Кафедра конструирования и производства радиоаппаратуры

УТВЕРЖДАЮ
Заведующий кафедрой КИПР
_____**В.Н. ТАТАРИНОВ**
“___” _____2012 г.

Объектно-ориентированное программирование (ООП)

Лабораторная работа по дисциплинам **«**Информатика**»** для студентов специальностей 211000.62 «Конструирование и технология электронных средств» (бакалавриат) и 162107.65 «Информатика и информационные технологии» (специалитет)

Разработчик:
Доцент кафедры КИПР
_____**Ю.П. Кобрин**

Томск 2012

Оглавление

1 Цели работы

- Ознакомление с основными принципами и возможностями объектно-ориентированного программирования (ООП) в ***Borland Pascal***.
- Приобретение практических навыков использования динамических структур данных.
- Закрепление практических навыков в работе с экраном в текстовом и графическом режиме в ***Borland Pascal***.

2 Порядок выполнения работы

В ходе выполнения этой работы следует:

1. Овладеть основными понятиями объектно-ориентированного проектирования среде ***Borland Pascal***, обратив особенное внимание на правила использования динамических структур данных. При необходимости использовать дополнительную литературу [1 - 9].
2. Ответить письменно на контрольные вопросы.
3. Войти в свой личный каталог, и настроить интегрированную среду ***Borland Pascal*** для последующей работы. Записать файл конфигурации в личный каталог.
4. Получить вариант индивидуального задания у преподавателя, составить и отладить программу.
5. Внимательно проштудировать учебный пример и по возможности максимально использовать его в своей задаче.
6. Ввести и отладить разработанную программу.
7. Продемонстрировать работоспособность программы.
8. Оформить отчет по лабораторной работе и защитить его у преподавателя.

3 Контрольные вопросы

Ответьте письменно на следующие контрольные вопросы:

1) Что такое указатели? Какие значения они могут принимать?
2) Какие операции возможны над указателями?
3) Что представляют собой динамические структуры данных? Для чего они используются? Чем отличаются от данных статического типа?
4) В чем особенности ООП? В чем его преимущества?
5) Что такое объект и правила его описания?
6) Что такое инкапсуляция, наследование и полиморфизм?
7) Каково назначение виртуальных методов?
8) Каковы функции конструкторов и деструкторов?

4 Отчетность

Отчет должен быть выполнен в соответствии с [10] и состоять из следующих разделов:

1) Тема и цель работы.
2) Индивидуальное задание.
3) Схема алгоритма решения задачи.
4) Текст программы и вводимые тестовые исходные данные.
5) Откомпилированный текст программы-заготовки (в электронном виде).
6) Результаты выполнения программы.
7) Ответы на контрольные вопросы.
8) Выводы.

При защите отчета по работе для получения зачета студент должен:

- *уметь* отвечать *на контрольные вопросы*;
- обосновать структуру выбранного алгоритма;
- уметь пояснять работу программы и доказать ее работоспособность;
- продемонстрировать *навыки работы в среде* ***Borland Pascal***.

5 Динамические переменные

5.1 Постановка задачи

В ходе компиляции программы для всех данных, описанных в программе в разделах *Var* и *Const*, в зависимости от их типа выделяется необходимое количество байтов в оперативной памяти компьютера, и устанавливаются необходимые связи между ними. Структурированные типы данных, такие, как массивы, множества, записи, представляют собой в ***Borland Pascal*** *статические структуры*, так как после определения их размеров на этапе компиляции до завершения работы программы, *память* занятую ими использовать для других целей *нельзя*. Это накладывает серьезные ограничения на размер программы и количество обрабатываемых данных. В первую очередь это относится, например, к сложным конструкторским чертежам и другим графическим приложениям, информационным базам данных, массивам с заранее неизвестным числом элементов, связанным спискам данных или объектов, если программа работает с большими объемами данных, общий объем которых превышает 64К.

Часто требуется, чтобы структуры данных меняли свои размеры в ходе решения задачи. Такие структуры (*стеки, очереди, списки, деревья* и другие) данных называются *динамическими*. ***Borland Pascal*** позволяет по ходу выполнения программы выделять и освобождать необходимую память для размещения в ней различных данных, что позволяет организовывать подобные динамические (т.е. изменяющиеся размеры) структуры данных. Использование динамических структур данных дает возможность писать более сложные и гибкие программы, что немаловажно при написании объектно-ориентированных программ.

5.2 Указатели и динамические переменные

Большая часть памяти компьютера, свободной от программ и данных (англ. *heap* - «куча»), может быть занята динамическими переменными.

Для работы с динамическими структурами используются указатели.

> ***Указатель*** – (*англ. - pointer*) это ссылка на данные или код Вашей программы.

Он представляет специальный тип данных - *адрес начала какого-либо элемента* в памяти компьютера. Это может быть адрес переменной, записи, объекта, либо процедуры или функции. Обычно память компьютера делится на *сегменты* – непрерывные участки памяти не более 65536 байт. Адрес памяти состоит из двух шестнадцатеричных слов – *адреса сегмента* и *смещения*, показывающего положение участка памяти относительно начала сегмента. Использование идентификатора указателя в программе означает обращение к адресу ячейки памяти, на которую он указывает.

Для хранения значений указателей необходимо сформировать переменные-указатели, для чего в Паскале предусмотрен специальный ссылочный тип (или тип «указатель»).

Переменная-указатель занимает 4 байта памяти (2 слова). Первое слово дает смещение адреса относительно начала сегмента, второе слово – адрес сегмента. Например, $03AE:BF29.

Простейшим ссылочным типом является ***нетипизированный указатель*** со стандартным типом ***Pointer***, который описывается следующим образом:

```
Var
  <имя указателя>: Pointer;
```

Переменная типа ***Pointer*** - это просто адрес. Он не содержит информации о том, на какой именно элемент он указывает, и совместим со всеми типизированными указателями

Типизированные указатели указывают *на конкретный вид элемента*, например, целое значение, запись данных, объект и т.п. Отметим, что указатели, ссылающиеся на объекты разных типов, сами являются объектами разных типов, и потому для них недопустимы операции присваивания значений друг другу. Указатели одного и того же типа можно лишь сравнивать с помощью операций **=** и **<>**.

Чтобы определить ***типизированный указатель*** на какой-либо элемент, следует описать новый тип указателя, определенный символом ***каре*** (**^**), за которым следует тип этого элемента:

```
Type
  <имя типа указателя> = ^<тип элемента>;

Var
  <имя указателя, …>: <имя типа указателя>;
```

Например,

```
Type  {описание нестандартных типов данных}
  pInteger = ^Integer; {указатель на переменную целого типа Integer}
  pVector = ^tVector;    {указатель на массив типа  tVector, описание типа
                          которого следует позже}
  tVector = array[1..10] of real; {тип данных – массив из десяти вещественных
                                   чисел}

Var
  Number: pInteger;  {создана переменная-указатель на переменные целого
                      типа}
  A: pVector; {создана переменная-указатель на массив из вещественных чисел}
```

Для обращения к конкретному содержимому динамической переменной, адрес которой содержит указатель, в Паскале используется операцией «*разыменования*»:

```
<имя указателя>^
```

Например,

```
Number^ := 5;  {элементу, на который указывает Number присваивается 5}
```


```
A^[3] := 1.6; {элементу 3 массива A в динамической памяти присвоить 1.6}
```

Для получения адреса какого-либо элемента в памяти (переменной, подпрограммы и т.п.) применяется операция @, например

p := @A; {*p* – адрес начала массива A, определенного нами ранее}

5.3 Выделение памяти для динамических переменных

Одним из наиболее важных моментов использования указателей является распределение динамических переменных в динамически распределяемой области памяти. Для динамического распределения памяти чаще всего используют процедуры ***New*** и ***Dispose***.

Пример 1. Выделить пространство в динамически распределяемой памяти для переменных типа ***Integer*** и ***String*** с помощью процедуры ***New***[1].

```
var
   pInt: ^Integer;          {указатель на переменную целого типа}
   pStroka: ^String;        {указатель на строку символов}
begin
   New(pInt);               {выделяет в динамически распределяемой области два байта
                             под переменную типа integer }
   New(pStroka);            {выделяет в динамически распределяемой области 256 байт
                             под строку символов}.
   …
end.
```

После вызова процедуры ***New*** переменная-указатель указывает на память, выделенную в динамически распределяемой памяти. В данном примере ***pInt*** указывает на двухбайтовую область, выделенную процедурой ***New***, а ***pInt^*** - это допустимая целочисленная переменная (хотя это целочисленное значение еще не определено). Аналогично, ***pStroka*** указывает на выделенный для строки 256-байтовый блок, а ***pStroka^*** дает доступную для использования строковую переменную.

Важно! Использовать динамические переменные можно только после выделения для них памяти. Иначе указатели на них будут показывать в «никуда» и компьютер, скорее всего, «зависнет».

После завершения работы динамическую память, распределенную для переменных с помощью ***New(p)***, необходимо освободить с помощью процедуры ***Dispose(p)***. Это позволит использовать область памяти, на которую указывает указатель ***p***, для дальнейшего использования. Значение указателя ***p*** после этого становится *неопределенным*. Например,

```
Dispose(pStroka);  {освобождает в динамически распределяемой области 256
                    байт, на которую указывал указатель pStroka}.
```

---

[1] ***Borland Pascal*** предусматривает и другие возможности работы с динамической памятью (в частности, с помощью процедур ***GetMem, FreeMem, Mark*** и ***Release***).

Dispose(pInt); {освобождает в динамически распределяемой области два байта, на которую указывал указатель на целое число ***pInt*** }

5.4 Использование пустого указателя

Если вы попытаетесь получить значение переменной, указатель которой не определен (к примеру, процедура ***New*** еще не задействована), то произойдет ошибка – обращение к несуществующему адресу. В Паскале предусмотрено зарезервированное слово ***nil***, которое вы можете использовать в качестве содержательного значения указателей, которые в данный момент ни на что не указывают. Указатель ***nil*** является допустимым, но ни с чем не связанным.

Запомните! Если содержимое указателя ***nil***, то для Вас это будет означать, что адрес этой переменной еще не определен.

6 Объектно-ориентированное программирование (ООП)

6.1 Постановка задачи

Одной из наиболее перспективных современных информационных технологий, активно влияющих на идеологию современного программирования, позволяющих существенно поднять качество разработки программного обеспечения и уменьшить время на его создание[2] является *объектно-ориентированное программирование* (ООП). ООП является дальнейшим развитием идей структурного и модульного программирования, новым, нетрадиционным уровнем абстрагирования процессов обработки информации, позволяющим на основе готовых *базовых* программных конструкций и компонент просто и наглядно решать многие проблемы при создании САПР, графических приложений и т.п., особенно трудные для традиционного программирования.

Объектно-ориентированное программирование (ООП) - это технология программирования, в основе которой лежит понятие объекта как некоторой модели, соответствующей объекту реального мира и его поведению.

Программа, созданная на основе технологии ООП, состоит из описаний объектов и последовательностей операций над объектами, изменяющих их состояние.

Информация, воспроизводимая на экране дисплея, может быть представлена в виде текста (набор отображаемых символов) и в графическом виде (рисунки, чертежи и т.п.). Паскаль позволяет выводить информацию как в текстовом, так и в графическом режиме. Наибольшие возможности предоставляет графический режим, в котором любое изображение синтезируется из множества отдельных точек (пикселей). Создание и отображение графических изображений с помощью пикселей требует основательных затрат памяти. Вследствие этого создание сложных изображений, модифицирующихся в диалоговом режиме – весьма выгодная область применения методов ООП.

[2] По некоторым оценкам – до 10 раз!

6.2 Основные понятия

В основе ООП лежит понятие *объекта*[3] (***object***), очень напоминающего запись. Объект служит "оболочкой" для соединения фиксированного числа связанных между собой компонент *под одним именем*. Каждая компонента - это или поле (описывает свойства объекта), которое содержит данные определенного типа, или ***метод***, который производит операции над объектом (реализует поведение объекта). Аналогично объявлению переменных, объявление поля в объекте указывает тип данных поля и идентификатор имени этого поля. Но кроме полей данных, в объявление объекта дополнительно включают заголовки методов - процедур и функций, предназначенных для выполнения операций над этими полями.

Объекты характеризуется тремя основными функциональными характеристиками:

1. *Инкапсуляция*: объединение записей с процедурами и функциями, работающими с полями этих записей, превращает их в новый тип данных - *объект*. Другими словами, объект – это пакет информации вместе с алгоритмом ее обработки. Отношения частей к целому и взаимоотношения между частями объекта становятся понятнее тогда, когда поля (определяющие свойства объекта) и методы (определяющие поведение объекта) содержатся вместе, т.е. *инкапсулированы* (встроены) в описание соответствующего объекта.

2. *Наследование*: заимствование и преемственность атрибутов и поведения от объекта к объекту. Любой объект-потомок наследует все поля данных, а также процедуры и функции своего объекта-предка. У объекта-потомка могут быть объявлены новые поля и методы, а некоторые методы объекта-предка перекрыты .

3. *Полиморфизм*: присваивание методу одного имени, которое затем совместно используется *вниз* и *вверх* по иерархии объектов, причем каждый объект иерархии выполняет это действие своим способом, именно ему подходящим. Полиморфизм повышает степень абстрагирования при написании программ. Программист указывает, *что* он хочет выполнить (например, нарисовать объект), а *как* это сделать - благодаря *позднему связыванию* уточняется (в зависимости от того, какому объекту послано сообщение) на этапе выполнения программы, а не на этапе компиляции, как при традиционном программировании.

6.3 Наследование

Далее мы будем рассматривать пример создания иерархической структуры объектов графической среды (*рис. 6.1*).

Для обозначения координат любого графического объекта в графической среде разумно взять вместе координаты ***X*** и ***Y*** позиции на графическом экране и назвать это типом ***объект*** с именем ***tLocation***:

```
Type
  tLocation = object        {родительский (автономный) объект}
    X, Y: Integer;          {координаты экрана}
  end;
```

Тип объект может быть задан либо как законченный, *автономный тип*, либо как порождение *существующего типа объекта*, путем помещения имени родительского типа в круглых скобках после зарезервированного слова ***object.***

---

[3] Синоним термина «класс» в языке C++, Delphi.

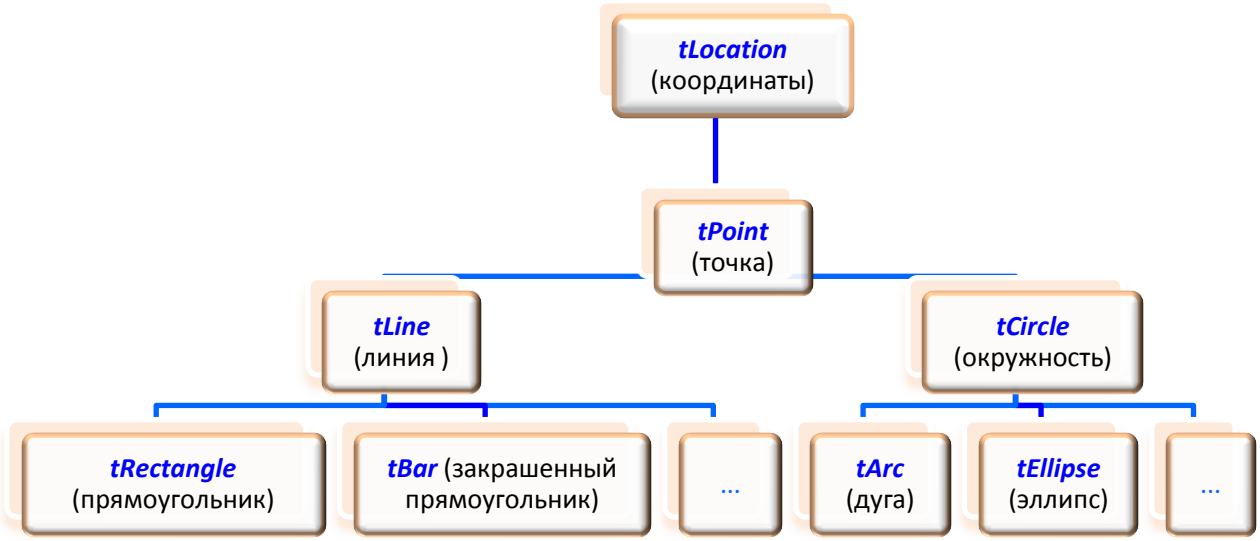


Рисунок 6.1 - Иерархическая структура объектов графической среды

Предположим, необходимо нарисовать *точку* в позиции, описанной на экране с помощью координат, заданных ***tLocation***. Целесообразно добавить булевское поле, указывающее, светится ли пиксель в данной позиции, и сделать это новым типом объекта ***tPoint,*** унаследовав все из объекта ***tLocation***:

```
Type
  {порожденный от объекта-предка tLocation объект tPoint }
   tPoint = object(tLocation) {поля X, Y наследуются от объекта-предка tLocation}
      visible: Boolean;          {добавляется поле - признак светимости}
   end;
```

Здесь ***tLocation*** является родительским типом, а ***tPoint*** - порожденным типом. Этот процесс может продолжаться бесконечно (см. *рис. 6.1*): можно задать наследников типа ***tPoint***, наследников типа, порожденного от ***tPoint*** и т.д.

Экземпляры типа объект объявляются либо как статические, либо как указатели на экземпляры объектов, размещенных в динамической памяти:

```
Type
   pPoint = ^tPoint; {тип – указатель на объект tPoint}
Var
   StatPoint: tPoint;     {статическая переменная - готова к использованию}
   DynamPoint: pPoint; {перед использованием разместить с помощью New}
```

Доступ к полям данных объекта можно получить либо применяя оператор *With*, либо используя составное имя (с точкой).

Например:

```
StatPoint.Visible := false; {Полю Visible объекта StatPoint  присваивается false }
With StatPoint do {Присвоение значений полям X и Y объекта StatPoint }
   begin
      X := 143;
      Y := 42;
   end;
```

Унаследованные поля доступны так же, как и поля, объявленные внутри данного типа объекта. Например, хотя ***X*** и ***Y*** не являются частью объявления ***tPoint*** (они наследуются из типа ***tLocation***), можно задавать их значения, как будто бы они объявлены внутри ***tPoint***[4]:

```
   DinamPoint^.X := 387;
```

Заметим, что любой порожденный объект наследует не только поля, но и правила поведения (методы) объекта-предка.

6.4 Методы

Для объединения полей и подпрограмм, обслуживающих их, в единое целое (инкапсуляции) применяются ***методы***. Метод (или правило поведения объекта) - это процедура или функция, объявленная внутри объекта для выполнения определенных действий над его полями. Заголовки методов указываются при объявлении типов объектов так же, как это делается при объявлении подпрограмм в секции ***interface*** для модулей. Так полное описание объекта ***tLocation*** может выглядеть так:

```
{объект tLocation - координаты графического примитива}
pLocation = ^tLocation;  {указатель на координаты графического примитива}
tLocation = object       {тип объекта - координаты графического примитива}
   X, Y: Integer;        {поля с координатами графического примитива}
   procedure Init(NewX, NewY: Integer);     {инициализация координат}
   function GetX: Integer;                  {определение текущей абсциссы}
   function GetY: Integer;                  {определение текущей ординаты}
end;
```

Объявления метода внутри объекта говорят, что метод делает, но не как. Как - задается вне определения объекта в отдельном определении процедуры или функции. Поскольку сразу несколько типов объектов могут включать методы *с одинаковыми именами*, при описании реализации любого метода перед заголовком через точку указывается имя типа объекта, которому метод принадлежит:

---

[4] Заметим, что прямое обращение - не лучший способ инициализации полей объекта. В ООП для этих целей, как правило, используется специальное правило (метод) ***Init***

```
procedure tLocation.Init(NewX, NewY: Integer); {инициализация координат}
begin
   X := NewX;
   Y := NewY
end;

function tLocation.GetX: Integer; {определение текущей абсциссы}
begin
   GetX := X
end;
function tLocation.GetY: Integer; {определение текущей ординаты}
begin
   GetY := Y
end;
```

При обращении к методу перед его именем следует указать через точку имя экземпляра объекта, для которого этот метод применяется (если к одному и тому же экземпляру объекта требуется последовательно применить несколько методов подряд, то удобнее употребить оператор ***with***):

```
Var
   Location: tLocation; {Объявление экземпляра объекта в статической памяти}
      …
   Location.Init(34, 167); {инициализация координат экземпляра объекта Location}
```

При желании методы порожденного объекта могут быть *переопределены*. Для переопределения наследуемого метода достаточно описать новый метод с тем же именем, что и наследуемый метод, *но с другим телом и другим множеством параметров* (при необходимости). Поля данных переопределяться *не могут*.

Методы, обсуждаемые выше получили название *статических*. Они являются статическими по той же причине, по которой статические переменные являются статическими: компилятор размещает их и разрешает ссылки на них *во время компиляции*[5].

Статические методы всегда сильно привязаны к типу объекта, для которого они были введены. При вызове статического метода какого-либо экземпляра объекта вначале ищется метод, имя которого определено внутри этого типа объекта. Если метод не обнаружен, то метод с таким именем ищется у родительского объекта, затем у следующего предка и т. д. по иерархии родительских объектов.

Обратите внимание! Если родительский метод вызывает другие методы, то вызываемые методы будут *родительскими методами*, даже если потомок имеет методы, которые отвергают родительские методы.

[5] Так называемое «*раннее связывание*» методов с данным типом объекта.

6.5 Использование объектов в модулях

Весьма удобно для создания библиотек объектов применять пользовательские модули. Объявление типов объектов выполняются, как правило, в интерфейсном разделе модуля (***interface***), а тела подпрограмм-методов, описываются в разделе реализации (***implementation***) модуля.

Чтобы использовать типы объектов и методы, определенные в модуле, достаточно просто использовать этот модуль в своей программе и объявить экземпляры объектов в ***var*** разделе своей программы.

6.6 Виртуальные методы и полиморфизм

Виртуальные методы реализуют чрезвычайно мощное средство для обобщения, называемого ***полиморфизмом***. Полиморфизм - означает способ задания одноименного действия, которое распределяется вверх и вниз по иерархии объектов, с выполнением этого действия способом, соответствующим каждому объекту в иерархии.

В рассматриваемом примере каждому типу объекта соответствуют различные фигуры на экране: точка, окружность, линия и т.д. Способы отображения различны для каждого типа объекта. Так точка рисуется с помощью подпрограммы ***PutPixel*** стандартного модуля ***Graph*** ***Borland Pascal***, окружность – ***Graph.Circle*** и т.п. Для всех фигур следует написать свои методы, которые изобразят их на экране, причем ***Borland Pascal*** обязывает, чтобы эти методы имели *разные имена*. Тем не менее, полиморфизм методов объектов требует, чтобы любые графические фигуры на экране изображались виртуальными методами с одним и тем же именем - например, ***Draw***[6], а стирались - с помощью метода ***Hide*** и т.п.

Метод становится виртуальным, если следом за его объявлением добавлено служебное слово ***virtual***. В отличие от статических методов виртуальные методы присоединяются к общей программе уже во время выполнения программы (позднее связывание). Использование виртуальных методов дает возможность использовать общие процедуры для выполнения однотипных групповых действий. Например, передвижения любых объектов осуществлять с помощью одного и того же метода ***MoveTo***.

Запомните! Если метод в родительском типе объявлен виртуальным, то все методы с таким же именем в любом порожденном типе должны быть также объявлены виртуальными, чтобы избежать ошибки компиляции.

Кроме того, каждый тип объекта, который имеет виртуальные методы, должен иметь и процедуру специального типа - конструктор (***constructor***), которая делает некоторую работу по настройке механизма обработки виртуальных методов. Конструкторы никогда не бывают виртуальными!

Для завершения работы с экземпляром объекта используют специальную процедуру *деструктор* (***destructor***).

[6] Также, как и метод «двигаться» применим и к человеку, и к животному, и к автомобилю и т д., хотя выполняется по разному.

Очень важно! Перед использованием виртуальных методов *любого отдельного экземпляра объекта* следует *обязательно вызвать его конструктор* (обычно это инициализация полей объекта - метод ***Init***). Если обратиться к виртуальному методу *до вызова конструктора*, то, скорее всего, компьютер «зависнет».

6.7 Динамические объекты

Объекты могут быть размещены в куче как указатель, на который ссылается процедура ***New***:

```
Var {Выделяем память:}
  pPoint := ^tPoint; { для указателя на экземпляр объекта «точка»}
  …
  New(pPoint); {выделяет необходимый объем памяти в куче для размещения
                объекта «точка» и возвращает адрес выделенной памяти в указа-
                теле.pPoint}
```

Если динамический объект содержит виртуальные методы, то его нужно затем проинициализировать с помощью вызова конструктора перед любыми вызовами методов этого объекта:

```
  pPoint^.Init(600,100); {вызывается метод Init для  экземпляра объекта,
                          динамически размещенного в куче,  адрес которого задан
                          указателем.pPoint}
```

Вызовы методов затем могут осуществляться обычным образом с помощью имени указателя и символа ссылки ^(знак вставки), используемых вместо имени экземпляра, которое применялось бы в вызове статически размещенного объекта:

```
  OldXPosition := pPoint^.GetX; {возвращается текущее значение абсциссы
                                 объекта, динамически размещенного в куче,
                                 адрес которого задан указателем.pPoint}
```

Расширенный синтаксис ***New Borland Pascal*** позволяет использовать более удобный способ распределения памяти для объекта, находящегося в куче с одновременной инициализацией значений полей объекта. ***New*** может теперь вызываться с помощью двух параметров: в качестве первого параметра используется имя указателя, а второго параметра - вызов конструктора:

```
  New(pPoint, Init(600,100)); {выделяется динамическая память для объекта
                               tPoint, адрес его будет находиться в указателе
                               pPoint на этот объект. Одновременно полям
                               объекта присваиваются значения X := 600, Y := 100}
```

Аналогичных результатов можно добиться, используя два оператора:

```
  New(pPoint);
  pPoint^.Init(600,100);
```

Когда объекты уже больше не нужны, память, отведенная им в куче, может быть освобождена с помощью процедуры ***Dispose***:

```
  Dispose(pPoint); {освобождается память, занятая объектом tPoint}
```

Для очистки кучи от динамически распределенных объектов в ***Borland Pascal*** также применяют деструкторы. Деструктор комбинирует шаг освобождения памяти в куче с некоторыми другими задачами, которые необходимы для данного типа объекта. Чтобы выполнить освобождение памяти при позднем связывании, деструктор нужно вызвать как часть расширенного синтаксиса для процедуры ***Dispose***, например:

Dispose(pPoint, Done); {где Done - деструктор}

7 Учебный пример

7.1 Пример объектно-ориентированной программы

С учетом сделанных разъяснений, в качестве примера приведем программу для изображения и перемещения простейших графических объектов точка и окружность. Учебная программа ***DemoObj*** использует сокращенную версию модуля ***GraphObj*** с простейшими графическими объектами.

```
Program DemoObj; {Демонстрация работы с графическими объектами}
Uses {подключаем модули}
  GraphObj, {простейшие графические объекты}
  Graph,    {системный модуль Паскаля с графическими подпрограммами}
  CRT;      {системный модуль Паскаля с подпрограммами работы с экраном}
Var
  StatPoint: tGrPoint;        {экземпляр статического объекта – точка}
  DinamPoint: pGrPoint;       {указатель на динамический объект – точка}
  StatCircle: tCircle;        {экземпляр статического объекта – окружность}
  DinamCircle: pCircle;       {указатель на динамический объект – окружность}

{==== Совместная буксировка двух окружностей ====}
procedure DragTwoCircle(DragBy: Integer); {DragBy - шаг перемещения}
Var
  DeltaX, DeltaY: Integer; {коды перемещения по осям X и Y}
begin
  StatCircle.Draw;          {изобразить 1-ю окружность}
  DinamCircle^.Draw;        {изобразить 2-ю окружность}

  {цикл буксировки 2-х объектов}
  while GetDelta(DeltaX, DeltaY) do
    begin (перемещаем окружности в новую позицию}
      StatCircle.MoveToRel(DeltaX * DragBy, DeltaY * DragBy);
      DinamCircle^.MoveToRel(DeltaX * DragBy, DeltaY * DragBy)
    end
```

```
end; {конец DragTwoCircle}

begin {начало основного блока программы DemoObj }
  StartGraphics; {инициализация графического режима}
  SetColor(White); SetBkColor(Cyan); {установка цвета символов и линий
                                      (белый) и фона (голубой) экрана}
  ClearDevice; {Очистка графического экрана}

  OutTextXY(190, 50, 'Демонстрация графического режима');

  SetColor(Red);  {установка цвета – красный}
  StatPoint.Init(100, 75); {Инициализация координат статической точки}
  StatPoint.Draw;          {Изобразить статическую точку}
  OutTextXY(180, 75, 'В этой строке красная статическая точка');

  SetColor(Magenta);     {установка цвета – фиолетовый}
  New(DinamPoint, Init(100, 90)); {Инициализация координат динамической точки}
  DinamPoint^.Draw;      {Изобразить динамическую точку}
  OutTextXY(180, 90, 'В этой строке фиолетовая динамическая точка');

  SetColor(Blue); {установка цвета – синий}
  StatCircle.Init(100, 200, 50);   {Инициализация координат статической
                                    окружности}
  StatCircle.Draw;                 {Изобразить статическую окружность}
  OutTextXY(180, 200, 'В этой строке синяя статическая окружность');

  SetColor(Yellow); {установка цвета – желтый}
  {Выделяем динамическую память под объект окружность и одновременно ини-
  циализируем значения ее координат и радиуса}
  New(DinamCircle, Init(100, 300, 75));
  DinamCircle^.Draw; {Изобразить динамическую окружность}
  OutTextXY(180, 300, 'В этой строке желтая динамическая окружность');

  SetColor(White);              {установка цвета – белый}
  OutTextXY(180,400, 'Для продолжения нажмите <<Enter>> ..');
  Readln;
  SetBkColor(LightGray);        {установка цвета фона - светло-серый}
  SetColor(Magenta);            {установка цвета – фиолетовый}
```

```
  ClearDevice;                  {Очистка графического экрана}

  OutTextXY(190, 50, 'Демонстрация перемещения объектов');
  StatCircle.Draw;          {Изобразить статическую окружность}
  DinamCircle^.Draw;        {Изобразить динамическую окружность}
  OutTextXY(180, 70, 'Перемещайте 1-ю окружность стрелками ..');
  OutTextXY(180, 400, 'Для продолжения нажмите <<Enter>> ..');

  StatCircle.Drag(10);      {буксировка 1-й окружности с шагом 10}
  ClearDevice;              {Очистка графического экрана}

  OutTextXY(190, 50, 'Демонстрация перемещения объектов');
  SetColor(LightRed);           {установка цвета - светло-красный}
  OutTextXY(180, 70, 'Перемещайте 2-ю окружность стрелками ..');
  OutTextXY(180, 400, 'Для продолжения нажмите <<Enter>> ..');

  StatCircle.Draw;              {Изобразить статическую окружность}
  DinamCircle^.Draw;            {Изобразить динамическую окружность}
  DinamCircle^.Drag(10);        {буксировка 2-й окружности с шагом 10}
  ClearDevice;                  {Очистка графического экрана}
  OutTextXY(190, 50, 'Демонстрация перемещения объектов');
  SetColor(Yellow);             {установка цвета – желтый}

  OutTextXY(180, 70, 'Перемещайте обе окружности стрелками ..');
  OutTextXY(180, 400, 'Для окончания работы нажмите <<Enter>> ..');
  DragTwoCircle(10);        {буксировка двух окружностей с шагом 10}

  {удаление объектов из памяти}
  StatPoint.Done;
  StatCircle.Done;
  DinamPoint^.Done;
  DinamCircle^.Done;
  CloseGraph {закрываем графический режим}
end.  {конец основного блока программы DemoObj }
```

7.2 Модуль GraphObj с простейшими графическими объектами для учебного примера

Unit GraphObj; {Модуль - Простейшие графические объекты}

{***tLocation*** - координаты графического объекта}

{***tGrPoint*** - графическая точка}

{***tCircle*** - окружность}

{=== Интерфейсный раздел с объявлениями экспортируемых типов данных, объектов, процедур и функций ===}

interface

uses

Graph, {системный модуль Паскаля с графическими подпрограммами}

CRT, {системный модуль Паскаля с подпрограммами работы с экраном}

Objects; {модуль *Turbo Vision* с базовыми подпрограммами работы с объектами}

Const

{PathToDrivers - строка, содержащая полный путь к драйверу графического режима ****.BGI***. По умолчанию предполагается, что драйвер находится в текущей директории, {иначе значение этой типизированной константы нужно уточнить в своей программе}

PathToDrivers: string[40] = '';

Var

GraphDriver: Integer; {номер графического драйвера}

GraphMode: Integer; {номер графического режима}

ErrorCode: Integer; {код ошибки графического режима}

{=== ***StartGraphics*** - инициализация графического режима ===}

procedure StartGraphics;

{=== ***GetDelta*** ===}

{Функция ***GetDelta*** возвращает ***DeltaX***, ***DeltaY*** - коды перемещения по осям ***X*** и ***Y***}

{Функция ***GetDelta*** = ***True***, если нажаты стрелки перемещения курсора}

{Функция ***GetDelta*** = ***False***, если введен код ***Enter***, признак конца буксировки}

{***DeltaX = -1***, если нажата стрелка влево, ***DeltaX = 1***, если нажата стрелка вправо}

{***DeltaY = -1***, если нажата стрелка вверх, ***DeltaY = 1***, если нажата стрелка вниз}

function GetDelta(var DeltaX: Integer; var DeltaY: Integer): Boolean;

```
Type  {описание нестандартных типов данных}

{=== объект tLocation - координаты графического примитива ===}

   pLocation = ^tLocation;  {указатель на координаты графического примитива}
   tLocation =  object      {тип объекта - координаты графического примитива}
      X, Y: Integer; {поля с координатами  графического примитива}
      procedure Init(NewX, NewY: Integer);  {инициализация координат}
      function  GetX: Integer;              {определение текущей абсциссы}
      function  GetY:  Integer;             {определение текущей ординаты}
      procedure Draw;                       {нарисовать объект}
   end; {=== tLocation ===}

{=== объект tGrPoint -  графическая точка ===}

   pGrPoint = ^tGrPoint;          {указатель на графическую точку}
   tGrPoint =  object (tLocation) {тип объекта - графическая точка,
                                   наследуются от объекта tLocation }
      {поля X, Y с координатами точки наследуются от tLocation }
      Visible: Boolean;       {поле - индикатор светимости точки}
      constructor Init(NewX, NewY: Integer); {инициализация параметров точки}
      destructor Done; virtual;              {удаление}
      procedure Draw; virtual;               {точку нарисовать}
      procedure Hide; virtual;               {точку погасить}
      function  GetVisible: Boolean;         {проверка видимости точки}
      procedure MoveTo(NewX, NewY: Integer); {перемещение  в новые
                                              координаты NewX, NewY}
      procedure MoveToRel(RelX, RelY: Integer);{относительное перемещение на
                                              RelX, RelY}
      procedure Drag(DragBy: Integer); virtual; {буксировка объекта клавишами}
   end; {=== tGrPoint ===}


{=== объект tCircle -  окружность ===}

   pCircle = ^tCircle;            {указатель на окружность}
   tCircle = object (tGrPoint)    {тип объекта - окружность, наследуется от объекта
                                   tGrPoint }
   {поля X, Y, Visible – наследуются от tGrPoint }
      Radius: Integer;    {поле - радиус окружности}
```

```
    {инициализация параметров окружности}
    constructor Init(NewX, NewY: Integer; NewRadius: Integer);
    function GetRadius: Integer; {определение текущего радиуса}
    procedure Draw; virtual;        {окружность нарисовать}
    procedure Hide; virtual;        {окружность погасить}
    {динамическое увеличение радиуса окружности на ExpandBy }
     procedure Expand(ExpandBy: Integer); virtual;
    {динамическое уменьшение радиуса окружности на ExpandBy }
    procedure Contract(ContractBy: Integer); virtual;
end; {=== tCircle ===}

        {=== Раздел реализации объявленных процедур и функций ===}
implementation

  procedure StartGraphics; {инициализация графического режима}
  begin
    GraphDriver := Detect; {определение типа дисплея}
    DetectGraph(GraphDriver, GraphMode); {определение типа драйвера и
                                          видеорежима}
    {инициализация графического режима с установленными параметрами}
    InitGraph(GraphDriver, GraphMode, PathToDrivers);
    if GraphResult <> GrOK {проверка выполнения инициализации}
      then begin
            Writeln('Ошибка при инициализации графического режима:');
            Writeln(GraphErrorMsg(GraphDriver));
            if GraphDriver = grFileNotFound
              then begin
                    Writeln('В директории ', PathToDrivers,
                            ' нет драйвера графического режима');
                    Writeln('Скопируйте необходимый драйвер в эту
                             директорию..');
                    Writeln
                  end;
            Writeln('Нажмите Enter...');
            Readln;
            Writeln('Программа завершена...');
            Halt
```

```
                end
    end; { StartGraphics }
{-------------------------------------------------------------------------------------------}
{   Методы объекта tLocation (координаты графического объекта)          }
{-------------------------------------------------------------------------------------------}
    procedure   tLocation.Init(NewX,  NewY:  Integer);  {инициализация}
    begin
        X := NewX;
        Y := NewY
    end; { tLocation.Init }

    function tLocation.GetX: Integer; {определение текущей абсциссы}
    begin
        GetX := X
    end; { tLocation.GetX }

    function tLocation.GetY: Integer; {определение текущей ординаты}
    begin
        GetY := Y
    end; { tLocation.GetY }

    procedure tLocation.Draw; {нарисовать объект}
    begin
        Abstract   {Дает ошибку: координаты нарисовать нельзя!}
    end; { tLocation.Draw }

{-------------------------------------------------------------------------------------------}
{       Методы объекта tGrPoint (графическая точка)                    }
{-------------------------------------------------------------------------------------------}
    constructor tGrPoint.Init(NewX, NewY: Integer); {инициализация параметров}
    begin
        {инициализация координат, вызывается метод предка - объекта tLocation}
        inherited Init(NewX, NewY);
         Visible := False {точка погашена}
    end; { tGrPoint.Init }
```

```
destructor tGrPoint.Done; {удаление объекта}
begin {объект погасить и удалить из памяти}
   Hide
end; { tGrPoint.Done }

procedure tGrPoint.Draw; {точку нарисовать}
begin
   Visible := True;              {точка нарисована}
   PutPixel(X, Y, GetColor)      {нарисовать точку цветом по умолчанию}
end; { tGrPoint.Draw }

procedure tGrPoint.Hide; {точку погасить}
begin
   Visible := False;             {точка погашена}
   PutPixel(X,  Y,  GetBkColor)  {нарисовать точку фоновым цветом (скрыть)}
end; { tGrPoint.Hide }

function tGrPoint.GetVisible: Boolean; {проверка видимости точки}
begin
   GetVisible := Visible
end; { tGrPoint.GetVisible }

procedure tGrPoint.MoveTo(NewX, NewY: Integer); {Передвижение точки на
                                                 новое место}
begin
   Hide;                  {точку в текущих координатах погасить}
   X := NewX; Y := NewY;  {установить новые координаты точки}
   Draw                   {нарисовать точку в новых координатах}
end; { tGrPoint.MoveTo }

procedure tGrPoint.MoveToRel(RelX, RelY: Integer); {относительный перенос
                                                    на RelX, RelY}
begin
   Hide;                  {точку в текущих координатах погасить}
   X := X + RelX; Y := Y + RelY;  {установить новые координаты точки}
   Draw                   {нарисовать точку в новых координатах}
end; { tGrPoint.MoveToRel }
```

```
{Управление перемещением графического объекта с помощью клавиатуры}
  function GetDelta (var DeltaX: Integer; var DeltaY: Integer): Boolean;
  {DeltaX, DeltaY - коды перемещения по осям X и Y}
  var
    KeyChar: Char; {код символа, введенного с клавиатуры}
    Quit:    Boolean; {Флаг допустимости введенного символа управления}
  begin
    DeltaX := 0;  DeltaY := 0;   {Нет изменения положения точки}
    GetDelta := True;            {Буксировка точки началась}
    repeat
      KeyChar := ReadKey;              {Ввод первой части скан-кода клавиши}
      Quit := True;                {предполагаем, что это допустимая клавиша}
      case Ord(KeyChar) of {проверка первой части введенного скан-кода}
      0:  begin {если 0 - введен управляющий символ, используется
                 расширенный код}
            KeyChar := ReadKey; {Ввод второго байта расширенного кода
                                 клавиши}
            case Ord(KeyChar) of {проверка второй части введенного кода}
              72: DeltaY := -1;  {стрелка вверх, уменьшить Y}
              80: DeltaY :=  1;  {стрелка вниз, увеличить Y}
              75: DeltaX := -1;  {стрелка влево, уменьшить X}
              77: DeltaX :=  1;  {стрелка вправо, увеличить X}
            else Quit := False {игнорировать любую другую клавишу}
            end {case }
          end;
      13: begin
            GetDelta := False; {Введен код Enter, признак конца буксировки}
            Exit
          end
      else  Quit := False; {игнорируем любую другую клавишу}
      end;  {case}
    until Quit
  end; { tGrPoint.GetDelta }

  procedure  tGrPoint.Drag(DragBy: Integer); {буксировка объекта}
  {DragBy - шаг перемещения}
  var
```

```
    DeltaX, DeltaY: Integer;   {коды перемещения по осям X и Y}
  begin
    Draw; {засветить буксируемый объект}
    {цикл буксировки объекта}
    while GetDelta(DeltaX, DeltaY) do
    MoveToRel(DeltaX * DragBy, DeltaY * DragBy) {перемещаем объект в новую
                                                 позицию}
  end; { tGrPoint.Drag }

{-------------------------------------------------------------------------------}
{            Методы объекта tCircle (окружность)                                }
{-------------------------------------------------------------------------------}
  {Инициализация параметров окружности}
  constructor tCircle.Init(NewX, NewY: Integer; NewRadius: Integer);
  begin
    {инициализация координат, вызывается метод предка - объекта tGrPoint}
    inherited Init(NewX, NewY);
    Radius := NewRadius {устанавливается текущее значение радиуса}
  end; { tCircle.Init }

  function tCircle.GetRadius: Integer; {определение текущего радиуса}
  begin
    GetRadius := Radius
  end; { tCircle.GetRadius }

  procedure tCircle.Draw; {окружность нарисовать}
  begin
    Visible := True;              {окружность нарисована}
    Graph.Circle(X, Y, Radius) {нарисовать окружность цветом по умолчанию}
  end; { tCircle.Draw }

  procedure tCircle.Hide; {окружность погасить}
  var
    TempColor: Word; {для хранения текущего цвета окружности}
  begin
    TempColor := Graph.GetColor;   {запомнили текущий цвет объектов}
    Graph.SetColor(GetBkColor);    {установили текущий цвет окружности -
                                    фоновый}
```

```
    Visible := False;                {окружность погашена}
    Graph.Circle(X, Y, Radius);      {нарисовали окружность фоновым цветом}
    Graph.SetColor(TempColor)        {восстановили текущий цвет объектов}
  end;  { tCircle.Hide }
  procedure tCircle.Expand(ExpandBy: Integer); {перерисовать окружность с
                                                большим радиусом}
  {ExpandBy - шаг приращения радиуса окружности}
  begin
    Hide;                            {окружность погасить}
    Radius := Radius + ExpandBy;     {вычислим новое значение радиуса}
    if Radius < 0                    {проверка допустимости значения радиуса}
     then Radius := 0;               {уточнение значения радиуса}
    Draw                  {окружность нарисовать}
  end; { tCircle.Expand }

  procedure tCircle.Contract(ContractBy: Integer); { перерисовать окружность с
                                                     меньшим радиусом }
  {ContractBy - шаг уменьшения радиуса окружности}
  begin
    Expand(-ContractBy)
  end; { tCircle.Contract }

end.        {Конец модуля GraphObj }
```

8 Индивидуальные задания

8.1 Условия задания, требования к программе и рекомендации по ее разработке

Разработать программу с использованием модуля ***GraphObj***, позволяющую перемещать по экрану с помощью клавиш управления курсором фигуру, выполненную в соответствии со своим вариантом.

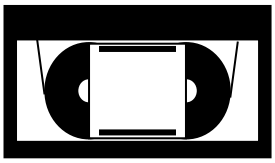
При составлении программы по индивидуальному заданию предусмотреть:

- использование структурного и модульного подхода;
- использование технологии объектно-ориентированного программирования;
- простейший диалог типа «запрос-ответ» при вводе данных, вывод справочной информации и т.п.

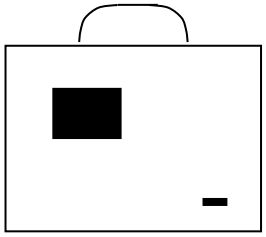
При выполнении задания в качестве заготовки программы разумно максимально использовать подходы, принятые в рассмотренном выше учебном примере. Для этого, в первую очередь, следует скопировать файлы ***DemoObj.pas*** и ***GraphObj.pas*** в свой каталог, переименовать их в соответствии со своим вариантом задания и взять затем за основу разрабатываемой программы.

8.2 Варианты заданий

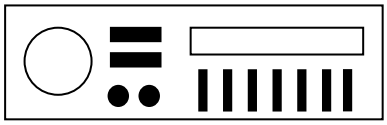
1



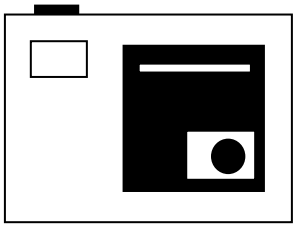
2



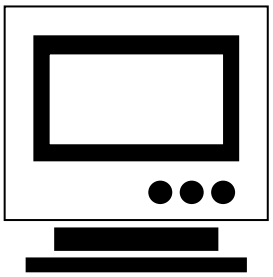
3



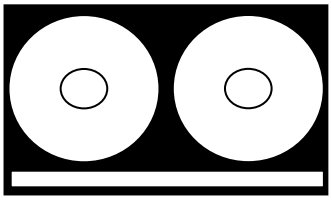
4



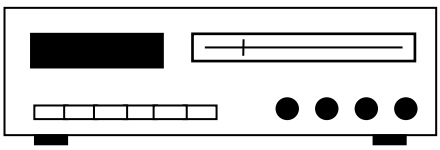
5



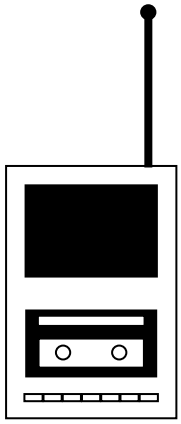
6



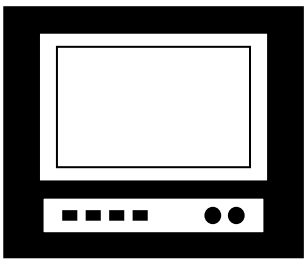
7



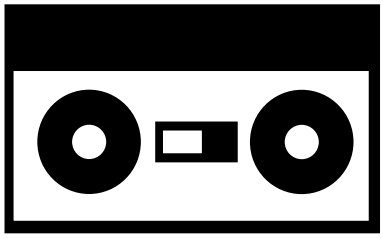
8



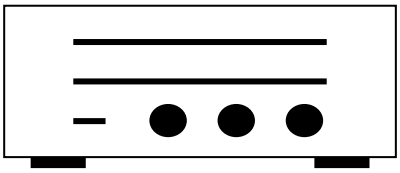
9



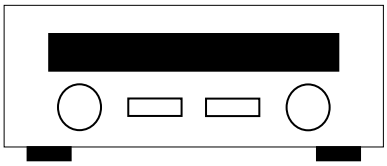
10



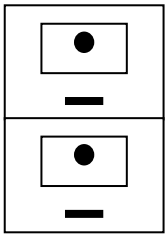
11



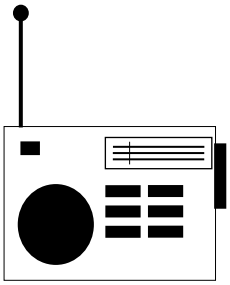
12



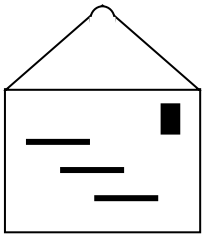
13



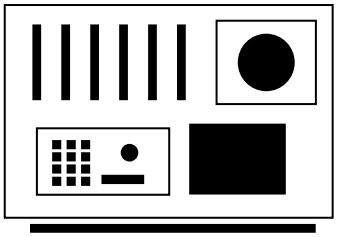
14



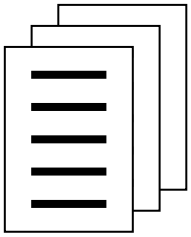
15



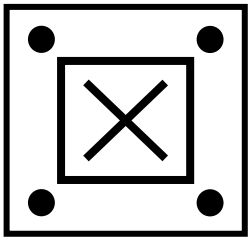
16



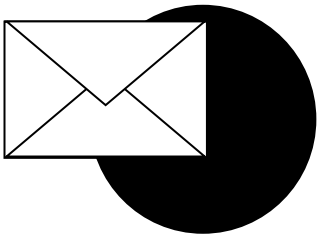
17



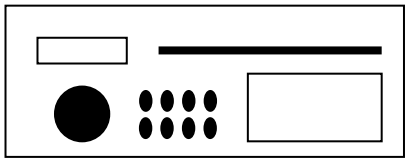
18



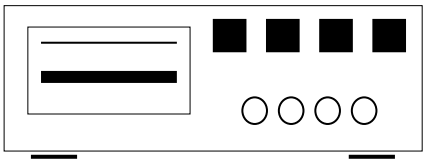
19



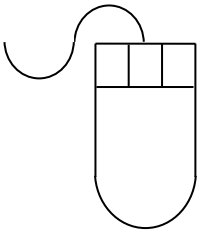
20



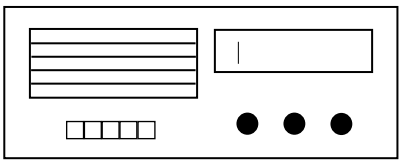
21



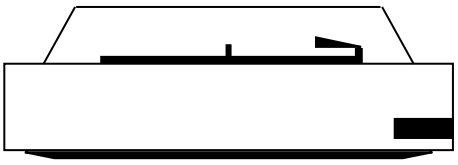
22



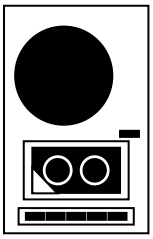
23



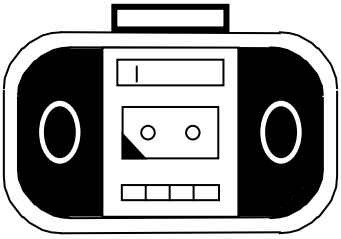
24



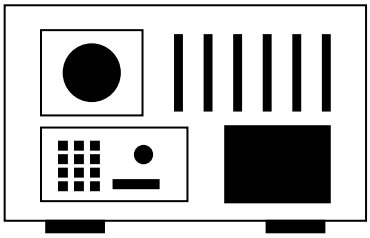
25



26



27



Список рекомендуемой литературы

1. Кобрин Ю.П. Работа с экраном в графическом режиме. Лабораторная работа по дисциплине «Информатика» для студентов специальностей 210201 (200800) и 201300. – Томск: ТУСУР, 2007. – 30 с.
2. Д. Ван Тассел. Стиль, разработка, эффективность, отладка и испытание программ: Пер. с англ. – М.: Мир, 1985. – 332 с.
3. Н. Вирт. Алгоритмы и структуры данных. : Пер. с англ. – М.: Мир, 1989. – 360 с.
4. Основы информатики. Учеб. Пособие / Аладьев В.З., Хунт Ю.Я., Шишаков М.Л. - М.: Информационно-издательский дом "Филинъ", 1998. - 496 с.
5. Марченко А.И., Марченко Л.А. Программирование в среде Borland Pascal 7.0. – К.: ЮНИОР, 1998. – 480 с.
6. Зуев Е.А. Программирование на языке Турбо Паскаль 6.0, 7.0. - М: Веста, Радио и связь, 1993. - 384 с.
7. Епанешников А., Епанешников В. Программирование в среде Turbo Pascal 7.0. - М.: "ДИАЛОГ-МИФИ", 1993. - 288 с.
8. Фаронов В.В. Turbo Pascal 7.0. Начальный курс. Учебное пособие.- М.: "НОЛИДЖ", 2001. - 576 с.
9. Фаронов В.В. Turbo Pascal 7.0. Практика программирования. Учебное пособие.- М.: "НОЛИДЖ", 1998. - 432 с.
10. ОС ТУСУР 6.1-97. Работы студенческие учебные и выпускные квалификационные. - Томск: ТУСУР, 1999.- 10 с.