Глава 15

Знакомство с объектами

В этой главе…

- Объекты и классы
- Инкапсуляция в классах
- Способы использования объектов
- Наследование и полиморфизм
- Виртуальные подпрограммы и динамическое связывание
- Контейнеры и итераторы

Механизмы ООП позволяют моделировать понятия предметной области наиболее прямым и естественным путем.

Объектная ориентация = инкапсуляция + наследование.

А. Элиенс

Программы в этой книге до сих пор создавались на основе подпрограмм, возможно, распределенных по отдельным модулям. Этот *процедурно-ориентированный подход* позволяет решать относительно простые задачи, но для сложных он не обеспечивает нужной надежности и гибкости программирования. Усложнение решаемых задач стимулировало развитие другого подхода — *объектно-ориентированного программирования* (*ООП*).

Программа, построенная на принципах ООП, состоит из *объектов*, которые взаимодействуют друг с другом и с окружающим миром. Объекты являются конкретными экземплярами абстрактных понятий, реализованных как *классы*. Модель ООП позволяет более адекватно представлять объекты реального мира и полнее воплощать принцип "разделяй и властвуй".

"Три кита" ООП — это *инкапсуляция*, *наследование* и *полиморфизм*. Инкапсуляция сама по себе является основным средством так называемого *объектного программирования* (*ОП*). Инкапсуляция в объектах существенно отличается от инкапсуляции в модулях и обеспечивает более адекватную реализацию понятия абстрактного типа данных (см. главу 5). Инкапсуляция и наследование обеспечивают естественность моделирования объектов реального мира, независимость реализации операций от их применения и повторное использование кода.

Эта глава только знакомит с ОП и ООП, почти не касаясь таких важных тем, как объектно-ориентированный анализ и проектирование (см. [10, 18, 26, 29]). Чтобы не усложнять знакомства, рассматриваются очень простые задачи, хотя ООП проявляет свои преимущества именно в сложных задачах. Более того, ООП позволяет решать задачи, на которых другие подходы "захлебываются". Недаром практически все современные технологии и инструменты программирования являются объектно-ориентированными.

Введение в ОП представлено в разделе 15.1, в ООП — в разделе 15.2. В этих разделах использованы примеры, связанные с обработкой текстов. В разделе 15.3 рассматриваются несколько приемов ООП, основанных на использовании инкапсуляции, наследования и полиморфизма.

15.1. Основы объектного подхода

15.1.1. Инкапсуляция, классы и объекты

Познакомимся с объектами и их типами (классами) с помощью такого примера.

Пример 15.1. Предположим, что нужно прочитать два текста и сравнить последовательности символов в них при условии: конец строки независимо от его конкретного представления рассматривается в последовательности как пробел.

Схема решения очень проста:

```
пока тексты не закончились и не обнаружено их отличие,
  получить из текстов пару новых символов и сравнить их.
```

Символы текста с пробелами вместо концов строк образуют последовательность (*поток*) символов. Итак, главная задача — *сравнить два потока символов*. Возложим эту обязанность на пока что абстрактную сущность — *сравниватель потоков*. Действуя по описанной схеме, он должен получать из текстов пары новых символов и сравнивать их. Обязанность *выдать новые символы потоков* возложим на две другие сущности (два *потока символов*), к которым сравниватель обращается с требованием "дай символ". В ответ на это требование каждый поток возвращает следующий символ "своего" текста (в конце строки возвращается пробел). Взаимодействие сравнивателя и потоков — *объектов* в нашей задаче — представлено на рис. 15.1.

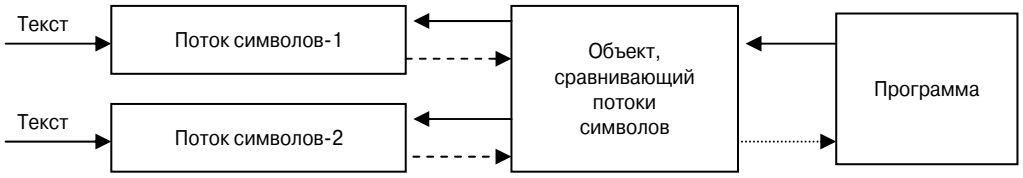


Рис. 15.1. *Взаимодействие сравнивателя и потоков*

- Важно пока то, что действия, необходимые для решения задачи, *распределены по различным исполнителям*. Дальше это позволит сделать очень важную вещь — *отделить использование действий от их реализации*.

Итак, выделены два объекта типа "поток символов" и один типа "сравниватель потоков", которые характеризуются собственным *поведением*, т.е. выполняемыми действиями. Поток выполняет требование "дай символ", обрабатывая текст, сравниватель сравнивает тексты, которые даются ему в виде потоков. Назовем эти действия *операциями*; назовем их, соответственно, `get` (получить символ) и `compareStreams` (сравнить потоки). Очевидно, что операцию `compareStreams` естественно реализовать путем многократных обращений к потокам с "требованием" `get` и сравнений полученных символов.

Определим другие операции наших объектов. Вначале объекты должны быть проинициализированы. Например, поток символов связывается с конкретным текстом, который будет читаться. Операцию инициализации назовем `init`. По окончании обработки текст нужно закрыть (операция `done`). Аналогичные операции (с неопределенным пока содержанием) введем и для сравнивателя.

Итак, объект-поток должен выполнять операции `init`, `get`, `done`, объект-сравниватель — `init`, `compareStreams`, `done`.

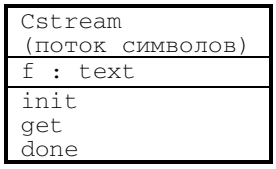
Объекты могут иметь внутренние состояния, изменяя их в процессе взаимодействия. Состояния представлены некоторыми данными. Например, поток обрабатывает текст, и состояния, которые возникают при чтении этого текста, естественно рассматривать как *состояния*

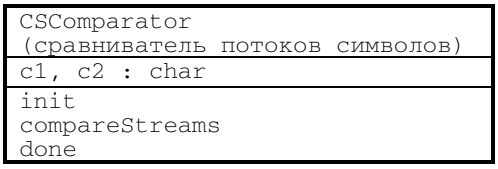
потока. Сравниватель потоков должен хранить пару символов, полученных из потоков, т.е. эта пара образует состояние сравнивателя.

Итак, объекты характеризуются *данными* (некоторым множеством значений) и *операциями*, которые они выполняют.

> Тип представителей некоторого абстрактного понятия, т.е. объектов с одними и теми же наборами операций и данных, называется *классом*, а представители — *объектами* класса. Данные в объектах класса называются *атрибутами*, набор операций — *интерфейсом* класса, а подпрограммы, которые реализуют операции, — *методами*.

Изобразим классы объектов "поток символов" и "сравниватель потоков" в виде прямоугольников с названиями, атрибутами и операциями.





Сравниватель использует операцию `get` класса потоков `CStream`, поэтому класс `CSComparator` является *клиентом* класса `Cstream`, а тот — *сервером*.

От абстрактного представления классов перейдем к их объявлению на языке Турбо Паскаль. Классы объявляют, используя *инкапсуляцию* (см. раздел 5.2) — именно она обеспечивает *защиту данных в объектах от несанкционированного доступа*. Например, текст, читаемый объектом-потоком, должен быть недоступным за пределами методов этого объекта, иначе результат сравнения потоков может быть ошибочным. Пара текущих символов в сравнивателе также должна быть недоступной за его предалами.

Кроме атрибутов и методов, объекты могут иметь вспомогательные подпрограммы, скрытые в объектах. Их также указывают в объявлении класса. В нашей задаче конец строки является особым случаем получения символа, поэтому обрабатывать конец строки будет функция `newLine`, вспомогательная к методу `get` класса `CStream`.

Класс `CStream` объявим на языке Турбо Паскаль как тип (в комментариях приведена часть пояснений).

```
type CStream = object
  {служебное слово "object" указывает на класс - тип объектов}
private {служебное слово "private" (частный, скрытый) указывает на}
        {сокрытие атрибутов и вспомогательных подпрограмм}
  f : text;   {файловая переменная, связанная с текстом}
  function newLine : char;
    {вспомогательная функция обработки конца строки в тексте;}
    {вызывается при условии, что текущим символом текста есть конец строки}
public {служебное слово "public" (публичный, открытый) указывает на}
       {интерфейс класса -- заголовки подпрограмм и открытые атрибуты}
  procedure init(fileName : string);
    {операция инициализации; параметр -- внешнее имя текста}
  function get : char;
    {операция возвращения нового символа}
  procedure done;
    {операция завершения работы с объектом}
end;
```

Как видим, операции класса объявлены вместе с атрибутами; синтаксически они являются "равноправными" компонентами объектов класса. Например, если объявить объект

`s : CStream`, то `s.f` обозначает файловую переменную, скрытую в объекте, а `s.init` — метод его инициализации. Как правило, хотя и не обязательно, атрибуты объявляются *скрытыми* — именно это гарантирует их защиту от несанкционированного доступа.
□

- Каждый объект класса, существующий во время выполнения программы, имеет собственные экземпляры полей-атрибутов. Коды подпрограмм и методов класса создаются в одном экземпляре, а индивидуальными в объектах являются экземпляры локальной памяти вызовов этих методов и подпрограмм.

✧ Объявление класса может состоять из произвольного количества секций, которые начинаются словами **public** или **private**. Однако лучше записать одну секцию открытых объявлений и одну — скрытых. Если ключевые слова **public** и **private** отсутствуют, то все компоненты считаются открытыми.

✧ В языке Турбо Паскаль объявления скрытых компонентов действуют в пределах модуля, содержащего объявление класса. В других языках программирования области действия могут быть другими.

✧ Инициализировать атрибуты в объявлении класса *запрещено*.

✧ Имена атрибутов и методов часто (хотя и не всегда) начинают со строчной буквы, имена классов — с прописной.

15.1.2. Использование объектов

Пример 15.2. Перейдем к решению задачи из примера 15.1. Класс можно объявить в программе или модуле; с целью повторного использования классы, как правило, объявляют в модулях.

Все объявления и определения, связанные с классами `CStream` и `CSComparator`, запишем в модуль `uCStream`. Однако вначале уточним детали взаимодействия объектов. Предположим, что два объекта-потока и объект-сравниватель объявляются в программе, использующей модуль. Потоки должны быть видимыми в методах сравнивателя, определенных в модуле, поэтому параметризуем методы потоками. Итак, объявим класс `CSComparator` так.

```
type CSComparator = object
private
  c1, c2 : char;   {символы, полученные из потоков}
 public
  procedure init(var s1, s2 : CStream);
    {операция инициализации, параметризованная потоками}
  function compareStreams(var s1, s2 : CStream) : boolean;
    {операция сравнения потоков}
  procedure done(var s1, s2 : CStream);
    {операция завершения работы со сравнивателем}
end;
```

Предположим также, что из метода `get` класса `CStream` в конце текста возвращается символ `#26`. Дадим этому значению имя `finch`. Оно должно быть общим для всех объектов и доступным за пределами класса, поэтому именование расположим не в объявлении класса, а на уровне модуля.

Методы и подпрограммы классов записывают в разделе реализации модуля с сокращенными заголовками, но с *квалификатором* — именем класса, для которого они объявлены. Эти составные имена позволяют объявлять в одном модуле несколько классов, а в разных классах — одноименные методы и подпрограммы.

Модуль `uCStream` представлен в листинге 15.1. Порядок расположения определений в разделе реализации не важен.

Подчеркнем еще раз: класс CStream гарантирует, что детали получения новых символов из текстов скрыты от сравнения; для сравнивателя важно лишь наличие символов. Обработку текстов можно модифицировать, сохранив интерфейс класса CStream — использование объектов этого класса не изменится. Например, класс CStream не имеет защиты от возможных ошибок, связанных с обработкой файлов (см. раздел 10.1), — добавьте эту защиту самостоятельно. Учтем также, что класс CSComparator позволяет закрывать еще неоткрытые потоки, поэтому его тоже можно усовершенствовать.

Листинг 15.1. Модуль с классами CStream и CSComparator

```
unit uCStream;
interface
  const finch = #26;
  type CStream = object
    ... {см. выше в тексте}
  end;

  type CSComparator = object
    ... {см. выше в тексте}
  end;

 implementation

  procedure CStream.init; {инициализация потока}
  begin assign(f, fileName); reset(f) end;

  procedure CStream.done; {завершение работы с потоком}
  begin close(f) end;

  function CStream.newLine; {обработка конца строки}
    {вызывается, когда в тексте закончена строка}
  begin
    if eof(f) then newLine := finch
    else begin readln(f); newLine := ' ' end
  end;

  function CStream.get; {возвращение следующего символа потока}
    var c : char;
  begin
    if eof(f) or eoln(f)
      then get := newLine
    else begin read(f, c); get := c end
  end;

  function CSComparator.init; {инициализация сравнивателя}
    var n1, n2 : string;
  begin
    writeln('Задайте имя первого текста'); readln(n1);
    s1. init(n1);
    writeln('Задайте имя второго текста'); readln(n2);
    if n1 = n2 then begin
      writeln('Нужно задать разные файлы.'); s1.done; exit;
    end;
    s2.init(n2);
  end;

  procedure CSComparator.done; {завершение работы со сравнивателем}
  begin s1.done; s2.done end;  {закрытие потоков}
```

```
  function CSComparator.compareStreams; {сравнение потоков}
  begin
    while true do begin
      c1 := s1.get; c2 := s2.get;
      if (c1 <> c2) or (c1 = finch) or (c2 = finch) then break
    end;
    compareStreams := (c1 = c2)
  end;

end.
```

- ✧ Однотипные переменные-объекты можно присваивать — при этом копируются значения всех атрибутов. Если среди атрибутов есть файловые переменные или указатели на файлы, последствия могут быть непредсказуемыми. Именно поэтому параметры-объекты в методах класса `CSComparator` объявлены как параметры-переменные, хотя вообще объекты могут быть и параметрами-значениями.

Наконец, программа решения исходной задачи очень проста.

Листинг 15.2. Модуль с классами `CStream` и `CSComparator`

```
program StreamsComparison;
  uses uCStreams;
  var s1, s2 : CStream;      {два потока}
      cmp    : CSComparator; {сравниватель}
begin
  cmp.init(s1, s2);
  writeln(cmp.compareStreams(s1, s2));
  cmp.done(s1, s2);
end.
```

□

15.1.3. Объекты как компоненты других объектов

Пример 15.3. Изменим объявления класса "сравниватель потоков" так, чтобы потоки были не объектами программы, а *атрибутами сравнивателя*. В этой ситуации методы сравнивателя не нужно параметризовать потоками. Объявление класса `CSComparator` приобретает такой вид.

```
type CSComparator = object
private
  c1, c2 : char;    {символы из потоков}
  s1, s2 : CStream; {потоки}
public
  procedure init;                    {инициализация}
  function compareStreams : boolean; {сравнение потоков}
  procedure done;      {завершение работы со сравнивателем}
end;
```

Поскольку имена новых атрибутов-потоков совпадают с именами параметров в предыдущем варианте класса `CSComparator`, раздел реализации модуля не изменится. Возможно, лучше было бы параметризовать метод `CSComparator.init` внешними именами файлов; получение имен должна была бы обеспечивать программа перед вызовом этого метода. Соответствующую модификацию класса и программы выполните самостоятельно.

□

Объект, компонентами которого являются другие объекты, называется *агрегатом*, а отношение между объектом и его компонентами — *отношением агрегирования*.

В приведенном примере агрегатом является сравниватель, его компонентами — потоки.

15.1.4. Создание и уничтожение объектов

Организуем взаимодействие объекта-клиента с объектом-сервером так, что атрибут клиента является *указателем на объект-сервер*. С помощью указателя объект-клиент сам создает и уничтожает объект-сервер.

Пример 15.4. Объявление класса `CSComparator` отличается от предыдущего тем, что вместо атрибутов-потоков оглашаются атрибуты-указатели на потоки.

```
  s1, s2 : ^CStream; {указатели на потоки}
```

Методы класса `CSComparator` в разделе реализации также изменяются.[31]

```
function CSComparator.init; {инициализация сравнивателя}
  var n1, n2 : string;
begin
  writeln('Задайте имя первого текста'); readln(n1);
  new(s1); s1^.init(n1);
  writeln('Задайте имя второго текста'); readln(n2);
  if n1 = n2 then begin
    writeln('Указаны одинаковые имена файлов. '); s1^.done; exit;
  end;
  new(s2); s2^.init(n2);
end;

procedure CSComparator. done; {завершение работы со сравнивателем}
  {-- закрытие потоков и освобождение памяти от объектов-потоков}
begin
  s1^.done; dispose(s1);
  s2^.done; dispose(s2)
end;
```

В теле метода `CSComparator.compareStreams` вместо имен потоков `s1` и `s2` записываются выражения `s1^` и `s2^`.

□

15.1.5. Доступность объектов-серверов в клиентах

Во время выполнения программы объект класса-клиента связывается с объектом класса-сервера, вызывая его методы. Для этого объект-сервер должен быть *доступным* в объекте-клиенте. В предыдущих подразделах представлены следующие способы обеспечить доступность объекта-сервера:

- объект-сервер (или ссылка на него) задан как *атрибут* в объявлении класса-клиента (см. подразд. 15.1.3 и 15.1.4);
- объект-сервер объявлен как *параметр* метода класса-клиента (см. подразд. 15.1.2);

Есть еще два очевидных способа:

- объект-сервер объявлен *локальной переменной* в методе класса-клиента;
- объект-сервер является *глобальным* относительно объекта-клиента.

[31] *Обработку ошибочных действий пользователя, связанных с вводом имен файлов, добавьте самостоятельно.*

Выбор способа зависит от характера связи между объектами в конкретной задаче. Важно *относительное время существования* объектов-клиентов и серверов. Атрибуты объектов или локальные переменные их методов существуют не дольше чем сами объекты. Объект-сервер, доступный через ссылку на него, может создаваться и уничтожаться в одном объекте-клиенте, а использоваться в другом, т.е. его существование может не зависеть от существования клиента. Аргументы в вызовах методов объекта-клиента и глобальные переменные также могут оставаться после уничтожения объекта-клиента.

Имеет значение также *продолжительность связи объектов*. Связь с объектом-сервером, который является параметром или локальной переменной метода клиента, существует только в пределах выполнения метода, т.е. является *временной*, а связь объекта со своим атрибутом или глобальным объектом *постоянна* (в пределах существования объекта).

15.2. Наследование и полиморфизм

15.2.1. Понятие наследования

Познакомимся с понятием наследования с помощью потоков символов (см. раздел 15.1).

Пример 15.5. Предположим, что длина строк текста не превышает 255; для чтения текста используем буфер — строку типа `string`. При вызовах функции `get` символы по одному читаются из этого буфера. Объявим буфер, его длину и текущую позицию в нем как скрытые атрибуты в объектах нового класса.

- ✧ Ограниченность длины строк в тексте предполагается только для упрощения изложения. В решении к задаче 15.1 будет представлен намного лучший вариант буферизированного чтения, который не ограничивает длину строк и контролирует ошибки ввода.

Объявим новый класс `CStreamBuf` потоков с интерфейсом класса `CStream`, но, учитывая использование буфера, с новыми методами и функцией `newLine`. Методы `init` и `get` придется изменить, но метод `done` этого не требует — его и атрибут `f` можно "позаимствовать" из класса `CStream`. Это "заимствование" реализуется как наследование классов.

> *Наследование классов* состоит в том, что в объявлении нового класса (*потомка*) используют другой, ранее объявленный класс (*отец*), к компонентам которого добавляют новые атрибуты и операции. Возможно, в классе-потомке изменяется реализация некоторых операций класса-отца.

В новом классе `CStreamBuf` указываются только новые атрибуты, связанные с буфером, а также новые и измененные методы.

```
type CStreamBuf = object (CStream) {ссылка на класс-отец}
private                            {добавленные компоненты}
  buf : string;                    {буфер с текущей строкой текста}
  bufLen, bufPos : byte; {длина буфера и текущая позиция в нем}
  {модифицированная функция обработки конца строки - теперь это}
  {функция заполнения буфера следующей строкой}
  function newLine : char;
public
  procedure init(fileName : string); {эти два метода}
  function get : char;               {модифицированы}
end;
```

Объекты класса-потомка `CStreamBuf` *наследуют* атрибуты класса-отца `CStream` (поле `f`), а также методы и подпрограммы, не объявленные в классе-потомке (метод `done`).

Запишем приведенное объявление класса CStreamBuf в модуле uCStreams; в раздел реализации модуля добавим новые подпрограммы.

Метод get увеличивает текущую позицию в буфере. Если после этого строка не исчерпана, возвращается символ в новой позиции, иначе — пробел или finch, полученный от newLine. Функция newLine вызывается, когда буфер прочитан. Если текст исчерпан, она возвращает finch, иначе она заполняет буфер следующей строкой текста и возвращает пробел как признак конца предыдущей строки.

```
procedure CStreamBuf.init; {инициализация}
begin
 {инициализация добавленных атрибутов}
  buf := ''; bufLen := 0; bufPos := 0;
 {инициализация унаследованных атрибутов выполняется унаследованным методом}
  CStream.init(fileName);
end;

function CStreamBuf.newLine;
begin
  if eof(f) then newLine := finch
  else begin
    readln(f, buf);
    bufPos := 0; bufLen := length(buf);
    newLine := ' '
  end;
end;


function CStreamBuf.get;
begin
  bufPos := bufPos+1;
  if bufPos <= bufLen
    then get := buf[bufPos]
  else {строка в буфере исчерпана} get := newLine
end;
```

После определения класса CStreamBuf объекты этого класса можно использовать так же, как объекты класса CStream — *изменилась реализация операций, а не их набор или форма использования*.

□

В языке Турбо Паскаль класс может иметь только одного отца. Это называется *простым наследованием*. В некоторых языках, например C++ или Eiffel, допускается *множественное наследование*, при котором классов-отцов может быть несколько. В языке Java множественное наследование реализуется с помощью *множественного наследования интерфейсов*.

Класс может иметь произвольное количество классов-потомков; они сами могут быть отцами для других классов. Класс может наследовать компоненты (поля или методы) от какого-нибудь предка — своего отца, отца своего отца и т.д. Итак, классы могут образовывать *древовидную иерархию*.

- Наследование облегчает создание новых классов и увеличивает возможности повторного использования кода. Оно позволяет эффективно строить и использовать иерархии классов, которые абстрагированно представляют иерархии объектов реального мира, обычные для человека.

Используя наследование, помните о таких его свойствах.

✧ Переобъявлять атрибуты класса-отца в классе-потомке запрещено.

- ✧ Одноименные методы, объявленные в классе-отце и в классе-потомке, могут иметь различные наборы параметров и типы возвращаемых значений. Такое “изменение поведения” возможно, но, как правило, нежелательно, поскольку изменяет интерфейс класса.
- ✧ Родительский класс *совместим по присваиванию* с классом-потомком. Присваивание объекту родительского класса означает копирование только полей, общих для обоих классов (см. также замечание к листингу 15.1). Класс-потомок *не* совместим по присваиванию с классом-отцом, поскольку собственные поля объектов класса-потомка при таком присваивании остаются неопределенными.

15.2.2. Виртуальные подпрограммы

Пример 15.6. Во многих задачах обработки текстов нужно помнить номер строки, из которой берется текущий символ. Например, синтаксический анализатор находит ошибку в Паскаль-программе и указывает на нее с помощью номера строки и позиции в ней.

К внутреннему состоянию потока символов добавим номер строки `lineNum`, из которой взят текущий символ. Для получения номера строки объявим новую операцию `getLineNum`. Инициализировать номер строки естественно в методе `init`, а увеличивать — в функции `newLine`, поэтому их придется изменить.

Напишем объявление нового класса `CStreamLn`, производного от класса `CStream`, добавив его к модулю `uCStream`. В первом приближении оно может выглядеть так (его и объявление класса `CStream` все-таки ***придется изменить***).

```
type CStreamLn = object(CStream)
public
  procedure init(fileName : string);
  function getLineNum : integer;

private
  lineNum : integer; {номер текущей строки текста}
  function newLine : char;
end;
```

Методы `get` и `done` в новом классе `CStreamLn` унаследованы от класса-отца `CStream`. Добавим в раздел реализации модуля `uCStream` новые и измененные подпрограммы.

```
procedure CStreamLn.init; {инициализация}
begin
  lineNum := 0;
  CStream.init(fileName);
end;

function CStreamLn.getLineNum : integer;
begin getLineNum := lineNum end;

function CStreamLn.newLine;
  var c : char;
begin
  {конец строки обрабатывается, как в классе-отце}
  newLine := CStream.newLine;
  lineNum := lineNum+1; {добавлено увеличение номера строки}
end;
```

Нам нужно, чтобы при выполнении родительского метода `get` вызывалась функция `newLine` из того класса `CStream` или `CStreamLn`, для объекта которого вызван метод `get`. Решим эту проблему с помощью *виртуальных подпрограмм* — после заголовка такой подпрограммы в объявлениях класса-отца и всех его потомков ставится зарезервированное слово **`virtual`**.

- ***Внимание! Новые объявления не обеспечивают правильной обработки*** поля `lineNum`. Дело в том, что метод `get` в новом классе наследуется от класса `CStream`. При выполнении метода `get` вызывается функция `newLine`, объявленная в родительском классе, которая вообще не имеет доступа к полю `lineNum` (полю класса-потомка).

В объявлениях классов `CStream` и `CStreamln` добавим слово **`virtual`** после обоих заголовков функции `newLine`.

```
function newLine : char; virtual;
```

Благодаря этому слову при выполнении `get` вызывается метод `newLine` того класса `CStream` или `CStreamln`, для объекта которого вызван метод `get`.

- Если класс имеет виртуальные методы или подпрограммы, то хотя бы один из его методов нужно объявить или унаследовать как *конструктор*.

Конструктор выполняет действия, заданные в его теле, и, в отличие от других методов, формирует в объекте информацию, необходимую для вызовов виртуальных подпрограмм (подробнее см. подраздел 15.2.4).

Синтаксически конструктор — это обычная процедура, только вместо слова **`procedure`** записывают слово **`constructor`**. Для классов `CStream` и `CStreamLn` конструкторами естественно объявить методы `init`, изменив их заголовки в объявлениях классов.

```
constructor init(fileName : string);
```

После этих изменений в объявлениях классов `CStream` и `CStreamLn` и перекомпиляции модуля `uCStream` используем новый класс в программе, аналогичной программе в листинге 15.2. Объявим потоки `s1` и `s2` как объекты класса `CStreamLn` и перед завершением выведем количества строк в этих потоках (значения их полей `lineNum`).

Листинг 15.3. Использование объектов класса `CStreamLn`

```
program StreamsComparison;
  uses uCStreams;
  var s1, s2 : CStream;
      cmp    : CSComparator; {как в подразделе 15.1.2}
begin
  cmp.init(s1, s2);
  writeln(cmp.compareStreams(s1, s2));
  writeln('количества строк в потоках : ',
          s1.getLineNum, ' ', s2.getLineNum);
  cmp.done(s1, s2);
end.
```

Если при выполнении этой программы связать новые потоки с текстами, содержащими символы `q w e`<***eol***> и `q`<***eol***>`w`<***eol***>`e`<***eol***>, то будет напечатано `true`, а затем `количества строк в потоках : 2 4`. В первом объекте функция `newLine` вызывается дважды, во втором — четырежды.

□

- ✧ Одноименные виртуальные методы или подпрограммы в классе-отце и всех его потомках должны иметь одинаковые наборы параметров и типы возвращаемых значений.
- ✧ Конструктор объекта должен вызываться раньше, чем его виртуальные методы и подпрограммы.
- ✧ Конструктор не может быть виртуальным.
- ✧ Класс может иметь несколько конструкторов, но с различными именами.

15.2.3. Понятие полиморфизма

В интерфейсах классов `CStream` и `CStreamLn` объявлен метод `init`, реализованный в этих классах по-разному. Одна и та же сущность операции `init` (инициализация объекта) имеет различные формы, поэтому эта операция называется *полиморфной* (дословно — "многоформной").

По-разному в этих классах объявлена и скрытая функция `newLine`, которая также полиморфна. Метод `get` класса `CStreamLn` наследуется от `CStream`, однако для объектов этих разных классов он выполняется по-разному благодаря виртуальности `newLine`, поэтому тоже является полиморфным.

> Существование различных реализаций одной операции для объектов различных типов называется *полиморфизмом*.

Полиморфизм операций, возникающий благодаря наследованию и переобъявлению подпрограмм в классах-потомках, называется *включающим*. Другой вид полиморфизма, *параметрический*, связан с параметризованным объявлением типов. Он отсутствует в языке Турбо Паскаль, но реализован, например, в C++ или Java. Еще один вид полиморфизма, *специальный*, обеспечивается системой программирования — операция может быть определена для разных стандартных типов языка. Например, операция сложения со знаком + применяется к числам, строкам или множествам, но выполняется для них по-разному.

- Полиморфизм позволяет использовать общий интерфейс различных классов, не задумываясь над различиями в реализации операций, скрытыми в классах. Это способствует созданию ясного, короткого и гибкого кода.

Использование полиморфных методов будет представлено в разд. 15.3.

15.2.4. Динамическое связывание

Рассмотрим механизм, который во время выполнения программы обеспечивает вызовы нужных виртуальных подпрограмм (см. подразд. 15.2.2).

Транслятор создает код подпрограммы, превращая ее имя в ссылку на начало кода. Подстановка этой ссылки на место вызова подпрограммы называется *разрешением ссылки* или *связыванием*. Связывание выполняет специальная программа — *редактор связей* (*компоновщик*).

В классах `CStream` и `CStreamLn` метод `get` вызывает функцию `newLine`. Однако `get` должен вызывать различные функции в зависимости от класса объекта, для которого вызван сам `get`. Поэтому во время трансляции метода `get` нельзя определить, какая из функций должна быть вызвана, и подставить в код `get` соответствующую ссылку. Определение необходимо отложить до выполнения программы — тогда известно, для объекта какого класса осуществляется вызов `get`.

> Связывание, выполняемое компоновщиком, называется *статическим* (*ранним*), а связывание в процессе выполнения программы — *динамическим* (*поздним*).

Рассмотрим реализацию динамического связывания с виртуальными методами и подпрограммами. Если объект содержит виртуальные методы или подпрограммы, то для него в процессе трансляции создается *таблица ссылок на виртуальные методы* (*ТВМ*). Ее элементы хранят ссылки на виртуальные методы и подпрограммы класса, к которому относится объект.

Таблицу заполняет конструктор объекта *во время выполнения программы*, когда известен класс объекта, а значит, и ссылки на "его" методы и подпрограммы. Например, для объектов типов `CStream` и `CStreamLn` конструкторами являются методы `init`, вызываемые первыми.

Если вызывается виртуальный метод или подпрограмма объекта, то ссылка на него или на нее берется из ТВМ этого объекта. Например, при выполнении вызова `get` для объекта класса `CStream` или `CStreamLn` ссылка на "его" функцию `newLine` уже определена в таблице объекта. Эта ссылка берется из таблицы, т.е. связывание `get` из `newLine` происходит *во время выполнения программы* (*динамически*).

- Динамическое связывание позволяет вызывать одноименные виртуальные методы, определенные по-разному в разных классах. Оно также позволяет методам, определенным только в родительском классе, обращаться к подпрограммам как своего класса, так и классов-потомков. Таким образом, динамическое связывание является способом реализации полиморфизма операций в классах.

15.2.5. Объекты в свободной памяти

В подразд. 15.1.4 уже рассматривалось создание объектов в свободной памяти.[32] Продолжая пример с потоками символов, предположим, что в области видимости имени класса `CStream` объявлена переменная-указатель `sPtr`.

```
var sPtr : ^CStream;
```

При выполнении вызова `new(sPtr)` создается объект, на который ссылается `sPtr`. Если объект имеет виртуальные методы, то сразу после создания объекта применяется конструктор: `sPtr^.init(...)`.

Создание объекта и применение к нему конструктора можно объединить в одном вызове `new`. Первый аргумент в вызове указывает на объект, а второй является вызовом конструктора.

```
new(sPtr, init(...))
```

Память, ранее выделенная объекту с помощью процедуры `new`, освобождается — `dispose(sPtr)`. В вызовах процедуры `dispose` используют *методы-деструкторы*, в определенном смысле симметричные конструкторам. Они выполняют действия, завершающие работу с объектом. Например, для объектов типа `CStream` в качестве деструктора естественно взять процедуру `done`, закрывающую входной файл. Деструкторы записывают, как обычные процедуры, только вместо слова **`procedure`** ставят **`destructor`**. Деструкторы могут наследоваться и быть виртуальными; в одном классе их может быть несколько.

Деструктор часто используют при освобождении памяти, занятой объектом после вызова `new`. Например, пусть указатель `sPtr` установлен на объект в куче. Тогда применение к нему деструктора `done` и освобождение из-под него памяти задается таким вызовом.

```
dispose(sPtr, done)
```

15.3. Несколько приемов ООП

15.3.1. Абстрактный класс студентов и его потомки

Рассмотрим создание класса и его потомков с полиморфными операциями, которые реализованы виртуальными методами. Использование такой иерархии классов позволит, в частности, создавать агрегаты, компоненты которых могут быть *разнотипными объектами*.

---

[32] *В языках C++ и Java переменные объектных типов являются указателями на объекты, которые размещаются* только в куче.

Пример 15.7. Предположим, что данные о студенте состоят из фамилии, оценки, признака пола и года рождения (для юношей) или цвета глаз (для девушек). Конечно, реальные данные намного сложнее, но для упрощения изложения ограничимся этими. Предположим, что данные о студенте поступают с клавиатуры и выводятся на экран.

Представим данные о студентах с помощью объектов. Общие атрибуты для юношей и девушек — это фамилия и оценка, а собственные — год рождения и цвет глаз. Можно объявить родительский класс студентов с общими атрибутами и два потомка, в каждом из которых добавлен собственный атрибут (год или цвет). Однако объявим класс-отец как *абстрактный* — он *не имеет атрибутов*, *фиксирует набор операций и их описание, но не определяет их*.

Операции класса-отца должны быть реализованы в его потомках по-разному, т.е. должны быть полиморфными, поэтому объявим их виртуальными, а конструктором будет операция инициализации `init`.

```
type Student = object {абстрактный класс студентов}
     public
       constructor init(data : StudData);
                   {инициализация данными}
       procedure sendData(var data : StudData); virtual;
                 {операция передачи данных}
       destructor done; virtual; {полиморфная операция завершения}
     end;
```

В этом объявлении используется имя `StudData` — имя типа структур, которые *передаются объекту* для инициализации и *получаются от него*. Уточним тип `StudData`, учитывая данные о юношах и девушках.

```
type StudData = record
  {тип для обмена данными с объектами классов-потомков класса Student}
       surname : string;          {фамилия}
       mark    : byte;            {оценка}
       case sex : char of         {признак пола}
         'm' : (year : integer); {юноша представлен годом рождения}
         'f' : (colour : string) {девушка - цветом глаз}
     end;
```

Объявим типы структур `StudBoyData` и `StudGirlData` для представления юношей и девушек в объектах классов-потомков класса `Student`.

```
type
StudBoyData = record {тип для юношей}
  surname : string;
  mark    : byte;
  year     : integer;
end;
StudGirlData = record {тип для девушек}
  surname : string;
  mark    : byte;
  colour   : string;
end;
```

В объектах классов, представляющих юношей и девушек, объявим скрытые указатели `pData` на структуры данных типов соответственно `StudBoyData` и `StudGirlData`. Вместо явного атрибута "пол" в объектах классов `StudBoy` и `StudGirl` будет использована скрытая операция возвращения пола объекта `getSex`.

```
type
StudBoy = object (Student)
private
  pData           : ^StudBoyData;
```

```
  function getSex : char;
public
  constructor init(data : StudData);
  procedure   sendData(var data : StudData); virtual;
  destructor  done; virtual;
end;
StudGirl = object (Student)
private
  pData           : ^StudGirlData;
  function getSex : char;
public
  constructor init(data : StudData);
  procedure   sendData(var data : StudData); virtual;
  destructor  done; virtual;
end;
```

- В отличие от записей с вариантами (см. раздел 9.5), эти классы защищают от ошибочного использования полей, поскольку нужный набор атрибутов задан в объявлении класса. Например, попытка использовать поле `colour` в объекте класса `StudBoy` вообще невозможна, поскольку ее обнаружит компилятор.

Все приведенные объявления соберем в раздел интерфейса модуля `uStudent`; добавим также типы указателей на объекты объявленных классов.

```
pStudent  = ^Student;
pStudBoy  = ^StudBoy;
pStudGirl = ^StudGirl;
```

В разделе реализации модуля запишем методы и вспомогательные функции объявленных выше классов. Методы абстрактного класса `Student` никак не уточняются, но компилятор Турбо Паскаль требует их наличия, поэтому определим их с пустым телом. Приведем только подпрограммы для класса `StudBoy` — для класса `StudGirl` они аналогичны. Запишите их самостоятельно.

```
constructor Student.init; begin end;
procedure   Student.sendData; begin end;
destructor  Student.done; begin end;

constructor StudBoy.init;
begin
  getmem(pData, sizeof(StudBoyData));
  with pData^ do begin
    surname := data.surname;
    mark := data.mark;
    year := data.year
  end;
end;
procedure StudBoy.sendData;
begin
  with pData^ do begin
    data.surname := surname;
    data.mark    := mark;
    data.sex     := getSex;
    data.year    := year
  end
```

```
end;
destructor StudBoy.done;
begin freemem(pData, sizeof(StudBoyData)); end;
function StudBoy.getSex;
begin getSex := 'm' end;
```

Запишите весь модуль самостоятельно. Объявления и определения из него будут использованы в следующих подразделах.
□

15.3.2. Отделение внешнего представления

Один из принципов ООП — *отделение поведения объектов от их внешнего представления*. По этому принципу *объект не отвечает за ввод и вывод своих данных* — это "поручается" специальным объектам, которые обмениваются данными с основным объектом. Применение этого принципа позволяет *изменять интерфейс и поведение объектов по отдельности и взаимно независимо*.

Например, в классе `Student` (см. подраздел 15.3.1) есть только процедуры `init` и `sendData` — инициализации на основе данных о студенте и их передачи. Процедуры ввода данных о студенте с внешних носителей и их вывод определяются как методы одного или нескольких специальных классов.

Создадим класс `StudInterface`, обеспечивающий ввод с клавиатуры данных о студенте в объекты классов-потомков класса `Student` и вывод данных на экран. Его операция `readData` должна считывать данные о студенте, в зависимости от пола студента создавать и инициализировать объект класса `StudBoy` или `StudGirl` и возвращать ссылку на этот объект. Типом значений, возвращаемых из операции `readData`, является *тип ссылок на объекты класса-отца* (в данном случае класса `Student`). Операция `writeData` должна получать как аргумент ссылку на объект одного из классов-потомков, получать от него данные и выводить их во "внешний мир".

```
type
StudInterface = object
public {операции ввода и вывода данных о студенте}
  function  readData : pStudent;
  procedure writeData(pStud : pStudent);
end;
```

Добавим это объявление к интерфейсу модуля `uStudent`. В разделе реализации запишем методы нового класса, которые в простейшем случае могут быть такими.

```
{метод ввода данных о студенте с клавиатуры}
function StudInterface. readData : pStudent;
  var data : StudData; {структура для данных о студенте}
      p    : Pointer;  {указатель на создаваемый объект}
begin
  with data do begin
    write('Фамилия>'); readln(surname);
    write('Оценка>'); readln(mark);
    write('Пол (m - юноша /f - девушка)>'); readln(sex);
    if sex = 'm' then begin
      write('Год рождения>'); readln(year);
      getmem(p, sizeof(StudBoy));
      pStudBoy(p)^.init(data);
      readData := pStudBoy(p);
    end
    else
    if sex = 'f' then begin
      write('Цвет глаз>'); readln(colour);
      getmem(p, sizeof(StudGirl));
```

```
      pStudGirl(p)^.init(data);
      readData := pStudGirl(p);
    end
    else readData := pStudent(nil);
    {возвращается ссылка "на ничто", если пол задан с ошибкой}
  end; {with}
end;
{метод вывода данных о студенте на экран}
procedure StudInterface.writeData;
  var data : StudData;
begin
  pStud^.sendData(data);
  with data do begin
    write(surname, ' ', mark, ' ', sex, ' ');
    if sex = 'm' then writeln(year)
    else writeln(colour)
  end;
end;
```

- Отделение внешнего представления объектов (операций ввода-вывода) от их поведения позволяет не перегружать класс и, если необходимо, изменять реализацию поведения объектов независимо от реализации внешнего представления и наоборот.

15.3.3. Массив ссылок на объекты различных классов

Рассмотрим на конкретном примере, как наследование и полиморфные операции позволяют объединять и обрабатывать разнотипные объекты в одной структуре.

Пример 15.8. Из студентов формируют академические группы. Предположим, что данные о группе должны храниться как *последовательность* данных о ее студентах и выводиться на экран.

Простейший (хотя не всегда наилучший) способ реализовать последовательность — использовать массив. Массив должен состоять из объектов *различных типов* `StudBoy` и `StudGirl` (см. предыдущие подразделы), поэтому *непонятно, каким должен быть тип его элементов*.

Для решения этой проблемы применим наследование и полиморфизм. Создадим класс `StudGroup` ("группа студентов"), атрибутами которого являются массив *указателей на объекты класса-отца* `Student` и количество студентов, представленных в массиве.

Вначале уточним операции класса `StudGroup`. Кроме операций инициализации `init` и завершения `done`, объявим такие:

`append` — добавление студента к группе,

`remove` — удаление студента, последнего в последовательности,

`isComplete` — проверка, заполнена ли группа,

`isEmpty` — проверка, пуста ли группа.

Конечно, реальные наборы операций с агрегатами, как правило, сложнее; приведенный набор выбран, чтобы не отвлекать внимание на несущественные детали.

Операции добавления и удаления объявим как функции, возвращающие признак, успешно ли выполнена операция. Итак, запишем в интерфейсный раздел модуля `uStudGr`, использующего модуль `uStudent`, следующие объявления.

```
const maxSize = 30; {максимальный размер группы}
type
  Index         = 1..maxSize; {индексное множество массива}
```

```
  ExtendedIndex = 0..maxSize+1; {расширенное индексное множество}
  apStudent  = array [Index] of pStudent; {тип массива указателей}
  StudGroup = object {класс "группа студентов"}
  private
    students  : apStudent; {массив указателей на объекты}
    groupSize : ExtendedIndex; {текущее количество студентов в группе}
  public
    constructor init; {инициализация}
    function    append(pSt : pStudent) : boolean; {добавление}
    function    remove : boolean ; {удаление последнего}
    function    isComplete : boolean; {проверка,  заполнена ли группа}
    function isEmpty : boolean; {проверка, пуста ли группа}
    destructor done; {завершение работы с группой}
  end;
  pStudGroup = ^StudGroup; {тип указателей на группы}
```

В разделе реализации модуля `uStudGr` запишем методы, которые реализуют приведенные операции класса `StudGroup`. Комментарии к ним записаны в коде.

```
constuctor StudGroup.init;
 {создается пустая группа, т.е. указатели в массиве}
 {инициализируются ссылками "на ничто"}
  var curr : Index;
begin
  for curr := 1 to maxSize do students[curr] := nil;
  groupSize := 0
end;

function StudGroup.append;
{ссылка на новый объект добавляется в конце последовательности;}
{возвращается признак того, что элемент был добавлен}
begin
  append := true;
  if (pSt = nil) or isComplete then begin
    append := false; exit
  end;
  inc(groupSize);
  students[groupSize] := pSt;
end;

function StudGroup.remove;
{удаляется последний элемент; память от объекта освобождается;}
{возвращается признак того, что элемент был удален}
begin
  remove := true;
  if isEmpty then begin remove := false; exit end;
  students[groupSize]^.done;
  dispose(students[groupSize]); students[groupSize] := nil;
  dec(groupSize);
end;

function StudGroup.isComplete;
{возвращается признак того, что массив заполнен}
begin isComplete := groupSize = maxSize end;

function StudGroup.isEmpty;
{возвращается признак того, что массив представляет пустую группу}
begin isEmpty := groupSize = 0 end;
```

```
destructor StudGroup.done;
{при завершении обработки группы удаляются все объекты-студенты}
begin while remove do; end;
```

Напишем программу создания объекта класса "группа студентов", заполнения его данными о студентах и вывода данных (листинг 15.4).

Листинг 15.4. Программа создания и печати данных о студентах группы

```
program useStudents;
  uses uStudent, uStudGr;
  var pStudGr : pStudGroup; {группа создается в свободной памяти}
  {процедура формирования массива указателей на студентов}
  procedure createGroup(var group : StudGroup);
    {объект для ввода и вывода данных о студенте}
    var studInterf : StudInterface;
    const c : char = 'y'; {признак продолжения ввода}
  begin
    group.init;
    writeln('Ввод данных о студентах группы');
    while true do begin
      writeln('Введите данные о студенте');
      if not group.append(studInterf.readData)
        then writeln('Данные о студенте не добавлены');
      if group.isComplete then begin
        writeln('Группа заполнена'); break;
      end;
      writeln('Продолжать? (y/n)'); readln(c);
      if (c <> 'y') then break;
    end;
  end;
  {процедура вывода данных о студентах группы}
  procedure printGroup(const group : StudGroup);
    var studInterf : StudInterface;
        i : ExtendedIndex;
  begin
    writeln('Вывод данных о студентах группы');
    for i := 1 to group.groupSize do
      studInterf.writeData(group.students[i]);
  end;

begin
  new(pStudGr, init);     {создание группы}
  createGroup(pStudGr^); {заполнение группы данными о студентах}
  printGroup(pStudGr^);  {вывод данных}
  pStudGr^.done;         {завершение обработки группы}
end.
```

Процедуры `createGroup` и `printGroup` фактически реализуют операции ввода и вывода данных о группе. Их можно сделать методами специального класса, аналогичного классу `StudInterface` (см. предыдущий подраздел). Определите и используйте этот класс самостоятельно.

□

15.3.4. Знакомство с контейнерами и итераторами

Данные в объектах многих классов организуются в виде записей, массивов, списков или других структур, образованных объектами других классов или ссылками на них. Классы такого рода называют *контейнерами*. Типичным примером контейнера является класс `StudGroup` (см. предыдущий подраздел) — его объекты содержат массив указателей на объекты других классов.

Элементы объекта-контейнера, как правило, обрабатываются по одному в определенном порядке. Например, в процедуре вывода данных о студентах группы `printGroup` к объектам-студентам применяется операция `sendData` в соответствии с расположением указателей на них в массиве (*итеративно*).

Работа с контейнером может требовать нескольких различных порядков доступа к его элементам, например данные о студентах группы могут выводиться с начала массива указателей, с его конца, в алфавитном порядке фамилий, по убыванию оценок и т.д. Также могут обрабатываться только данные, удовлетворяюшие некоторому условию, например, только данные о юношах. Один из принципов ООП — за порядок доступа к элементам отвечает не сам объект-контейнер, а *отдельный объект*.

> *Итератором* называется класс, который определяет интерфейс обхода объектов класса-контейнера. Объект класса-итератора также называется итератором.

- Класс-итератор *упрощает интерфейс* класса-контейнера, забирая у него операции, связанные с обходом.
- Использование итераторов позволяет *изменять* и *добавлять* новые способы обхода, *не* изменяя контейнер.
- Итераторы включают состояния обхода и позволяют для одного контейнера выполнять одновременно *несколько* обходов.

Реализовать итератор можно по-разному. В частности, различают *внешние* и *внутренние* итераторы. Если обходом контейнера руководит клиент итератора с помощью операций итератора, то итератор называется *внешним* (операции применяются вне итератора), а если сам итератор — *внутренним*.

Внутренние итераторы, в отличие от внешних, сами определяют алгоритм обхода контейнера, поэтому их проще использовать, чем внешние. По этой же причине внешние итераторы позволяют клиентам более гибко руководить обходом. Конкретные внутренний и внешний итераторы рассмотрим в двух примерах.

Пример 15.9. Процедура `printGroup` (см. листинг 15.4) сама определяет первый элемент для обработки, способ перехода к следующему элементу и условия завершения итерации. В модуле `uStudGr` объявим *внутренний итератор*, метод итерации которого будет похож именно на эту процедуру. Объекты этого итератора содержат ссылки на контейнер, к которому применяется итерация, и переменную, значениями которой являются номера текущих элементов контейнера (*состояния обхода*). Ссылка на контейнер передается при инициализации итератора.

```
type
  StudGroupInnerIter = object
  private
    pGroup : pStudGroup;    {указатель на объект-контейнер}
    curr   : ExtendedIndex; {номер текущего элемента}
  public
    procedure init(p : pStudGroup); {инициализация итератора}
```

```
    procedure printGroup; {итерация}
    procedure done; {завершение работы с итератором}
  end;
```

В разделе реализации запишем методы нового класса.

```
procedure StudGroupInnerIter.init;
begin pGroup := p; curr := 1 end;
procedure StudGroupInnerIter.printGroup;
  var studInterf : StudInterface;
      i : ExtendedIndex;
begin
  for i := 1 to pGroup^.groupSize do begin
    curr := i;
    studInterf.writeData(pGroup^.students[curr]);
  end
end;
procedure StudGroupInnerIter.done;
begin pGroup := nil; curr := 0 end;
```

С использованием нового класса программа из примера 15.8 (см. листинг 15.4) приобретает следующий вид (листинг 15.5).

Листинг 15.5. Программа с внутренним итератором

```
program useStudents_InnIter;
  uses uStudent, uStudGr;
  var pStudGr     : ^StudGroup;
      stGrInnIter : StudGroupInnerIter;
  {процедура формирования массива указателей на студентов}
  procedure createGroup(var group : StudGroup);
    ... {см. листинг 15.4}
  end;
begin
  new(pStudGr); pStudGr^.init;
  createGroup(pStudGr^);
  stGrInnIter.init(pStudGr);
  stGrInnIter.printGroup; {вызов метода итерации}
  stGrInnIter.done;
  pStudGr^.done;
end.
```

□

Пример 15.10. *Внешний итератор* обеспечивает операции управления итерацией. Рассмотрим их минимальный набор:

установка первого элемента,

переход к следующему элементу,

проверка условия завершения итерации,

возвращение текущего элемента.[33]

Объявим эти операции в новом классе-итераторе для класса-контейнера, представляющего группу студентов.

[33] *Операции перехода к следующему элементу, проверки условия завершения итерации и возвращения текущего элемента можно объединить — операция проверяет, есть ли следующий элемент, переходит к нему и возвращает его. Если следующего элемента нет, т.е. выполняется условие завершения итерации, операция возвращает заранее зафиксированное значение (признак завершения итерации).*

```
type
  StudGroupOuterIter = object
  private
    pGroup : pStudGroup;    {указатель на объект-контейнер}
    curr   : ExtendedIndex; {номер текущего элемента}
  public
    procedure init(p : pStudGroup); {инициализация итератора}
    procedure first;  {установка первого элемента}
    procedure next; {переход к следующему элементу}
    function  isDone : boolean; {проверка условия завершения итерации}
    function currentItem : pStudent; {возвращение ссылки}
      {на текущий элемент контейнера}
    procedure done; {завершение работы с итератором}
  end;
```

В разделе реализации модуля запишем методы нового класса.

```
procedure StudGroupOuterIter.init(p : pStudGroup);
begin pGroup := p; curr := 1 end;
procedure StudGroupOuterIter.first;
begin curr := 1 end;
procedure StudGroupOuterIter.next;
begin if not isDone then inc(curr) end;
function StudGroupOuterIter.isDone : boolean;
begin isDone := curr > pGroup^.groupSize end;
function StudGroupOuterIter.currentItem : pStudent;
begin currentItem := pGroup^.students[curr] end;
procedure StudGroupOuterIter.done;
begin pGroup := nil; curr := 0 end;
```

Используя этот класс, модифицируем программу из листинга 15.4. Новая процедура печати `printGroup` управляет итерацией с помощью методов нового класса-итератора, который вместе с контейнером становится ее параметром (листинг 15.6).

Листинг 15.6. Программа создания группы с внешним итератором

```
program useStudents_OutIter;
  uses uStudent, uStudGr;
  var pStudGr     : ^StudGroup;
      stGrOutIter : StudGroupOuterIter;
  {процедура формирования массива указателей на студентов}
  procedure createGroup(var group : StudGroup);
    ... {см. листинг 15.4}
  end;
  {процедура вывода данных о студентах группы}
  procedure printGroup(pGroup : pStudGroup; outIter : StudGroupOuterIter);
    var studInterf : StudInterface;
  begin
    outIter.init(pGroup); outIter.first;
    while not outIter.isDone do begin
      studInterf.writeData(outIter.currentItem);
      outIter.next
    end;
  end;
begin
  new(pStudGr); pStudGr^.init;
  createGroup(pStudGr^);
```

```
  printGroup(pStudGr, stGrOutIter);
  stGrOutIter.done;
  pStudGr^.done;
end.
```

Параметризация процедуры `printGroup` итератором позволяет применять ее для решения других задач. Например, для печати данных только о студентах-юношах достаточно написать соответствующий клас-итератор и указать объект этого класса как аргумент в ее вызове. Сделайте это самостоятельно.

□

15.3.5. Обозначение объекта в его собственных методах

В методе объекта может понадобиться явно обозначить сам объект.

Пример 15.11. Продолжая примеры с классами `StudGroup` (группа студентов) и `StudGroupInnerIter` (внутренний итератор группы), предположим, что объект-контейнер сам может создавать для себя объект-итератор. Для этого к интерфейсу класса `StudGroup` в модуле `uStudGr` добавим операцию создания итератора, представленную таким заголовком.

```
procedure createInnIter(var pGrInnIter : pStudGroupInnerIter);
```

Соответствующий метод должен создать объект-итератор и установить на него указатель, заданный в вызове как аргумент.

Объект-итератор имеет доступ к "своему" контейнеру с помощью атрибута-указателя `pGroup` — адрес контейнера передается ему как параметр метода `Init` итератора. Таким образом, в вызове `StudGroupInnerIter.Init`, записанном в методе `StudGroup.createInnIter`, *нужно обозначить именно тот объект, которому принадлежит этот метод*.

- Объект в своих собственных методах обозначается с помощью специального имени `self`.

С помощью имени `self` определим метод создания итератора в виде следующей процедуры, добавив ее к разделу реализации модуля `uStudGr`.

```
procedure StudGroup.createInnIter(var pGrInnIter :
pStudGroupInnerIter);
begin
  new(pGrInnIter, init(@self))
end;
```

Как видим, метод `init` итератора с помощью аргумента `@self` получает *ссылку на объект-контейнер*, создающий итератор.

Используем обновленный класс `uStudGr` в следующей программе (листинг 15.7).

Листинг 15.7. Контейнер создает внутренний итератор для себя

```
program useStudents_InnIter_by_Container;
  uses uStudent, uStudGr;
  var pStudGr     : ^StudGroup;
      pStudGrInnIter : StudGroupInnerIter;
  procedure createGroup(var group : StudGroup);
    ... {см. листинг 15.4}
  end;
begin
  new(pStudGr); pStudGr^.init;
```

```
  createGroup(pStudGr^);
  {контейнер сам создает внутренний итератор для себя:}
  pStudGr^.createInnIter(pStGrInnIter);
  stGrInnIter.printGroup;
  stGrInnIter.done;
  pStudGr^.done;
end.
```

□

15.3.6. Операция с элементами как параметр внутреннего итератора

В предыдущих подразделах был рассмотрен класс внутренних итераторов `StudGroupInnerIter` для контейнера класса `StudGroup`. Процедура итерации `printGroup` проходила по контейнеру и обрабатывала объекты, ссылки на которые содержал контейнер. Обработка состояла в применении метода `sendData` (он вызывался в методе `writeData` класса `studInterface`, отвечавшего за вывод данных о студенте).

Операцию, которую нужно применить к объектам контейнера во время его обхода, можно задать *как параметр метода внутренней итерации*.

Пример 15.12. Группа студентов представлена объектом-контейнером, как в предыдущих подразделах. Нужно сначала напечатать данные только о студентах (юношах), а потом только о студентках (девушках).

Рассмотрим одно из возможных решений этой задачи. Предположим, что операция, которую нужно применить к каждому объекту во время итерации, реализуется как процедура, параметром-значением является указатель на объекты типа `Student`. К модулю `uStudGr`, в котором определены классы контейнера и итераторов, добавим объявления имени типа таких процедур `tProc`.

```
type tProc = procedure (pstud : pStudent);
```

В объявлении класса внутренних итераторов `StudGroupInnerIter` к заголовку операции `printGroup` добавим параметр типа `tProc`.

```
procedure printGroup(op : tProc);
```

В разделе реализации модуля модифицируем метод итерации `printGroup`.

```
procedure StudGroupInnerIter.printGroup;
  var i : Index0;
begin
  for i := 1 to pGroup^.groupSize do begin
    curr := i; op(pGroup^.students[curr]);
  end
end;
```

Внесем приведенные изменения в модуль `uStudGr`.

В программе, похожей на программу в листинге 15.5, напишем процедуру `printMale` печати данных о студенте, если это юноша, а не девушка. На объект, представляющий студента, указывает параметр этой процедуры. Данные получаются методом объекта `sendData` и выводятся с помощью объекта класса `studInterface`.

```
{$F+}
  procedure printMale(pstud : pStudent);
    var data       : StudData;
        studInterf : StudInterface;
  begin
    pstud^.sendData(data);
```

```
    if data.sex = 'm'
      then studInterf.writeData(pstud);
  end;
 {$F-}
```

Процедура печати данных о студентке `printFemale` аналогична, только в условии разветвления вместо `'m'` будет записано `'f'`.

С использованием этих процедур и нового класса внутренних итераторов тело программы, решающей задачу, будет таким.

```
  new(pStudGr); pStudGr^.init;
  createGroup(pStudGr^);
  stGrInnIter.init(pStudGr);
  stGrInnIter.printGroup(printMale);   {для юношей}
  stGrInnIter.printGroup(printFemale); {для девушек}
  stGrInnIter.done;
  pStudGr^.done;
```

□

Резюме

- Программа, построенная на принципах ООП, образуется из *объектов* — отдельных фрагментов кода и данных, которые взаимодействуют друг с другом и окружающим миром. Тип объектов называется *классом*. Объекты — это переменные, компонентами которых являются поля данных и подпрограммы. Поля данных, как правило, скрываются внутри объектов; доступ к ним возможен только с помощью специальных подпрограмм (*методов*). Методы реализуют операции, образующие *интерфейс класса*.
- *Инкапсуляция* — это объединение данных и операций их обработки в целостную структурную единицу (объект), которая защищает данные от несанкционированных способов обработки. Инкапсуляция позволяет использовать операции объектов независимо от представления их состояний.
- *Наследование* — это использование ранее объявленного класса (*отца*) в объявлении нового класса (*потомка*). Класс может иметь любое число классов-потомков, которые сами могут быть отцами для других классов. Класс может наследовать компоненты (поля или подпрограммы) от какого-нибудь из предков (отца, отца своего отца и т.д.), т.е. классы могут образовывать *древовидную иерархию*.
- Инкапсуляция и наследование обеспечивают модульность, независимость реализации, повторное использование и естественность представления при моделировании объектов реального мира.
- Существование различных реализаций одной и той же операции для объектов различных типов называется *полиморфизмом*. Полиморфизм операций позволяет использовать один и тот же интерфейс для различных классов, не думая об отличиях реализации классов.
- Если объект имеет виртуальные методы или подпрограммы, для него в процессе трансляции создается *таблица ссылок на виртуальные методы* (*ТВМ*). Таблицу заполняет конструктор объекта во время выполнения программы, когда известен класс объекта, а следовательно, и ссылка на "его" методы и подпрограммы.
- Связывание в процессе выполнения программы называется *динамическим* (*поздним*). Когда вызывается виртуальный метод, ссылка на его код берется из ТВМ. Динамическое связывание позволяет вызывать одноименные виртуальные методы, определенные по-разному в различных классах. Оно также позволяет методам, определенным только в родительском классе, обращаться к подпрограммам как своего класса, так и классов-потомков.

- Отделение внешнего представления объектов (операций ввода-вывода) от их поведения позволяет не перегружать класс и, если необходимо, изменять реализацию поведения объектов независимо от реализации внешнего представления и наоборот.
- *Абстрактный класс* не имеет атрибутов, фиксирует набор операций и их описание, но не определяет их.
- *Контейнер* — это класс, объекты которого содержат несколько объектов других классов или ссылки на них.
- *Итератор* — это класс, который определяет интерфейс обхода объектов класса-контейнера. Объект класса-итератора также называется итератором. Клас-итератор упрощает интерфейс класса-контейнера, забирая у него операции, связанные с обходом. Использование итераторов позволяет изменять и добавлять новые способы обхода, не изменяя контейнер. Итераторы включают в себя состояния обхода и позволяют для одного контейнера выполнять одновременно несколько обходов.
- Объект в своих собственных методах обозначается с помощью специального имени `self`.

Контрольные вопросы

15.1. Почему классы более пригодны для реализации абстрактных типов данных, чем модули?

15.2. Чем инкапсуляция в объявлениях классов отличается от инкапсуляции в модулях?

15.3. Что такое интерфейс класса?

15.4. Можно ли в одном и том же классе объявить несколько различных, но одноименных конструкторов?

15.5. Какие виды полиморфизма реализованы в языке Турбо Паскаль?

15.6. Что такое позднее связывание? Как оно реализует полиморфизм операций?

15.7. Чем внешние итераторы отличаются от внутренних?

Задачи

15.1.* Недостатком буферизированного чтения символов текста (см. пример 15.5 в подразд. 15.2.1) является предположение об ограниченности длины строк текста. Определить класс буферизированного чтения символов текста, который не накладывает никаких ограничений на длину строк текста и за счет буфера позволяет возвращать в поток несколько символов, полученных из него ранее.

15.2. Дополнить класс из задачи 15.1, чтобы можно было получать номер строки в тексте, из которой взят текущий символ.

15.3. Защитить методы класса `CStream` (см. разд. 15.1) от возможных ошибок, связанных с обработкой файлов (см. разд. 10.1).

15.4. Параметризовать метод `CSComparator.init` (см. подразд. 15.1.3) внешними именами файлов; получение имен должна обеспечить программа перед вызовом этого метода.

15.5. Защитить класс `CSComparator` (см. разд. 15.1) от возможного некорректного использования, например попыток закрыть еще неоткрытые потоки.

15.6. Модифицировать операции ввода-вывода данных о студенте, чтобы они позволяли обрабатывать тексты, в каждой строке которых записана фамилия, оценка, пол и год или цвет, разделенные пробелами (см. подразд. 15.3.2).

15.7. Модифицировать класс `StudGroup` (см. подразд. 15.3.3), чтобы количество студентов группы определялось в процессе выполнения программы, а массив указателей с этим количеством элементов инициализировался в свободной памяти.

15.8. Модифицировать класс `StudGroup` (см. подразд. 15.3.3), используя связанный список вместо массива.

15.9. Добавить к модулю `uStudGr` (см. подразд. 15.3.3) класс, обеспечивающий ввод данных о студентах группы с клавиатуры или из текстового файла в объекты класса `StudGroup`, а также вывод данных на экран или в текстовый файл.

15.10. Модифицировать класс `StudGroup` (см. подразд. 15.3.3), чтобы выводить данные о студентах группы в алфавитном порядке. Использовать связанный список вместо массива.

15.11. Добавить к модулю `uStudGr` определение внешнего итератора, позволяющего обрабатывать только студентов-юношей (см. подразд. 15.3.4). Модифицировать программу из примера 15.10 для вывода данных о юношах.

15.12. Определить класс стеков символов (см. задачу 11.5) и с его помощью решить задачу:

***а)** 11.6;

б) 11.7.

15.13. Решить задачу 11.8, используя класс стеков целых чисел (см. задачу 11.5).

15.14. Определить класс очередей целых чисел (см. задачу 11.9) и с его помощью решить задачу 11.10.

15.15. Определить класс очередей, в обоих концах которых можно добавлять и удалять элементы. С использованием этого класса решить задачу 11.10.