

Министерство общего и профессионального образования Российской  
федерации

Ростовский ордена Трудового Красного Знамени  
государственный университет

А.А.Чекулаева, Я.М.Демяненко

**Основы  
объектно-ориентированного программирования  
в языке Паскаль**

Методические указания для студентов вечернего отделения  
механико-математического факультета

Ростов - на - Дону  
1999

Печатается по решению учебно-методической комиссии механико-математического факультета РГУ от 27 мая 1999г.

## АННОТАЦИЯ

Методические указания дают представление об объектно-ориентированном программировании, содержат определения основных понятий, рассматривают методы объектно-ориентированного программирования на примере языка Паскаль, приводятся программы реализации описываемых примеров.

Методические указания предназначены для студентов вечернего отделения механико-математического факультета РГУ и могут быть использованы как в контактных, так и в дистанционных формах обучения.

Авторы: А.А.Чекулаева, Я.М.Демяненко

## **Понятие объектно-ориентированного программирования**

Объектно-ориентированное программирование – это методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного класса, а классы образуют иерархию на принципах наследуемости. В частности, программирование не основанное на иерархических отношениях, относится не к ООП, а к программированию на основе абстрактных типов данных.

Можно сказать, что язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

- Имеется поддержка объектов в виде абстракции данных, имеющих интерфейсную часть в виде поименованных операций и защищенную область локальных данных.
- Объекты относятся к соответствующим типам (классам).
- Типы (классы) могут наследовать атрибуты от супертипов (суперклассов).

В объектном подходе акцент переносится на конкретные характеристики физической и абстрактной системы, являющейся предметом программного моделирования. Объекты обладают целостностью, которую не следует нарушать. Объект может только менять состояние, поведение, управляться или становиться в определенное отношение к другим объектам. Объект обладает неизменными качествами, но может изменить свое состояние. Стефик и Бобров утверждают, что объект можно рассматривать как нечто, объединяющее свойства процедур и данных в процессе выполнения вычислений и сохраняющее локальное состояние. Такое определение допустимо, но все же главным в понятии объекта является все же объединение идей абстрагирования данных и алгоритмов. Иными словами объект можно определить как осязаемую сущность, которая четко определяет свое поведение (Гради Буч).

Объектно-ориентированный подход представляет программы в виде набора объектов, взаимодействующих между собой. Взаимодействие объектов осуществляется через сообщения. Когда мы говорим, что объекту передается сообщение, то на самом деле мы вызываем некоторую процедуру (функцию) этого объекта (метод).

Базовыми блоками объектно-ориентированной программы являются объекты и классы. Класс – это множество объектов, имеющих общую структуру и общее поведение. Если объект – это что-то осязаемое, то класс – только абстракция, используемая для описания общей структуры и поведения множества объектов.

К базовым принципам объектно-ориентированного стиля программирования относятся:

1. Пакетирование (encapsulation).
2. Наследование (inheritance).
3. Полиморфизм (polymorphism).

#### 4. Передача сообщений.

Пакетирование предполагает соединение в одном объекте данных и методов, которые манипулируют этими данными.

Наследование позволяет использовать библиотеки классов и развивать их (совершенствовать и модифицировать библиотечные классы в конкретной программе).

Полиморфизм позволяет использовать одни и те же методы для решения разных задач. Виртуальные (модифицируемые) методы являются примером реализации этого принципа.

Передача сообщений выражает основную методологию построения объектно-ориентированных программ.

Теперь дадим более точное определение. Объектно-ориентированное программирование – это метод построения программ в виде множества взаимодействующих объектов, структура и поведение которых описаны соответствующими классами, и все эти классы являются компонентами иерархии классов, выражающей отношение наследуемости. Базовым абстрактным типом объектно-ориентированной программы является класс. Если класс D наследует структуру и поведение класса B, то класс B – базовый (base), суперкласс (superclass), родитель (parent), предшественник (predecessor); класс D – производный (derive), подкласс (subclass), ребенок (child), потомок (descendant).

Когда мы объявляем класс, мы вводим новый абстрактный тип данных (описываем компоненты данные и объявляем и описываем его компоненты-функции). Объявление метода-функции (метода-процедуры) содержит описание имени функции, типов ее параметров и типа возвращаемого значения. Описание функции задает имена параметров и тело (код) функции.

Когда мы описываем объекты, мы хотим выделить память, структура которой соответствует описанию класса, и, возможно, выполнит инициализацию. Поле-данные может быть как переменной, так и указателем. Во втором случае, в процессе инициализации можно динамически выделить память и назначить адрес ее начала соответствующему указателю. Значения некоторых полей-данных могут оставаться неопределенными. Описание объекта задает его тип (т.е. имя класса) и, возможно, некоторые параметры, которые необходимы для инициализации.

При построении объекта вызывается специальный метод класса, называемый конструктором. Конструктор управляет выделением памяти для объекта, строит объект в памяти и, возможно инициализирует его поля-данные. Наряду с конструктором существует другой метод класса, который называется деструктором. Деструктор разрушает объект, т.е. удаляет его из памяти.

Объявление класса состоит из двух частей: интерфейса и реализации. Интерфейс (interface) показывает, как мы можем использовать класс. Реализация (implementation) – внутренняя особенность класса. Они должны быть максимально независимы друг от друга.

Все данные делятся на локальные и глобальные. Изменение локальных данных и получение их значений можно осуществлять только в методах и полях-данных этого же объекта (существуют некоторые исключения). Глобальные данные можно менять и получать их значения извне.

Существует несколько разновидностей защиты данных и методов:

- Private – недоступен за пределами класса.
- Protected – доступен в производном классе.
- Public – доступен везде.

Заметим, что перегрузке методов меняется реализация, но сохраняется интерфейс.

При написании программ очень важным является выявление и устранение ошибок. Для этих целей в объектно-ориентированных программах используется специальный метод, называемый обработкой исключений (exception handling), частным случаем которого является обработка ошибок (error handling). Опишем идею этого метода:

- Пусть функция F посылает сообщение объекту O.
- Если объект O обнаруживает в сообщении ошибку, неоднозначность или то, невозможно выполнить, он прерывает свои функции и генерирует исключение.
- Функция F представляется в виде двух частей: собственно функции обработчика исключения. Следствием генерации исключения в объекте является активизация обработчика исключения.
- Обработчик исключения делает вывод о возможности продолжения программы (изменяет ошибочное сообщение) или о необходимости ее завершения по ошибке.

### Определение объектов

Объект – это структура, компонентами которой являются данные-поля объекта и процедуры (функции) - методы объекта. Все данные и методы *инкапсулированы* в одном объекте.

Определение типа ОБЪЕКТ:

```
TYPE ИмяОбъекта = ОБЪЕКТ  
                Поля данных;  
                Заголовки методов;  
                END.
```

Все поля предшествуют методам. Поля – любые структуры данных кроме файлов.

В определении типа Объект его методы представляются своими заголовками. Описание методов приводится после определения типа Объект так же, как это делается для подпрограмм в модулях. В заголовке процедуры или функции при описании метода имя метода дополняется через точку именем типа объекта, к которому относится данный метод. Это нужно, так как несколько разных методов, относящихся к разным объектам, могут иметь одно и то же имя. При этом в заголовке метода можно не включать список формальных параметров. Наряду с обычными параметрами в

описании методов можно использовать и поля объектов. Все данные объекта становятся автоматически глобальными по отношению к методам. Как правило, к данным объекта извне непосредственно не обращаются. Имена формальных параметров метода и имена полей данных этого объекта не должны совпадать.

В качестве методов может использоваться также особый вид процедур: конструкторы и деструкторы. Если в объекте нет виртуальных методов, то в нём может не быть ни одного конструктора (деструктора).

Переменную, описываемую типом `ИмяОбъекта`, называют *экземпляром объекта*:

```
VAR ЭкземплярОбъекта : ИмяОбъекта;
```

В программе может быть сколько угодно экземпляров одного объекта. Вызов метода – это оператор (или выражение) вида:  
<ИмяЭкземпляраОбъекта>.<ИмяМетода>[<СписокФактическихПараметров>].

**Пример 1.** Рассмотрим пример использования объектов при работе с символьной и текстовой информацией в текстовом режиме. На нижнем уровне определения типов объектов будем использовать переменные (X1,Y1) и (X2,Y2) - координаты углов окна, X,Y – переменные, определяющие позицию курсора в окне, C – цвет окна. Для задания значений координат углов окна и цвета фона будем использовать процедуру `INIT`, которая полям X1,Y1,X2,Y2,C присваивает передаваемые значения. Для задания значений, определяющих позицию курсора в окне, используем процедуру `INIT1`. Процедуру `SHOW` будем использовать для изображения окна.

```
USES CRT;
TYPE TypeWin = OBJECT
    x1,y1,x2,y2,c : integer;
    x,y           :integer;
    Procedure INIT(ax1,ay1,ax2,ay2,ac:integer);
    Procedure INIT1(ax,ay:integer);
    Procedure SHOW;
    End;
{реализация методов}

Procedure TypeWin.INIT;
Begin
    x1:=ax1; y1:=ay1; x2:=ax2; y2:=ay2; c:=ac
End;

Procedure TypeWin.INIT1;
Begin
    x:=ax; y:=ay
End;
```

```

Procedure TypeWin.SHOW;
Begin
    TextBackground(c);
    Window(x1,y1,x2,y2);
    ClrScr
End;
{описание экземпляров объекта}

Var o1 : TypeWin;

```

```

{операторная часть программы}           {или с оператором WITH}

Begin
    O1.INIT(10,5,50,20,red);
    O1.SHOW;
Readln
End.

```

```

With o1 do
Begin
    INIT(10,5,50,20,red);
    SHOW
End

```

В рассмотренном примере поля x,y и метод INIT1 не использовались. Они будут использоваться на следующем уровне определения объектов.

### Наследование.

Наследование – порождение новых типов объектов (потомков) из уже имеющихся типов объектов (предков или прародителей). Объекты Потомки получают по наследству все поля и методы объекта Предка, могут быть дополнены новыми полями и методами, и, кроме того, могут переопределять методы объекта Предка, сохраняя или изменяя при этом у одноимённых методов списки формальных параметров и типы возвращаемых значений.

Поля объекта Предка в объекте Потомке заменять нельзя.

Любой метод объекта Предка может быть вызван в любом методе объекта Потомка. В этом случае перед именем вызываемого метода указывается имя типа объекта Предка.

При вызове метода прямого Предка в версии Турбо Паскаль 7.0 достаточно предварительно указать ключевое слово **INHERITED**.

Синтаксически наследование выражается следующим образом:

```

Type Имя Объекта Наследника= ОБЪЕСТ(ИмяОбъектаПредка)
    Новые поля объекта наследника;
    Новые методы объекта наследника;
END;

```

Продолжим рассмотрение примера 1. При помощи типа объекта TypeWin можно задавать значения координат для указания позиции

выводимого элемента. Добавим новые действия: установить значение выводимого элемента и вывести элемент с заданной позиции на экран. Добавим также новые поля для обозначения выводимых элементов. Выводить будем символ или строку. Определим типы объектов наследников для выполнения этих действий.

```

                                {Для символа}
Type TypeSym = OBJECT(TypeWin)
    Ch:char;
    Procedure INIT1(ax,ay:integer; ach:char);
    Procedure SHOW1;
    END;

Procedure TypeSym.INIT1;
Begin
    Inherited INIT1(ax,ay);
    Ch:=ach
End;

Procedure TypeSym.SHOW1;
Begin
    GoToXY(x,y);
    Writeln(ch)
End;

                                {Для строки}

Type TypeStr = OBJECT(TypeWin)
    S : String;
    Procedure INIT1(ax,ay:integer; as:char);
    Procedure SHOW1;
    END;

Procedure TypeStr.INIT1;
Begin
    Inherited INIT1(ax,ay);
    S:=as
End;

Procedure TypeStr.SHOW1;
Begin
    GoToXY(x,y);
    Writeln(s)
End;

{описание экземпляров новых типов объектов}
Var och : TypeSym;
```

```
Os : TypeStr;
```

```
{пример возможных операторов после добавления новых объектных  
типов и новых описаний}
```

```
Begin  
    Och.INIT(10,5,50,20,red);  
    Och.INIT1(10,2,'!');  
    Och.SHOW;  
    Och.Show1;  
    Os.INIT1(3,2,'привет');  
    Os.Show1;  
    Readln
```

```
End.
```

Из программы видно, что одноимённые методы INIT1 и SHOW1 содержатся в разных объектовых типах TypeSym и TypeStr. Связь объекта с нужным методом (размещение ссылок на методы) устанавливается в процессе компиляции, так что операторы och.init1, os.init1 (а также och.show1 и os.show1) выполняют вызов разных методов. Говорят, что происходит раннее связывание. Методы, с которыми осуществляется раннее связывание, называют статическими. Действие компилятора сводится к следующему:

1)при вызове метода компилятор устанавливает тип объекта, вызывающего метод ;

2)в пределах найденного типа объекта компилятор ищет нужный метод, и, найдя его, назначает вызов метода;

3)если метод в пределах типа объекта не найден, то компилятор устанавливает тип непосредственного прародителя и там ищет метод и т.д.;

4)если вызванный метод нигде не найден, выдаётся сообщение об ошибке.

Можно продолжить ряд наследования. Добавим поле, соответствующее цветовому атрибуту символа, и расширим методы INIT1 и SHOW1. Новый объектовый тип, производный от TypeSym, может быть представлен в следующем виде:

```
Type TypeSymCol=OBJECT(TypeSym)  
    Color : Integer;  
    Procedure INIT1(ax,ay,acolor:integer;  
                    ach:char);  
    Procedure SHOW1;  
End;
```

```
Procedure TypeSymCol.INIT1;  
Begin  
    Inherited Init1(ax,ay,ach);  
    Color:=acolor
```

```

End;

Procedure TypeSymCol.SHOW1;
Begin
    TextColor(color);
    Inherited Show1
End;

{Описание экземпляра нового объектового типа:}
Var OchCol : TypeSymCol;

```

Можно показать в окне на дисплее символ, используя экземпляр объекта Och, и символ с его цветовым атрибутом, используя экземпляр объекта нового объектового типа :

```

Och.Init(10,5,50,20,red);
Qch.Init1(2,3,'o');
Och.Show; Och.Show1;
OchCol.Init1(3,3,yellow,'!');
OchCol.Show1.

```

Экземпляры объектов Och и OchCol вызывают различные одноимённые методы Init1 и Show1, причём метод Init1 в объектовых типах Och и OchCol имеет разное количество параметров.

Замечание: программист должен следить за тем, в каком окне следует отображать экземпляры объектов, нужно ли создавать новое окно и нужно ли удалять окно с экрана.

**Пример 2.** Рассмотрим пример использования объектов в графическом режиме. Опишем на нижнем уровне тип объекта для показа, стирания и передвижения по экрану точки с заданным цветовым атрибутом *c* и координатами изображения *x,y*. На следующем уровне добавим необходимые описания для выполнения тех же действий с окружностью. Наследуемые поля *x,y* будут соответствовать координатам центра окружности, добавленное поле *r* – радиусу окружности. Все методы в типе объекта наследника переопределим, сохранив за ними старые имена. На том же уровне наследования можно в дальнейшем расширить круг наследников, определив типы объектов для работы, например, с линией, прямоугольником и другими фигурами.

```

Uses Crt, Graph;
Var gd,gm:integer;

Type Tp=Object
    X,y,c : integer;
    Procedure Init(ax,ay,ac:integer);

```

```
        Procedure Show;
        Procedure Hide;
        Procedure Moveto(dx,dy:integer);
    End;
```

```
Procedure Tp.Init;
Begin
    X:=ax; y:=ay; c:=ac
End;
```

```
Procedure Tp.Show;
Begin
    Putpixel(x,y,c);
End;
```

```
Procedure Tp.Hide;
Begin
    Putpixel(x,y,Getbkcolor)
End;
```

```
Procedure Tp.Moveto;
Begin
    Delay(1000); Hide;
    X:=x+dx; y:=y+dy;
    Show
End;
```

```
Type Tc=Object(Tp)
    R : integer;
    Procedure Init(ax,ay,ac,ar : integer);
    Procedure Show;
    Procedure Hide;
    Procedure Moveto(dx,dy : integer);
End;
```

```
Procedure Tc.Init;
Begin
    Inherited Init(ax,ay,ac);
    R:=ar
End;
```

```
Procedure Tc.Show;
Begin
    Setcolor(c);
    Circle(x,y,r)
End;
```

```

End;

Procedure Tc.Hide;
Begin
    Setcolor(Getbkcolor);
    Circle(x,y,r)
End;

Procedure Tc.Moveto;
Begin
    Delay(1000); Hide;
    X:=x+dx; y:=y+dy;
    Show
End;

Var P : Tp;
    C : Tc;
Gd,gm: integer;
Begin
    Gd:=Detect;
    Initgraph(Gd,Dm,'c:\bp\bgi');
    P.Init(100,120,yellow);
    P.Show; P.Moveto(50,50);
    Readln;
    C.Init1(200,300,Green,150);
    C.Show; C.Moveto(10,10);
    Readln;
    Closegraph
End.

```

Благодаря связи, устанавливаемой между объектами при наследовании, возможны присваивания между переменными экземпляров объектов разных объектовых типов. Совместимыми по присваиванию являются кроме эквивалентных типов, объектовые типы, состоящие в отношении наследования **от** типа-потомка **к** родительскому типу, но не наоборот.

```

O1:=Och; O1:=Os; {в прмере 1}
P:=C;           {в примере2}

```

При этом копируются (присваиваются) только те поля, которые являются общими для обоих типов.

### Виртуальные методы.

Объектовые типы Tr и Tc содержат поля и методы для рисования, стирания и передвижения точек и окружностей на экране дисплея. Эти два объектовых типа связаны отношениями наследования и содержат одноимённые методы Show (нарисовать), Hide (удалить с экрана) и Moveto (передвинуть). Для различных геометрических фигур алгоритмы методов Show и Hide существенно отличаются. Алгоритм метода Moveto для обеих фигур одинаков (удалить фигуру со старого места, изменить координаты размещения фигуры и нарисовать ту же фигуру на новом месте). Естественным является желание определить метод Moveto для объектового типа Tr и наследовать этот метод без переопределения во всех типах объектов-потомков. Поясним невозможность такого подхода в данной проблеме без дополнительных затрат.

Допустим, что метод Moveto определён только в объектовом типе Tr. Если имеются экземпляры двух объектов: var P : Tr; C : Tc;, то вызов метода P.Moveto начнёт своё выполнение с метода Tr.Hide. Последующие действия метода Moveto приведут к ожидаемому результату. Теперь рассмотрим вызов C.Moveto. Экземпляр типа-потомка вызывает унаследованный метод Moveto, который жёстко связан с методами Tr.Show и Tr.Hide. Методы Show, Hide, Moveto были откомпилированы в одном контексте – в одном объектовом типе Tr. Поэтому метод Moveto всегда будет вызывать методы Tr.Show и Tr.Hide. Связь этих методов является **статической**, так как она была определена **при компиляции**. Методы C.Show и C.Hide вызваны не будут. Вызов C.Moveto приведёт к перемещению точки.

Если мы хотим иметь один метод Moveto для различных объектов, необходимо разорвать статическую связь этого метода с методами Show и Hide и обеспечить возможность для метода Moveto вызывать либо методы Tr.Show и Tr.Hide, либо Tc.Show и Tc.Hide в зависимости от того, какой объект вызывает метод Moveto. Такой механизм называют **динамическим** или **поздним связыванием** в отличие от статического или раннего связывания. Он достигается введением виртуальных методов.

Для определения метода как виртуального после заголовка метода в объектовом типе указывается служебное слово **VIRTUAL**.

При виртуализации методов должны выполняться следующие условия:

- 1)если прародительский объектовый тип описывает метод как виртуальный, производные типы метод с тем же именем также должны описывать как виртуальный;
- 2)заголовок в заново определённом виртуальном методе не может быть изменён;
- 3)если объектовый тип содержит виртуальный метод, он должен содержать хотя бы один метод-конструктор;
- 4)метод-конструктор должен быть применён к экземпляру объекта до первого вызова виртуального метода;
- 5)каждый экземпляр объекта должен быть инициализирован отдельным вызовом конструктора;
- 6)сам конструктор не может быть виртуальным.

На практике в качестве конструктора используют метод, устанавливающий начальные значения экземпляра объекта. В частности, конструктор может быть пустым. В рассмотренном примере конструктором является метод Init. Конструктор – это обычный метод-процедура, в котором служебное слово procedure заменено на **constructor**. Он помимо действий, заданных в его теле, выполняет установочную работу для механизма виртуальных методов, обеспечивая вызов в процессе выполнения программы именно того виртуального метода, который определён для вызывающего объекта. В примере 2 один и тот же метод Moveto будет работать по-разному (передвигать различные фигуры) в зависимости от того, экземпляр какого объектового типа этот метод вызывает. Такое свойство называется полиморфизмом.

Полиморфизм возникает на стыке принципов наследования и динамических связей. Это свойство является самым существенным в ООП. Именно это свойство отличает ООП от более традиционных методов программирования с использованием типов абстрактных данных.

“Полиморфизм – один из принципов теории типизации, состоящий в том, что имена могут соответствовать различным классам объектов, входящих в суперкласс. Следовательно, один объект, отмеченный таким именем, может по-разному реагировать на некоторое множество действий” /Гради Буч/.

**Полиморфизм** означает возможность определения единого по имени метода в каждом объектовом типе иерархической структуры разными способами.

Текст программы примера 2, использующий виртуальные методы, может быть следующим:

```
Uses Crt, Graph;
Var gd, gm : integer;

Type Tp=Object
    X,y,c : integer;
    Constructor Init(ax,ay,ac : integer);
    Procedure Show; Virtual;
    Procedure Hide; Virtual;
    Procedure Moveto(dx,dy : integer);
End;

Constructor Tp.Init;
Begin
    X:=ax; y:=ay; c:=ac
End;

Procedure Tp.Show;
Begin
```

```

        Putpixel(x,y,c);
End;

Procedure Tp.Hide;
Begin
    Putpixel(x,y,Getbkcolor)
End;

Procedure Tp.Moveto;
Begin
    Delay(1000); Hide;
    X:=x+dx; y:=y+dy;
    Show
End;

Type Tc=Object(Tp)
    R : integer;
    Constructor Init(ax,ay,ac,ar : integer);
    Procedure Show; Virtual;
    Procedure Hide; Virtual;
    End;

Constructor Tc.Init;
Begin
    Inherited(ax,ay,ac);
    R:=ar
End;

Procedure Tc.Show;
Begin
    Setcolor(c);
    Circle(x,y,r)
End;

Procedure Tc.Hide;
Begin
    Setcolor(Getbkcolor);
    Circle(x,y,r)
End;

Var P : Tp;
    C : Tc;

Begin
    Gd:=Detect;

```

```

    Initgraph(Gd,Dm,'c:\bp\bgi');
    P.Init(100,120,yellow);
    P.Show; P.Moveto(50,50);
    Readln;
    C.Init(200,300,Green,150);
    C.Show; C.Moveto(10,10);
    Readln;
    Closegraph
End.

```

Свойство совместимости, называемое полиморфизмом, можно продемонстрировать также и на примере 1. Опишем общую процедуру, выводящую символ или строку на экран в установленном окне и имеющую в качестве параметра переменную родительского типа.

```

Procedure PrintObj(Var Obj : TypeWin);
Begin
    Obj.Show1
End;

```

Телом этой процедуры является вызов метода Show1, по разному описанный в объектовых типах. В родительском объектовом типе TypeWin метод Show1 ранее не был определён. Теперь в объектовый тип TypeWin этот метод необходимо добавить. Добавить его можно и в виде “пустой” процедуры. Мы же поместим в эту процедуру вывод сообщения. В список заголовков методов объектового типа TypeWin добавим заголовок:

```

Procedure Show1:Virtual;

```

Реализация метода будет иметь вид:

```

Procedure TypeWin.Show1;
Begin
    Writeln('не вызывайте пустую процедуру')
End;

```

В объектовых типах TypeSym, TypeStr метод Show1 также объявим виртуальным.

В методах TypeWin, TypeSym и TypeStr объявим конструкторы. Конструктором можно объявить метод Init:

```

Constructor Init(ax1,ay1,ax2,ay2,ac:integer);

```

Передача параметров в процедуру или функцию с параметром-переменной объектового родительского типа (в нашем случае в процедуру PrintObj) осуществляется подобно правилам присваивания для операторов присваивания между переменными - экземплярами объектов. Параметру переменной родительского типа можно передавать в качестве фактического параметра переменную типа потомка (любой параметр, имеющий тип,

производный от типа TypeWin). При этом процедура PrintObj вызывает нужный метод обработки полей данных. Например ,  
Och.Init(10,5,50,red); Och.Show; Och.Init1(10,2,'!'); PrintObj(Och);  
Os.Init(10,5,50,red); Os.Show; Os.Init1(10,2,'строка'); PrintObj(Os);

**Пример 3.** Приведём пример использования виртуальных методов при работе с вектором и матрицей (предусмотрим ввод значений элементов рассматриваемой структуры данных, нахождение в структуре элемента с минимальным значением и вывод результата).

```
const nmax=10;
type tvect = array[1..nmax] of real;
    tmatr = array[1..nmax,1..nmax] of real;
    vect = object
        n:integer; {фактический размер вектора, количество строк матрицы }
        min : real;
        a    : tvect;
        constructor init;
        procedure inpt ; virtual; {ввод}
        procedure obr ; virtual; {нахождение минимального элемента}
        procedure out; {вывод}
        procedure work; {полная обработка структуры данных}
    end;

    constructor vect.init; begin end;

    procedure vect.inpt;
        var i : integer;
    begin
        writeln('n=?');
        readln(n);
        for i:=1 to n do
            begin
                write('a[i]-?'); readln(a[i])
            end;
        end;

    procedure vect.obr;
        var i : integer;
    begin
        min:=a[1];
        for i:=2 to n do
            if a[i]<min then
                min:=a[i]
            end;
        end;
```

```

procedure vect.out;
    begin writeln('min=',min:7:3) end;

procedure vect.work;
    begin inpt; obr; out end;

type matr=object(vect)
    m : integer; {количество столбцов матрицы}
    b : tmatr;
    procedure inpt; virtual; {ВВОД}
    procedure obr; virtual; {нахождение мин. элемента}
    end;

procedure matr.inpt;
    var i,j : integer;
begin
    writeln('n,m=?');
    readln(n,m);
    for i:=1 to n do
        for j:=1 to m do
            begin
                writeln('b[i,j]=?');
                readln(b[i,j])
            end;
        end;
end;

procedure matr.obr;
    var i,j : integer;
begin
    min:=b[1,1];
    for i:=1 to n do
        for j:=1 to m do
            if b[i,j]<min then
                min:=b[i,j]
        end;
    end;

var ObjVect : vect;
    ObjMatr : matr;

begin
    ObjVect.init;
    ObjVect.work;
    ObjMatr.init;
    ObjMatr.work;

```

```
readln  
end.
```

Здесь метод work осуществляет по требованию “общение” с одной из структур данных: вектором или матрицей.

**Пример 4.** Приведём пример реализации в виде объекта списка целых чисел. В качестве методов будем использовать следующие: инициализацию списка, добавление в список элемента с заданным значением, обработку элементов списка (печать) и метод, предназначенный для освобождения ранее выделенной памяти.

```
Type Link=^Rec;  
  Rec=Record  
    Inf : Integer;  
    Next : Link  
  End;  
Tsp=Object  
  Start : Link;  
  Procedure Init;  
  Procedure In_Spisek(X:integer);  
  Procedure Print_Spisek;  
  Procedure Done;  
  End;  
  
Procedure Tsp.Init; Begin Start:=Nil End;  
  
Procedure Tsp.In_Spisek;  
  Var p : Link;  
Begin  
  New(p);  
  P^.inf:=x;  
  P^.next:=Start;  
  Start:=p  
End;  
  
Procedure Tsp.Print_Spisek;  
  Var p : Link;  
Begin  
  P:=Start;  
  While p<>Nil do  
  Begin  
    Write(P^.inf:6);  
    P:=P^.next  
  End;  
End;
```

```

End;

Procedure Tsp.Done;
Var p : Link;
Begin
    While Start<> Nil do
        Begin
            P:=Start;
            Start:=Start^.next;
            Dispose(p)
        End
    End;
End;

Var Spisok : Tsp;
    A : integer;
    Ch : char;

Begin
    Spisok.Init;
    Repeat
        Readln(A);
        Spisok.In_Spisok(A);
        Readln(Ch);
    Until Ch in ['n', 'N'];
    Spisok.Print_Spisok;
    Spisok.Done
End.

```

Как и любые другие данные, экземпляры объектов можно размещать в динамической памяти с помощью процедуры **NEW** и удалять из неё с помощью процедуры **DISPOSE**. Сделаем необходимые добавления и изменения в примере 4.

Добавления к разделу описаний:

```

Type Ptsp=^Tsp;
Var Pspisok : Ptsp;

```

Добавим инициализацию указателя на объект:

```

New(Pspisok).

```

Покажем изменения в операторной части программы:

```

Pspisok^.Init;
Repeat
    ...
    Pspisok^.In_Spisok(A);
    ...
Until ...;
Pspisok^.Done;

```

Dispose(Pspisok)

Процедуры New и Dispose можно использовать в модифицированном виде (в качестве второго параметра записывать обращение к конструктору – в процедуре New, обращение к деструктору – в процедуре Dispose):

**New(Pspisok,Init);                      Dispose(Pspisok,Done).**

New можно использовать в виде функции:

**Pspisok:=New(Ptsp,Init).**

Здесь первый параметр – тип инициализируемой переменной, второй – обращение к конструктору. В левой части этого оператора используется инициализируемая переменная. В этом случае Dispose используется в том же виде, как и в предыдущем случае:

**Dispose(Pspisok,Done).**

**Пример 5.** Приведём пример размещения объектов в динамической памяти. Воспользуемся уже разработанной программой, работающей с геометрическими фигурами. Оформим работу с геометрическими фигурами в виде модуля, уменьшив количество методов. В качестве методов будем использовать методы Init и Show.

```
Unit Gr_obj;
  Interface
  Uses Crt, Graph;
  Type Тр=Object{объектовый тип - точка}
    X,y,c : integer;
    Constructor Init(ax,ay,ac : integer);
    Procedure Show; Virtual;
  End;
  Type Тс=Object(Тр){объектовый тип - окружность}
    R : integer;
    Constructor Init(ax,ay,ac,ar : integer);
    Procedure Show; Virtual;
  End;

  Implementation {раскрытие методов}
  Constructor Тр.Init;
  Begin
    X:=ax; y:=ay; c:=ac
  End;

  Procedure Тр.Show;
  Begin
    Putpixel(x,y,c);
  End;
```

```

Constructor Tc.Init;
Begin
    Inherited init(ax,ay,ac);
    R:=ar
End;

Procedure Tc.Show;
Begin
    Setcolor(c);
    Circle(x,y,r)
End;

```

End.

Объединим в список графические объекты. Каждый элемент линейного списка будет иметь два поля. Поле **Node** будет содержать указатель на графическую фигуру. Поле **Next** будет содержать ссылку на следующий элемент списка или значение **Nil**, если это последний элемент в списке. Создадим объектовый тип **List**, в котором предусмотрим необходимые для организации списка поле **Start** (указатель начала списка) и правила для работы с этим полем. С помощью правила **Add** можно добавлять новые элементы в список, с помощью правила **Out** -выводить содержимое списка на экран, а правило **Done** позволит уничтожить ранее созданный список.

В силу расширенного правила совместимости существует право на присваивание между указателями на родственные объекты. Так, если *pp* - указатель на объект *tp*, а *pc* – указатель на объект *tc*, причём, объекты *tp* и *tc* являются родственными, то возможны присваивания *pp:=pc* и *pc:=pp*. Расширенное правило совместимости для указателей на объекты позволяет создавать списки с указателями на элементы любых геометрических фигур. Используем это в программе .

```
Uses Crt, Graph, Gr_obj;
```

```

type  tp_ptr=^tp; {тип указателя на объектовый тип - точка}
      tc_ptr=^tc; {тип указателя на объектовый тип – окружность}
      link=^rec;  {тип указателя на тип элемента списка}
      rec=record {тип элемента списка }
          node : tp_ptr; {указатель на графическую фигуру}
          next : link   {указатель на следующий элемент списка}
      end;
      list=object {объектовый тип – список}
          Start : link;      {указатель на начало списка}
          constructor init; {инициализация списка}
          destructor done; {уничтожение списка}
          procedure add(elem:tp_ptr); {добавление элемента в список}
          procedure out; {просмотр списка с выдачей объектов на
экран}

```

```
end;
```

```
Constructor list.init; begin Start:=nil end;
```

```
destructor list.done;  
  var n:link;  
begin  
  while Start<>nil do  
  begin  
    n:=Start;  
    dispose(n^.node);  
    Start:=n^.next;  
    dispose(n)  
  end;  
end;
```

```
procedure list.add(elem : tp_ptr);  
  var n : link;  
begin  
  new(n);  
  n^.node:=elem;  
  n^.next:=Start;  
  Start:=n  
end;
```

```
procedure list.out;  
  var n : link;  
begin  
  n:=Start;  
  while n<>nil do  
  begin  
    n^.node^.show;  
    delay(5000);  
    n:=n^.next  
  end  
end;
```

```
Var  
  Alist : list;  
  pp   : tp_ptr;  
  pc   : tc_ptr;  
  ch   : char;  
  otv  : integer;  
  u,v,w,rad : integer;
```

```

gd,gm :integer;

Begin
  Alist.init;
  with alist do
  begin {создание списка геометрических фигур}
    ch:='y';
    repeat
      writeln('укажите тип геометрической фигуры 1 –
окружность, 2 - точка ');
      readln(otv);
      case otv of
        1:begin
          writeln('задайте координаты центра, цвет и радиус ');
          readln(u,v,w,rad);
          pc:=new(tc_ptr,init(u,v,w,rad));
          add(pc)
        end;
        2: begin
          writeln('задайте координаты точки и её цвет');
          readln(u,v,w);
          pp:= new(tp_ptr,init(u,v,w));
          add(pp)
        end
      end;
    until ch in ['n','N'];
    Gd:=Detect;
    Initgraph(Gd,gm,'c:\bp\bgi');
    out; {вывод элементов списка на графический экран}
    readln;
    done {возврат ранее выделенной памяти в “кучу”}
    end;
  Closegraph
  end.

```

Используя указатели на различные типы данных можно создавать массивы, содержащие в качестве элементов данные различного типа (полиморфные массивы) [5].

### Приложение

Рассмотрим приложение объектно-ориентированного подхода к решению следующей задачи о “Ханойской башне”. *Доска имеет три*

кольшика. На первом нанизано  $t$  дисков убывающего вверх диаметра. Расположить диски в том же порядке на другом кольшике. Диски можно перекладывать с кольшика на кольшик по одному. Класть больший диск на меньший не разрешается.

USES Graph;

CONST

n=5; {кол-во передвигаемых дисков}

TYPE

arr=array[1..n] of word; {тип массива дисков на оси }

{ Опишем два типа объектов: тип объекта-оси и тип объекта-администратора }

ax=object {тип объекта-оси: }

last, {перечислим член-данные }

col, {индекс верхнего диска оси }

x,y :word; {цвет дисков }

ar :arr; {координаты центра оси }

{ массив дисков }

{ флаги состояний : }

movf, {с оси передвигается верхний диск }

putf , {на ось кладётся диск }

parf:boolean; {чётность последнего диска оси }

{перечислим член-функции и процедуры }

procedure Init (f:boolean;xx,yy:word); {процедура инициализации }

procedure AbleToPut (d:word);{ устанавливает флаги parf и }

{ putf для значения d }

procedure AbleToMove (lm,d1,d2:word);{ устанавливает флаг movf }

procedure PutD (n:word);{ кладёт на ось диск n }

procedure GetD; { снимает верхний диск с оси }

function KnowL:word; { возвращает значение верхнего диска }

procedure Drow; { отображает ось }

procedure Hide; { стирает ось с экрана }

end;

admin=object { тип объекта-администратора }

l {перечислим член-данные }

astm:word; { значение передвигаемого диска }

parf, { флаг чётности количества }

передвигаемых }

```

vis :boolean;           { дисков }
axar :array[1..3] of ax { флаг видимости осей }
par,                    { массив объектов-осей }
                        { массив состояний осей:чётности,
                        }
put,                    { возможности положить диск }
mov :array[1..3] of boolean; { возможности взять диск }
процедуры               { перечислим член-функции и
                        }
  procedure Init;
  procedure Show;       { отображение состояния осей }
  procedure SetFArs;    { установка флагов состояния в }
                        { объектах-осях и массивов состояний
                        }
  procedure MoveD;      { перемещение дисков на осях }
  procedure Done;
end;

```

```

{*****
*****}

```

```

{ Описываем тела подпрограмм для объекта-оси }

```

```

procedure ax.Init;
var i:integer;
begin
  for i:=1 to n do
    if f then ar[i]:=n-i+1
      else ar[i]:=0;
    if f then last:=n
      else last:=1;
  parf:=(ar[last] mod 2=0);
  x:=xx;
  y:=yy;
  col:=12
end;

```

```

procedure ax.PutD(n:word);
{ кладем верхний диск }
begin
  if ar[last]>0 then last:=last+1;
  ar[last]:=n
end;

```

```

procedure ax.GetD;
{ снимаем верхний диск }

```

```
begin
  ar[last]:=0;
  if last>1 then last:=last-1
end;
```

```
function ax.KnowL:word;
{ возвращаем значение верхнего диска }
begin
  KnowL:=ar[last]
end;
```

```
procedure ax.Hide;
{ стереть ось }
var c:word;
begin
  c:=col;
  col:=GetBkColor;
  SetColor(col);
  Drow;
  col:=c;
  SetColor(white);
end;
```

```
procedure ax.Drow;
{ отобразить ось }
var i,yy:integer;
begin
  yy:=y;
  SetFillStyle(1,col);
  for i:=1 to last do
    if ar[i]>0 then begin
      bar(x-10*ar[i],yy,x+10*ar[i],yy,x+10*ar[i],yy-10);
      yy:=yy-10;
    end;
  if ar[last]=0 then line(x-25,y,x+25,y);
end;
```

```
procedure ax.AbleTomove(lm,d1,d2:word);
begin
  if (ar[last] mod 2 = 0) then parf:=true
    else parf:=false;

  if (ar[last]=lm)or(ar[last]=0) then movf:=false
    else
      if (ar[last]<d1)or(d1=0)or(d2=0)or(ar[last]<d2) then movf:=true
```

```

else movf:=false
end;

procedure ax.AbleToPut(d:word);
begin
  if (movf) or ((d>ar[last])and(ar[last]>0)) then putf:=false
  else
    if (ar[last]>0)and(parf=(d mod 2=0)) then putf:=false
    else putf:=true
  end;
{*****}
{ Описываем тела подпрограмм для объекта-администратора }

procedure admin.Init;

var i,gd,gm,
    grres :integer;

begin
  {инициализируем графический режим }
  DetectGraph(gd,gm);
  InitGraph(gd,gm,'c:\bp\bgi');
  grres:=graphresult;
  if grres<>grOk then writeln(grapherrormsg(grres))
  else
    begin
      {инициализируем оси }
      for i:=1 to 3 do
        if i=1 then axar[i].Init(true,i*(GetMaxX div 4),4*(GetMaxY div 5))
        else axar[i].Init(false,i*(GetMaxX div 4),4*(GetMaxY div 5));
        lastm:=0;
        vis:=true; { устанавливаем флаг видимости }
        self.Show; { прорисовываем }
        parf:=n mod 2=0;
        SetFArs
      end
    end;

procedure admin.Show;
var i:integer;
{ администратор в зависимости от значения флага видимости }
{ либо прорисовывает ось с дисками либо делает ее невидимой }
begin
  for i:=1 to 3 do

```

```

    if vis then axar[i].Drow
        else axar[i].Hide
end;

procedure admin.Setfars;
var i:integer;
begin
    { устанавливаем флаги о возможности перемещения }
    axar[1].abletomove(lastm,axar[2].knowl,axar[3].knowl);
    axar[2].abletomove(lastm,axar[1].knowl,axar[3].knowl);
    axar[3].abletomove(lastm,axar[2].knowl,axar[1].knowl);
    for i:=1 to 3 do
        begin
            par[i] :=axar[i].parf;
            mov[i]:=axar[i].movf;
            if mov[i] then lastm:=axar[i].knowl
        end;
    for i:=1 to 3 do
        begin
            axar[i].abletoput(lastm);
            put[i]:=axar[i].putf
        end;
    if put[2] and put[3] then if parf then put[3]:=false
        else put[2]:=false
end;

procedure admin.MoveD;
var i:integer;
begin
    repeat
        asm
            xor ax,ax
            int 16h
        end;
    vis:=not vis; { переключение флага видимости }
    self.Show;
    i:=1;
    { перемещаем диски, исходя из имеющихся условий }
    while not mov[i] do i:=i+1;
    case i of
        1:begin
            if put[3] then axar[3].PutD(lastm)
                else axar[2].PutD(lastm);
            axar[1].Getd;
        end;

```

```

2:begin
  if put[3] then axar[3].PutD(lastm)
    else axar[1].PutD(lastm);
  axar[2].Getd;
  end;
3:begin
  if put[2] then axar[2].PutD(lastm)
    else axar[1].PutD(lastm);
  axar[3].Getd;
  end
end;
vis:=not vis; { переключение флага видимости      }
self.Show; { прорисовываем                          }
SetFars { установка флагов состояния в            }
        { объектах-осях и массивов состояний      }
until axar[3].ar[n]=1
end;

procedure admin.Done;
begin
  Show;
  asm
  xor ax,ax
  int 16h
  end;
vis:=not vis; { переключение флага видимости      }
Show;
CloseGraph { выключаем графический режим          }
end;
{*****}
{*****}
      {              ОСНОВНАЯ ПРОГРАММА      }
VAR ad1:admin;
      { Заводим объект для администратора      }
BEGIN
  with ad1 do
  { Теперь выполняем подпрограммы              }
  begin
    Init; { инициализации,                      }
    MoveD; { управляющую передвижением дисков, }
    Done { программу завершения работы         }
  end;
END.

```

1. Фаронов В.В. Программирование на персональных ЭВМ в среде Турбо Паскаль. – М.: Изд-во МГТУ, 1992. – 448 с.
2. Фаронов В.В. Основы Турбо-Паскаля. – М.: МВТУ, 1992.- 285с.
3. Поляков Д.Б., Круглов И.Ю. Программирование в среде Турбо Паскаль М.: МАИ, 1992. – 575с.
4. Зуев Е.А. Система программирования Turbo Pascal. М.: Радио и связь, 1992. – 279с.
5. Абрамян М.Э. Указатели и объекты в Турбо Паскале. Часть 3. Ростов-на-Дону: УПЛ РГУ, 1996 – 42 с.