оказывается, что одна из ее сторон (клиент) пользуется услугами другой (сервера).

На самом деле, один класс может использовать другой по-разному. В нашем примере это происходит в заголовке функции-члена. Можно представить, что CTemperatureController внутри реализации функции Schedule использует, например, объект класса CPredictor (предсказатель значения температуры). Отношения целого и части тут ни при чем, поскольку этот объект не входит в объект CTemperatureController, а только используется. В типичном случае такое отношение использования проявляет себя, если в реализации какой-либо функции-члена происходит объявление локального объекта используемого класса.

6. Отношение параметризации

Рассмотрим описание простого класса "Стек " CStack.

```
class CStack {
  public:
    CStack();
    void Push( int val );
    int  Pop();
    bool IsFull();
  protected:
    int data[100];
    int top;
};
```

У этого класса есть очевидный недостаток: он годится только для работы с целочисленными стеками. Чтобы стек можно было применять для хранения значений других типов, можно усовершенствовать описание класса с помощью конструкции параметризованных классов.

```
template<class T>
class CStack {
  public:
    CStack();
    void Push( T val );
    T    Pop();
    bool IsFull();
  protected:
    T   data[100];
    int top;
};
```

В этом новом варианте класса он описывается с помощью шаблона, в котором тип с формальным именем T является параметром шаблона. При создании объектов параметризованного класса требуется указывать тип-параметр шаблона в угловых скобках, например:

```
CStack<double> doubleStack;
CStack<CGraphicalObject*> graphStack;
```

Объекты doubleStack (стек вещественных чисел) и graphStack (стек указателей на графические объекты) – это объекты совершенно разных классов, которые даже не имеют общего родительского класса. Тем не менее, они получены из одного параметризованного класса CStack.

Синтаксис реализации функций-членов параметризованного класса дополняется указанием параметра шаблона, например:

```
template <class T>
CStack<T>::CStack()
{
  top = 0;
}

template <class T>
void CStack<T>::Push( T val )
{
  data[top++] = val;
}
```

Параметризация безопасна с точки зрения типов. Например, при работе с объектами doubleStack и graphStack по правилам Си++ будет отвергнута любая попытка поместить в стек или извлечь из него что-либо кроме, соответственно, вещественных чисел или указателей на подклассы CGraphicalObject.

Отношения между параметризованным классом Stack и его инстанцированием для класса CGraphicalObject показаны на рис. 7.4.

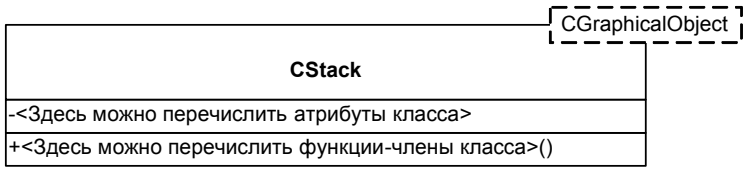

Рис. 7.4. Отношение параметризации.

Как видно из рис. 7.4, чтобы инстанцировать параметризованный класс CStack, необходимо использовать другой класс, например, CGraphicalObject. Благодаря этому отношение параметризации подразумевает отношение использования.

При проектировании параметризованные классы позволяют выразить некоторые свойства протоколов классов. Класс экспортирует операции, которые можно выполнять над его объектами. Наоборот, параметризующий параметр класса служит для импорта классов и значений, предоставляющих некоторый протокол. Си++ проверяет их взаимное соответствие при компиляции, когда фактически и происходит инстанцирование.

7. Упражнения

1) На основе перечисленных ниже понятий сформируйте описания классов, связанных друг с другом отношениями различных видов. Удостоверьтесь, что описания классов успешно проходят этап компиляции и можно создавать объекты этих классов (конструкторы и функции-члены классов можете не уточнять и просто сделать пустыми).

- Классы "Двигатель", "Паровой двигатель", "Дизель" и "Двигатель внутреннего сгорания" связаны отношениями наследования.
- Классы "Автомобиль", "Двигатель" и "Колесо" связаны отношением агрегации (предусмотрите, что у автомобиля должно быть 4 колеса).
- Классы "Автомобиль", "Фирма-производитель автомобилей" и "Владелец автомобиля" связаны отношением ассоциации.
- Классы "Автомобиль" и "Маршрутный лист " связаны отношением использования (у класса "Автомобиль" должна быть некоторая функция-член "Поездка по маршруту").

2) Предположим, что на основе класса "Связный список" `CList` мы хотим построить класс "Упорядоченный список" `COrderedList`, в котором узлы вставляются в определенном порядке, а не просто в начало списка (порядок определяется операцией сравнения, применяемой к значениям узлов, например, для числовых значений возможен порядок "по возрастанию" или "по убыванию"). Наследование или агрегацию вы будете использовать в данном случае? Обоснуйте ваш ответ (описание классов на Си++ записывать не обязательно).

3) Полутоновое цифровое изображение – это двумерный массив значений яркости. Опишите класс "Изображение" `CImage`, предназначенный для хранения подобных изображений.
   У класса должно быть два конструктора: без параметров и с параметрами высота/ширина изображения. Во втором конструкторе должна динамически распределяться память для хранения данных изображения из расчета 1 байт на пиксел. В деструкторе эта память должна освобождаться.
   Функции-члены класса должны позволять: получать и задавать значения яркости отдельного пиксела изображения по его координатам (строка/столбец изображения), рисовать изображение на экране и сохранять/загружать изображения из файла. Функции рисования и работы с файлами сделайте пустыми, только обеспечьте у них осмысленные заголовки.
   Убедитесь, что описание класса `CImage` и реализация его функций-членов записаны корректно. Для этого попробуйте создать несколько объектов этого класса. Затем преобразуйте описание класса `CImage` в параметризованный класс, так, чтобы для хранения пиксела можно было использовать не только байты, но и целые или вещественные числа.

4) Абстрактный тип данных "стек" можно реализовать по-разному – например, на основе массива (как это было показано ранее) или на основе связного списка. Разработайте описания классов `CList` и `CStack` так, чтобы показать, как можно реализовать стек целых чисел на основе связного списка с помощью наследования и агрегации. Можете ввести какие угодно методы для обоих классов. Какая техника реализации кажется вам более подходящей в данном случае?

Лекция 8. Обработка исключительных ситуаций. Средства динамической идентификации типа

1. Обработка исключительных ситуаций

В Си++ есть встроенный механизм обработки ошибок, который называется *обработкой исключительных ситуаций* (exception handling). Этот механизм упрощает реализацию реакции на ошибки, которые происходят во время выполнения программ. Обработка исключительных ситуаций (исключений) организуется с помощью трех операторов: `try`, `catch` и `throw`.

Операторы программы, во время выполнения которых надо обеспечить обработку исключений, располагаются внутри блока `try`. Если в процессе выполнения блока `try` обнаруживается ошибка, то с помощью оператора `throw` возбуждается исключение. Оно перехватывается и обрабатывается операторами, которые находятся в блоке `catch`. Этот блок располагается непосредственно за блоком `try`, возбуждающим исключительную ситуацию. Правила расположения операторов `try` и `catch` можно представить следующим образом:

```
try (
      // блок возбуждения исключительной ситуации
}
catch (type1 arg) {
      // блок перехвата исключительной ситуации типа type1
}
catch (type2 arg) {
      // блок перехвата исключительной ситуации типа type2
}
  ...
catch (typeN arg) {
      // блок перехвата исключительной ситуации типа typeN
}
```

Блок `try` должен содержать ту часть программы, в который требуется отслеживать ошибки. Это могут быть как несколько операторов внутри одной функции, так и все операторы внутри функции `main()` (что ведет к отслеживанию ошибок во всей программе).

Возбужденное исключение перехватывается оператором `catch`, который соответствует этому конкретному типу исключения. С блоком `try` может быть связано несколько блоков `catch`. После возбуждения исключения все оставшиеся операторы блока `try` игнорируются (т.е. сразу после того, как какой-то оператор внутри блока `try` сгенерировал исключение, управление передается соответствующему оператору `catch`**,** минуя оставшиеся операторы блока `try`**).** При перехвате исключения его значение внутри блока `catch` доступно под именем `arg`. Если доступ к информации об исключении не нужен, то в операторе `catch` можно указать только ее тип `type`**,** а имя параметра `arg` не указывать.

Возбуждение исключительной ситуации выполняется оператором `throw`:

```
throw значение;
```

Оператор `throw` должен выполняться либо внутри блока `try`**,** либо в любой функции, которая вызывается из этого блока (прямо или косвенно). Значение после служебного слова `throw` – это информация о возникшем исключении. Это значение может иметь любой тип, в том числе быть динамически созданным объектом класса.

Если в программе возбуждается исключение типа, для которого нет соответствующего оператора `catch`, то в качестве обработки по умолчанию происходит аварийное завершение работы программы.

Простейший пример обработки исключений показан в программе 8.1. В ней есть блок `try`, содержащий три оператора, и блок `catch` для перехвата и обработки целочисленного исключения.

```
#include <iostream.h>

void main()
{
  cout << "Начало\n";
  try {
    cout << "Оператор внутри блока try\n";
    throw 10;   // возбуждение исключения типа int
    cout << "Этот оператор не будет выполняться";
  }
  catch ( int i ) { // перехват исключения типа int
    cout << "Перехвачено исключение со значением ";
    cout << i << "\n";
  }
  cout << "Завершение работы программы";
}
```

Программа 8.1. Обработка исключения.

Программа 8.1 выводит на экран следующие сообщения:

```
Начало
Оператор внутри блока try
Перехвачено исключение со значением 10
Завершение работы программы
```

Исключения могут генерироваться внутри функций, которые вызываются из блока `try`. В программе 8.2 приведен соответствующий пример, в котором также показано расположение нескольких блоков `catch` для перехвата разнотипных исключений.

```
#include <iostream.h>
#include <string.h>

// Класс "Информация об исключении"
class CExceptionMsg {
  public:
    CExceptionMsg( char* sMsg )
      { msg = new char[strlen( sMsg )+1]; strcpy( msg, sMsg ); }
    ~CExceptionMsg()
      { delete msg; }
    char* GetMsg()
      { return msg; }

  private:
    char* msg;
};

void func(int test)
{
```

```
  if ( test == 0 )
    throw "Значение равно нулю";
  if ( test == 2 )
    throw new CExceptionMsg( "Exception Info: значение равно 2" );
  throw test;
}

void main()
{
  cout << "Начало\n";
  try {
    func(0);
    func(1);
    func(2);
    func(3);
  }
  catch (int i) {
    cout << "Перехвачено целое число: " << i << "\n";
  }
  catch(char *str) {
    cout << "Перехвачена строка: " << str << "\n";
  }
  catch(CExceptionMsg* pEx) {
    cout << "Перехвачен объект-исключение с сообщением: ";
    cout << pEx->GetMsg() << "\n";
    delete pEx;
  }

  cout << "Завершение работы программы";
}
```

Программа 8.2. Обработка разнотипных исключений.

На экран программа 8.2 выводит следующие сообщения:

```
Начало
Перехвачена строка: Значение равно нулю
Завершение работы программы
```

Как видите, каждый оператор `catch` перехватывает только исключения определенного типа. Для обработки исключений произвольного типа (например, чтобы выдать сообщение о неизвестной ошибке и завершить работу программы) можно завести блок `catch`, у которого вместо типа данных указано многоточие:

```
catch(...) {
    cout << "Перехвачено какое-то исключение.\n";
    exit( 1 );
  }
```

Оператор `catch(...)` бывает удобно использовать в качестве последнего оператора в группе операторов `catch`. В этом случае `catch(...)` по умолчании становится блоком, который перехватывает все необработанные исключения.

2. Традиционные способы обработки ошибок

Приведем несколько традиционных способов обработки ошибок на примере проверки успешности динамического выделения памяти:

```
// 1-й способ: проверка условия и завершение работы программы
int* pData = new int[10000];
if ( pData == NULL )
  {
  cout << "Не хватило памяти!\n";
  exit(1);
  }

// 2-й способ: проверка условия и выход из функции с возвратом
// некоторого оговоренного кода ошибки
int* pData = new int[10000];
if ( pData == NULL )
  return -10;

// 3-й способ: никакой проверки в надежде на то, что все будет хорошо
int* pData = new int[10000];

// 4-й способ: проверка условия и вызов функции обработки ошибки
int* pData = new int[10000];
if ( pData == NULL )
  NoMemoryError();

// 5-й способ: проверка условия с помощью библиотечного макроса assert(),
// который в случае ложности условия выводит данную строку исходного
// текста в поток вывода и завершает работу программы
int* pData = new int[10000];
assert( pData != NULL );
```

1-й способ "Завершить программу" слишком прямолинеен и опасен, например, тем, что при аварийном выходе могут пропасть важные несохраненные данные. 5-й способ является разновидностью аварийного завершения, которое оформлено более коротким образом. Для многих ошибок требуется обеспечивать более разумную обработку.

2-й способ "Возвратить значение-код ошибки" не всегда возможно применить, так как в чистом виде не существует приемлемого значения для "ошибки". Например, у некоторой целочисленной функции `int func()` может оказаться так, что любое значение `int` является корректным возвращаемым значением и ни одно число нельзя будет считать кодом ошибки. Если даже такой подход возможен, это зачастую неудобно, так как при каждом вызове нужно проверять возвращаемое значение. Это может значительно удлинить программу. Поэтому такой подход редко используется для систематического обнаружения всех ошибок.

3-й способ "Не проверять ошибки и оставить программу в пост-ошибочном состоянии" небезопасен, так как вызывающая функция может не заметить, что в вызванной функции произошло что-то неподходящее.

4-й способ "Вызвать функцию-обработчик ошибки" лучше предыдущих тем, что в программе явно отделяются фрагменты для обработки ошибок. К сожалению, решение о вызове обработчика опять приходится принимать программисту на основе проверки кодов или условий возникновения ошибки. Кроме того, проверку надо ор-

ганизовывать непосредственно в том месте, где произошла ошибка или после вызова функции, возвращающей код ошибки.

Например, многие функции библиотеки Си устанавливают глобальную переменную `errno` для индикации ошибки. Поэтому в программах без последовательных проверок `errno` будут появляться ошибки, вызванные ошибочными значениями, возвращаемые предыдущими вызовами. Более того, использование одной глобальной переменной для различных ошибок недопустимо в многозадачных программах.

Механизм обработки исключений является альтернативой традиционным способам во многих случаях, когда традиционные способы недостаточны, некрасивы и чреваты ошибками. Можно выделить два основных преимущества нового механизма:

1) Изоляция кода обработки от "обычного" кода. Программа становится более читаемой и легче поддающейся обработке. Механизм обработки исключений обеспечивает более формальный стиль обработки ситуаций, упрощая взаимодействие между независимо написанными фрагментами программы.
2) Жесткая реакция на необработанные ошибки. Реакцией на ситуацию по умолчанию (особенно в библиотечной функции) является завершение программы. Традиционная же реакция заключается в продолжении выполнения программы в надежде на лучший исход. Таким образом, обработка исключений программу более "хрупкой" в том смысле, что требуется приложить больше усилий для того, чтобы заставить программу работать нужным образом. Это выглядит предпочтительнее, так как ошибочные результаты будут получены во время написания программы (а не после того, как программы будет написана и передана пользователям). Когда завершение программы неприемлемо, традиционный подход может быть имитирован обработкой всех ситуаций или обработкой всех ситуаций, принадлежащих определенному классу.

Средства обработки исключений могут быть рассмотрены как динамический (run-time) аналог проверки типов и обработки неоднозначностей во время компиляции. Поэтому важность процесса написания программы возрастет, а заставить программу работать труднее, чем на Си. Тем не менее, генерируемый код с большой вероятностью будет функционировать как предполагалось и приемлемо работать как составная часть большой программы. Кроме того, программа будет более понятна для других программистов и легче поддаваться обработке. Просто обработка ситуаций является средством для поддержки "хорошего стиля" (подобно тому, как "хороший стиль" поддерживается другими средствами Си++), которое может быть использовано лишь неформально и неполно в языках, подобных Си.

Необходимо понять, что обработка исключений остается сложной задачей и механизм обработки исключений – хотя и больше формализован, чем заменяемые им средства – все еще остается относительно неструктурированным по сравнению с языковыми средствами для локального управления выполнением программы.

3. Динамическая идентификация типа RTTI

Одно из проявлений полиморфизма в Си++ – виртуальные функции в подклассах, которые можно вызывать через указатели на родительские классы. При таком подходе указатель родительского класса может использоваться либо для указания на объект родительского класса, либо для указания на объект любого подкласса, унасле-

дованного от этого родительского класса. На этапе компиляции нельзя узнать, на объект какого класса указывает подобный указатель. Средства динамической идентификации типа (Run-Time Type Information, RTTI) позволяют определить тип объекта во время выполнения программы. Возможности RTTI используются довольно редко, но о них полезно иметь представление, чтобы лучше понимать свойства полиморфизма.

Для получения информации о типе объекта предназначен оператор `typeid`. Чтобы использовать его, в программу надо включить заголовочный файл `typeinfo.h`. Оператор `typeid` допускает две формы использования:

```
typeid(obj)
typeid(typename)
```

Здесь `obj` – этот тот объект, информацию о типе которого необходимое получить. Параметр `typename` – имя типа, информацию о котором надо получить (например, `int` или `double`).

Оператор `typeid` возвращает ссылку на объект класса `type_info`, который предназначен для хранения информации о типе. В классе `type_info` наиболее часто используемыми членами являются:

```
boot operator==(const type_info& obj);
boot operator!=(const type_info& obj);
const char* name();
```

Перегруженные операторы `==` и `!=` применяются для сравнения типов объектов. В операторах сравнения обычно совместно используются обе формы оператора `typeid`. Функция `name()` возвращает указатель на строку с именем типа.

Оператор `typeid` позволяет получать типы разных объектов, но он наиболее полезен в случаях, когда в качестве его аргумента задается указатель полиморфного родительского класса. Тогда оператор автоматически возвращает тип реального объекта, на который указывает указатель. Этим объектом может быть как объект родительского класса, так и объект любого его подкласса. То же самое относится и к ссылкам.

Применение оператора `typeid` демонстрируется в программе 8.3. Сначала с помощью этого оператора мы получаем информацию об одном из встроенных типов данных Си++ (типе `int`). Затем оператор `typeid` дает возможность вывести на экран типы объектов, на которые указывает указатель `p`, являющийся указателем родительского класса `CBaseClass`.

```
#include <iostream.h>
#include <typeinfo.h>

class CBaseClass {
  virtual void f() {;}
};
class CDerivedl: public CBaseClass {
  virtual void f() {;}
};
class CDerived2: public CBaseClass {
  virtual void f() {;}
};

void main()
{
  CBaseClass *p, base_obj;
  CDerivedl obj1;
```

```
  CDerived2 obj2;
  int i = 5;

  cout << "Тип переменной i - это " << typeid(i).name() << '\n';

  p = &base_obj;
  cout << "p указывает на объект типа " << typeid(*p).name() << '\n';

  p = &obj1;
  cout << "p указывает на объект типа " << typeid(*p).name() << '\n';

  p = &obj2;
  cout << "p указывает на объект типа " << typeid(*p).name() << '\n';
}
```

Программа 8.3. Использование оператора `typeid`.

Программа 8.3 выводит на экран следующие сообщения:

```
Тип переменной i - это int
p указывает на объект типа class CBaseClass
p указывает на объект типа class CDerivedl
p указывает на объект типа class CDerived2
```

4. Динамическое преобразование типа данных

В Си++ продолжают поддерживаться характерные для языка Си операторы явного приведения типов данных (например, `(CDerived2*)base_obj`), но вместе с тем имеется несколько новых. Это операторы `dynamic_cast`, `const_cast`, `reinterpret_cast` и `static_cast`. Из них чаще всего используется оператор `dynamic_cast`.

Оператор `dynamic_cast` выполняет приведение типов в динамическом режиме, что позволяет контролировать правильность этой операции во время работы программы. У оператора `dynamic_cast` следующий синтаксис:

```
dynamic_cast<целевой тип>(выражение)
```

Здесь `целевой_тип` – это тип, к которому надо привести параметр `выражение`. И выражение, и целевой тип могут быть указателями или ссылками. Таким образом, оператор `dynamic_cast` используется для приведения типа одного указателя к типу другого или типа одной ссылки к типу другой.

Основное назначение оператора `dynamic_cast` заключается в реализации операции приведения полиморфных типов. Например, пусть дано два полиморфных класса `B` и `D`, причем класс `D` является производным от класса `B`. Тогда оператор `dynamic_cast` всегда может привести тип указателя `D*` к типу указателя `B*`. Это возможно потому, что указатель базового класса всегда может указывать на объект производного класса. Оператор `dynamic_cast` может также привести тип указателя `B*` к типу указателя `D*`, но только в том случае, если объект, на который указывает указатель, действительно является объектом типа `D`.

Оператор `dynamic_cast` выполняется успешно, когда указатель (или ссылка) после приведения типов становится указателем (или ссылкой) либо на объект целевого типа, либо на объект производного от целевого типа. В противном случае приведения типов не происходит. При неудачной попытке приведения типов результатом выполнения оператора `dynamic_cast` является нулевой указатель, если в операции ис-

пользовались указатели. Если же в операции использовались ссылки, возбуждается исключительная ситуация класса `bad_cast`.

Применение оператора `dynamic_cast` показано в программе 8.4.

```
#include <iostream.h>

class CBase {
  public:
    virtual void f()  { cout << "Внутри класса CBase\n"; }
};

class CDerived: public CBase {
  public:
    void f()          { cout << "Внутри класса CDerived\n"; }
};

void main()
{
  CBase *bp, b_obj;
  CDerived *dp, d_obj;

  dp = dynamic_cast<CDerived*>(&d_obj);
  if (dp)
    {
    cout << "Тип CDerived* к типу CDerived* приведен успешно\n";
    dp->f();
    }
  else
    cout << "Ошибка\n";
  cout << '\n';

  bp = dynamic_cast<CBase*>(&d_obj);
  if (bp)
    {
    cout << "Тип CDerived* к типу CBase* приведен успешно\n";
    bp->f();
    }
  else
    cout << "Ошибка\n";
  cout << '\n';

  bp = dynamic_cast<CBase*>(&b_obj);
  if (bp)
    {
    cout << "Тип CBase* к типу CBase* приведен успешно\n";
    bp->f();
    }
  else
    cout << "Ошибка\n";
  cout << '\n';

  dp = dynamic_cast<CDerived*>(&b_obj) ;
  if (dp)
    cout << "Ошибка\n";
  else
    cout << "Тип CBase* к типу CDerived* не приведен\n";
  cout << '\n';

  bp = &d_obj; // bp указывает на объект класса CDerived
```

```
  dp = dynamic_cast<CDerived*>(bp);
  if (dp)
    {
    cout << "Указатель bp к типу CDerived* приведен успешно, т.к. bp в "
            "действительности указывает на объект типа CDerived\n";
    dp->f() ;
    }
  else
    cout << "Ошибка\n";
  cout << '\n';

  bp = &b_obj; // bp указывает на объект класса CBase
  dp = dynamic_cast<CDerived*>(bp);
  if (dp)
    cout << "Ошибка\n";
  else
    cout << "Указатель Ьр к типу CDerived* не приведен, т.к. Ьр в "
            "действительности указывает на объект типа CBase\n";
  cout << '\n';

  dp = &d_obj; // dp указывает на объект типа CDerived
  bp = dynamic_cast<CBase*>(dp);
  if (bp)
    {
    cout << "Указатель dp к типу CBase* приведен успешно\n";
    bp->f();
    }
  else
    cout << "Ошибка\n";
}
```

Программа 8.4. Использование оператора динамического преобразования типа.

Программа 8.4 выводит на экран следующие сообщения:

```
Тип CDerived* к типу CDerived* приведен успешно
Внутри класса CDerived

Тип CDerived* к типу CBase* приведен успешно
Внутри класса CDerived

Тип CBase* к типу CBase* приведен успешно
Внутри класса CBase

Тип CBase* к типу CDerived* не приведен

Указатель bp к типу CDerived* приведен успешно, т.к. bp в
действительности указывает на объект типа CDerived
Внутри класса CDerived

Указатель Ьр к типу CDerived* не приведен, т.к. Ьр в
действительности указывает на объект типа CBase

Указатель dp к типу CBase* приведен успешно
Внутри класса CDerived
```

5. Упражнения

1) Тип исключения должен соответствовать типу, заданному в операторе `catch`**.** Измените в программе 8.1 тип данных в операторе `catch` на `double`. Убедитесь, что исключение не будет перехватываться и программа завершается аварийно.
2) Что неправильно в данном фрагменте? Как его исправить?
```
try {
  // ...
  throw 'a';
  // ...
}
catch (char*) {
  // ...
}
```
3) Поочередно внесите такие изменения в программу 8.2, чтобы она сгенерировала исключения каждого их трех возможных типов.
4) Ниже приведен каркас функции `divide()`, которая возвращает результат деления числа `a` на `b`.
```
double divide(double a, double b)
{
  // добавьте обработку ошибок
  return a/b;
}
```
Добавьте в эту функцию обработку исключительных ситуаций для отслеживания деления на ноль. С помощью соответствующей тестовой программы убедитесь, что функция работает корректно.
5) Оператор `typeid` может определить тип полиморфного объекта по указателю на объект базового класса. В программе 8.3 попробуйте ликвидировать полиморфизм у класса `BaseClass` (уберите служебное слово `virtual` в прототипе функции `f()`) и посмотрите, что программа будет выводить на экран.
6) Правилен ли следующий оператор?
```
cout << typeid( float ).name();
```
7) Дан фрагмент программы. Как определить, является ли `p` указателем на объект класса `D2`?
```
class B {
  virtual void f() {;}
};
class D1 : public B {
  void f() {;}
};
class D2 : public B {
  void f() {;}
};
void main()
{
  B* p;
  ...
}
```
8) Применительно к фрагменту программы из упр. 7 покажите, как с помощью оператора `dynamic_cast` сделать так, чтобы указатель `p` указывал на некоторый объект `obj` только в том случае, если этот объект `obj` является объектом класса `D2`.

Лекция 9. Стандартная библиотека шаблонов STL

1. Введение

Абстрактные типы данных (например, динамические массивы, связные списки, стеки, очереди, деревья, множества, словари) используются во многих программах. Поскольку эти типы данных являются типичными, хорошо было бы иметь их в качестве многократно используемых компонент. Богатый набор структур данных был добавлен в стандарт Си++ в виде дополнительной библиотеки – STL. В этой библиотеке содержатся классы для создания динамических массивов, списков, множеств, ассоциативных списков (словарей), стеков, очередей, очередей с приоритетами.

Библиотека STL (STL – Standard Template Library, стандартная библиотека шаблонов) появилась в результате многолетних исследований под руководством Александра Степанова и Менга Ли из компании Hewlett-Packard и Дэвида Мюссера из Rensselaer Polytechnic Institute.

Одна из наиболее необычных идей в STL – это обобщенные алгоритмы. Обобщенные алгоритмы в STL напоминают параметризованные классы (которые часто называются классами-шаблонами). Идея параметризации в Си++ может применяться к отдельным функциям. Чтобы понять концепцию обобщенных алгоритмов, кратко рассмотрим использование инкапсуляции в большинстве объектных библиотек.

Инкапсуляция в ООП – одна из главных целей. Хорошо разработанный класс старается инкапсулировать все состояние и поведение, необходимые для выполнения своей задачи, и в то же время скрывает как можно больше деталей внутреннего устройства. Во многих предшествующих объектно-ориентированных библиотеках этот философский подход воплощался в контейнерных классах, обладающих широкой функциональностью и богатым интерфейсом (например, классы `CList`, `CMap` и `CArray` в библиотеке MFC).

Разработчики STL пошли в совершенно другом направлении. Поведение, обеспечиваемое их стандартными компонентами, является минимальным. Вместо этого каждый компонент предназначен для функционирования совместно с большим набором обобщенных алгоритмов (например, для поиска и сортировки), имеющихся в библиотеке. Эти алгоритмы не зависят от контейнеров и поэтому могут работать со многими различными типами.

Отделяя функционирование алгоритмов от контейнерных классов, библиотека STL значительно выигрывает в размере – как в объеме самой библиотеки, так и в генерируемом коде. Вместо того чтобы дублировать алгоритмы для дюжины контейнерных классов, одно-единственное описание библиотечной функции может использоваться с любым контейнером. Более того, описание этих функций является настолько общим, что они могут применяться с обычными массивами, указателями и встроенными типами данных.

2. Основные элементы STL

Ядро библиотеки STL образуют три основополагающих элемента: *контейнеры*, *алгоритмы* и *итераторы*. В программах на Си++ эти элементы обычно функционируют в тесной взаимосвязи.

Контейнеры (*containers*) – это объекты, предназначенные для хранения других объектов. Контейнеры бывают различных типов. Например, в классе `vector` опреде-

ляется динамический массив, в классе `queue` – очередь, в классе `list` – связный список. В каждом классе-контейнере определен набор функций для работы с ним. Например, в списке есть функции для вставки и удаления элементов, для слияния двух списков. В стеке есть функции для помещения элемента в стек и извлечения элемента из стека.

Алгоритмы (*algorithms*) выполняют операции над содержимым контейнеров. Существуют алгоритмы для инициализации, сортировки, поиска или замены содержимого контейнеров. Многие алгоритмы предназначены для работы с *последовательностью* (*sequence*), которая представляет собой связный список элементов внутри контейнера.

Итераторы (*iterators*) – это объекты, которые по отношению к контейнерам играют роль указателей. Они позволяют получать доступ к содержимому контейнера примерно так же, как указатели используются для доступа к элементам массива.

В качестве примера рассмотрим применение алгоритма `find` для поиска первого вхождения заданного значения в контейнере. Итераторы стандартной библиотеки состоят из пар значений, отмечающих начало и конец структуры данных внутри контейнера. Алгоритм `find` использует пару итераторов и ищет первое вхождение заданного значения. Функция, реализующая алгоритм `find`, определена следующим образом:

```
template<class InputIterator, class T>
  InputIterator find( InputIterator first, InputIterator last,
                      const T& value )
{
  for ( ; first != last; ++first )
    if ( *first == value )
      break;
  return first;
}
```

Алгоритм будет работать с объектами любого класса, в том числе и с обычными массивами. В программе 9.1 демонстрируется поиск первого вхождения значения 7 в целочисленном массиве.

```
#include <iostream.h>
#include <algorithm>  // У заголовочных файлов STL нет расширения

// Обязательный оператор для подключения "пространства имен std"
// библиотеки STL к глобальному пространству имен программы
using namespace std;

void main()
{
  int data[100];
  memset( data, 0, sizeof( data ) );
  data[50] = 7;
  int* where = find( data, data+100, 7 );
  cout << *where << '\n';
}
```

Программа 9.1. Применение алгоритма `find` к целочисленному массиву.

В программе 9.2 показано, как выполнить поиск первого вхождения в целочисленном списке. Вызов функции `find()` выглядит ненамного сложнее, чем в программе 9.1.

```
#include <iostream.h>
#include <algorithm>
#include <list>

using namespace std;

void main()
{
  list<int> intList;
  for ( int i = 0; i < 100; i++ )
    intList.push_back( i );

  list<int>::iterator where;
  where = find( intList.begin(), intList.end(), 7 );
  cout << *where << '\n';
}
```

Программа 9.2. Применение алгоритма `find` к связному списку, в котором хранятся целые числа.

3. Итераторы

В библиотеке STL служебные объекты-итераторы играют очень важную роль. Они обеспечивают пользователям и алгоритмам доступ к содержимому контейнеров. Можно считать, что итератор – это объект, выполняющий по отношению к контейнеру роль указателя, позволяющего организовать прохода по всем значениям, хранящимся в контейнере.

Как и обычные указатели, итераторы применяются для различных целей. Итератор может обозначать конкретное значение (как указатель указывает на конкретную переменную). С другой стороны, пара итераторов может задавать диапазон значений (два указателя отмечают границы непрерывного участка памяти). Однако в случае итераторов описываемые значения расположены друг за другом не физически, а логически. Это происходит, поскольку они взяты из одного контейнера и, следовательно, следуют друг за другом в том порядке, как они хранились в контейнере.

Обычные указатели иногда принимают значение `NULL` – то есть не указывают ни на какой объект. Итераторы аналогичным образом могут не определять какое-либо конкретное значение. Подобно тому, как логической ошибкой является разыменование и использование указателя со значением `NULL`, нельзя разыменовывать и применять итератор, который не определяет никакого значения.

Когда в языке Си++ используются два указателя, которые ограничивают область памяти, по соглашению второй указатель не рассматривается как часть области. Например, массив с именем `x` и длиной 10 иногда описывается как занимающий область от `x` до `(x+10)`, хотя элемент `(x+10)` не является частью массива. На самом деле указатель `(x+10)` ссылается на запредельный элемент, то есть элемент, следующий за последним элементом описываемого диапазона. Итераторы определяют диапазон аналогичным образом. Второе значение рассматривается не как часть определяемого диапазона, а как запредельный элемент, описывающий значение, следующее в последовательности за последним значением из заданного диапазона.

Подобно традиционным указателям, основное действие, которое модифицирует итератор, это инкрементирование (оператор ++). Когда инкрементирование применяется к итератору, который указывает на последнее значение в последовательности, то итератор принимает описанное выше запредельное значение. Оператор разыменования (*) осуществляет доступ к данным, определяемым итератором.

Диапазон может описывать весь контейнер при задании итератора, который указывающего на первый элемент, и итератора, имеющего специальное "последнее" значение. Диапазоны могут описывать последовательности, входящие в контейнер (двум итераторам присваиваются конкретные значения). В стандартных контейнерах итератор начала контейнера возвращается функцией `begin()`, а итератор, обозначающий конец контейнера, – функцией `end()`.

4. Объекты-функции

Некоторые алгоритмы из библиотеки STL требуют функций в качестве параметров. Примером служит алгоритм `for_each()`, который вызывает функцию, переданную в качестве параметра, для каждого значения в контейнере. В программе 9.3 показано, как с помощью алгоритма `for_each()` можно вывести все элементы связного списка на экран.

```
#include <iostream.h>
#include <algorithm>
#include <list>

using namespace std;

void PrintElement( int value )
{
  cout << "Список содержит значение " << value << '\n';
}

void main()
{
  list<int> intList;
  for ( int i = 0; i < 100; i++ )
    intList.push_back( i );
  for_each( intList.begin(), intList.end(), PrintElement );
}
```

Программа 9.3. Применение алгоритма `for_each` с параметром-функцией `PrintElement`.

Понятие функции в STL было обобщено до понятия объекта-функции. Объект-функция – это объект класса, в котором перегружен оператор вызова функции "круглые скобки" `()`. В ряде случаев удобно заменить функции на объекты-функции. Когда объект-функция используется в качестве функции, то при ее вызове используется оператор "круглые скобки".

Рассмотрим следующее определение класса:

```
class CBiggerThanThree {
  public:
    bool operator () (int v)
      { return v > 3; }
};
```

Если мы создадим объект класса `CBiggerThanThree`, то каждый раз, когда мы будем ссылаться на него с использованием синтаксиса вызова функции, то будет вызываться перегруженный оператор "круглые скобки". Следующий шаг – обобщить этот класс, добавив к нему конструктор и неизменяемое поле данных, которое устанавливается конструктором:

```
class CBiggerThan {
  public:
    CBiggerThan( int x ) : testValue(x) {;}
    bool operator () (int val)
      { return val>testValue; }
    const int testValue;
};
```

В результате мы получили функцию общего вида, которая выполняет целочисленное сравнение "больше чем *X*", где значение *X* определяется при создании объекта класса. Подобную функцию можно использовать в качестве параметра при вызове некоторых алгоритмов STL, например, алгоритма логического поиска `find_if`. Ниже приведен пример вызова этого алгоритма для поиска в целочисленном списке первого элемента со значением больше 12:

```
list<int>::iterator firstBig = find_if( intList.begin(),
                                        intList.end(),
                                        CBiggerThan(12) );
```

5. Пример программы: инвентаризация

Рассмотрим пример, иллюстрирующий создание и обработку объектов в библиотеке STL. Допустим, что требуется написать складскую программу для учета некоторых приспособлений – виджетов. Это какие-то устройства, различаемые по идентификационным номерам:

```
class CWidget
{
  public:
    CWidget(int a) : id(a) {}
    CWidget() : id(0) {}
    int id;
};
```

Для упорядочения и сравнения объектов-виджетов предназначены несколько перегруженных операторов:

```
bool operator== ( const CWidget& lhs, const CWidget& rhs )
{
  return lhs.id == rhs.id;
}
bool operator!= ( const CWidget& lhs, const CWidget& rhs )
{
  return lhs.id != rhs.id;
}
bool operator< ( const CWidget& lhs, const CWidget& rhs )
{
  return lhs.id < rhs.id;
```

```
}
bool operator> ( const CWidget& lhs, const CWidget& rhs )
{
  return lhs.id > rhs.id;
}
```

Виджеты в программе описываются двумя списками. В одном хранятся виджеты, имеющиеся в данный момент на складе. Второй список содержит типы виджетов, которые были заказаны покупателями. Первый список содержит собственно виджеты, а второй – идентификационные типы виджетов. Для работы со складом требуются две функции-члена:

- `Order()` – обслуживание заказа;
- `Receive()` – отслеживание поставок новых виджетов на склад.

```
class CInventory
{
  public:
    void Order( int wid );    // обработка заказа виджета типа wid
    void Receive( int wid ); // получение виджета типа wid

  private:
    list<CWidget> on_hand;
    list<int> on_order;
};
```

Когда поступает новый виджет, надо сравнить его идентификационный номер со списком заказанных виджетов. С помощью алгоритма `find()` можно найти виджет в списке заказов. Если виджет был заказан, то его надо немедленно переслать покупателю. В противном случае он добавляется к списку виджетов на складе.

```
void CInventory::Receive( int wid )
{
  cout << "Пришла партия виджетов типа " << wid << endl;
  list<int>::iterator weNeed = find( on_order.begin(),
                                     on_order.end(), wid );
  if ( weNeed != on_order.end() )
    {
    cout << "Отправить " << wid << " покупателю \n";
    on_order.erase(weNeed);
    }
  else
    on_hand.push_front( CWidget(wid) );
}
```

Когда покупатель заказывает новый виджет, мы просматриваем с помощью функции `find_if()` список имеющихся на складе виджетов, чтобы определить, нельзя ли обслужить заказ немедленно. Для этого определена унарная функция, которая берет в качестве аргумента виджет и определяет, соответствует ли он требуемому типу. Эта функция записывается следующим образом:

```
class CWidgetTester {
  public:
    CWidgetTester( int t ) : testid(t) { }
    bool operator() ( const CWidget& wid )
```

```
      { return wid.id == testid; }
    const int testid;
};
```

Функция, обслуживающая заказы виджетов, выглядит так:

```
void CInventory::Order( int wid )
{
  cout << "Получен заказ на виджеты типа " << wid << endl;
  list<CWidget>::iterator weHave = find_if( on_hand.begin(),
                  on_hand.end(), CWidgetTester(wid) );
  if ( weHave != on_hand.end() )
    {
    cout << "Отправить покупателю виджет " << weHave->id << endl;
    on_hand.erase(weHave);
    }
  else
    {
    cout << "Заказать виджет типа " << wid << endl;
    on_order.push_front(wid);
    };
}
```

Главная функция для тестирования класса `CInventory`, описывающего состояние склада и заказы, может выглядеть следующим образом:

```
void main()
{
  CInventory inv;
  inv.Receive( 5 );
  inv.Order( 10 );
  inv.Order( 5 );
  inv.Receive( 10 );
  inv.Receive( 10 );
}
```

6. Ассоциативные списки

В библиотеке STL реализованы различные АТД. Более сложным по сравнению со связным списком является ассоциативный список `map`, в котором каждому значению соответствует уникальный ключ. Частным случаем ассоциативного списка является обычный массив, в нем ключом является целочисленный индекс. В ассоциативном списке ключ может быть любого типа, и в общем можно сказать, что ассоциативный список представляет собой список пар ключ/значение. Важное достоинство ассоциативных списков – возможность получения значения по данному ключу. Например, в ассоциативном списке можно хранить имена телефонных абонентов в качестве ключей, а номера телефонов – в качестве значений.

В ассоциативном списке можно хранить только уникальные ключи. Дублирования ключей не допускается. Для создания ассоциативного списка с неуникальными ключами предназначен отдельный класс-контейнер `multimap`.

Пары ключ/значения в ассоциативном списке хранятся в виде объектов класса `pair`. Он имеет следующее описание:

```
template <class Ktype, class Vtype> struct pair {
```

```
    Ktype first;      // Ключ
    Vtype second;          // Значение
    // Конструкторы
    pair();
    pair( const Ktype& k, const Vtype& v );
  }
```

В программе 9.4 приведен пример ассоциативного списка, предназначенного для хранения десяти пар ключ/значения следующего вида:

```
A  0
B  1
C  2
...
J  9
```

Пользователь может набрать на клавиатуре ключ (одну из букв от `A` до `J`) и программа выведет на экран соответствующее этому ключу значение.

Поиск нужного значения по заданному ключу выполняется с помощью алгоритма `find()`. Эта функция возвращает итератор, указывающий на соответствующий ключу элемент или на итератор конца списка, если указанный ключ не найден.

```
#include <iostream>
#include <map>
using namespace std;

void main()
{
  map<char, int> m;
  // Размещение пар буква/число в ассоциативном списке
  for ( int i = 0; i < 10; i++ )
    m['A'+i] = i;

  char ch;
  cout << "Введите ключ: ";
  cin >> ch;

  map<char, int>::iterator p;

  // Поиск значения по заданному ключу
  p = m.find( ch );
  if ( p != m.end() )
    cout << p->second;
  else
    cout << "Такого ключа в ассоциативном списке нет\n";
}
```

Программа 9.4. Пример ассоциативного списка.

Как и в других контейнерах STL, в ассоциативных списках можно хранить значения любых типов данных. В программе 9.5 показан пример списка строк, которые проиндексированы тоже строками. Для хранения строк используется строковый класс STL `string`. В этом классе перегружены операторы (например, +, -, ==), позволяющие записывать операции со строками в более естественном виде (например, сравнивать их с помощью выражения `s1==s2`, а не `strcmp(s1, s2)`). Программа 9.5 предназначена для хранения слов вместе с их антонимами (противоположными по смыслу). Для поиска значения по ключу вместо алгоритма `find()` можно использовать перегруженный в классе `map` оператор индекса `[]`.

```
#include <iostream>
#include <map>
#include <string>
#pragma warning( disable : 4786 )
using namespace std;

void main()
{
  map<string, string> m;
  m["yes"] = "no";
  m["good"] = "bad";
  m["left"] = "right";
  m["up"] = "down";

  // Поиск антонима по заданному слову
  string str;
  cout << "Введите слово: ";
  cin >> str;

  if ( m[str] != "" )
    cout << "Антоним: " << m[str];
  else
    cout << "Такого слова в ассоциативном списке нет\n";
}
```

Программа 9.5. Ассоциативный список в качестве словаря антонимов.

7. Упражнения

1) Запустите программы 9.1, 9.2 и 9.3 и убедитесь, что понимаете, как они работают.
2) Сформируйте программу инвентаризации из приведенных в п.5 фрагментов. В исходный файл необходимо включить заголовочные файлы `algorithm` и `list`.
3) По аналогии с программой 9.5 разработайте ассоциативный список для хранения имен абонентов и их телефонных номеров. Имена и номера телефонов должны вводиться пользователем, а поиск нужного номера должен выполняться по введенному имени абонента.

Литература

1) Бадд Т. Объектно-ориентированное программирование в действии. СПб.: Питер, 1999. (Введение в ООП, написанное на основе курса лекций, читаемого автором этой книги студентам Орегонского университета. В книге удачно совмещены рассмотрение общих понятий ООП и описание особенностей конкретных языков. Среди рассматриваемых языков программирования есть Си++ и Java.)
2) Буч Г. Объектно-ориентированное проектирование с примерами применения. М.: Конкорд, 1992. (Подробное описание метода объектно-ориентированного проектирования и примеров его применения для разработки ряда программных систем на нескольких языках программирования. Описаны свойства сложных систем, подробно рассмотрены характеристики классов и объектов, много внимания уделяется проблеме классификации.)
3) Вирт Н. Алгоритмы + структуры данных = программы. М.: Мир, 1985. (Монография по структурному программированию на основе абстрактных типов данных. Приведено решение задачи о восьми ферзях с помощью структурного подхода.)
4) Страуструп Б. Язык программирования С++. Вторая редакция. К.:"ДиаСофт", 1993. (Подробное описание языка Си++, написанное разработчиком этого языка. Во вводной части приводится краткое, но достаточно содержательное описание различных парадигм программирования.)
5) Шилдт Г. Самоучитель Си++. СПб.: BHV–Санкт-Петербург, 2000. (Детальное описание синтаксических особенностей международного стандарта Си++. Книга рассчитана на программистов, знакомых с процедурным языком Си.)

Учебно-методическое издание

А.А. Богуславский, С.М. Соколов

Основы программирования на языке Си++
В 4-х частях.
(для студентов физико-математических факультетов
педагогических институтов)

Компьютерная верстка Богуславский А.А.

Технический редактор Пономарева В.В.

Сдано в набор 12.04.2002 Подписано в печать 16.04.2002
Формат 60x84x1/16 Бумага офсетная
Печ. л. 20,5 Учетно-изд.л. ____ Тираж 100
Лицензия ИД №06076 от 19.10.2001

140410 г.Коломна, Моск.обл., ул.Зеленая, 30. Коломенский государственный педагогический институт.