4. Разработка простейшего объекта-контейнера

Контейнер – **объект**, позволяющий **хранить** и **обрабатывать** набор некоторых **объектов**. Возможные варианты реализации контейнера:

- на основе массива
- на основе динамических списков разных типов
- на основе поисковых деревьев
- на основе хеш-таблиц

Контейнеры полезны сами по себе, но кроме того дают хорошую практику по освоению объектного подхода. В качестве первого примера рассмотрим реализацию простейшего контейнера для графических объектов-окружностей на основе **классического** массива. В дальнейшем по мере освоения новых объектных механизмов будут приводиться и другие варианты реализации контейнеров.

Разработку контейнера будем проводить поэтапно:

- неформальное проектирование контейнера
- формальное описание в виде класса
- программная реализация методов
- создание тестовой программы для проверки работоспособности контейнера

Этап 1. Проектирование контейнера

Для того, чтобы описать контейнер как программный объект, необходимо определить:

- набор необходимых свойств и их типы
- набор конструкторов и их формальные параметры
- набор необходимых методов доступа
- набор специализированных методов
- форму каждого из методов с точки зрения входных и выходных параметров

Для реализации контейнера-массива минимально необходимы следующие **свойства**, которые в соответствии с принципом инкапсуляции необходимо **закрыть** от постороннего воздействия:

- массив объектных переменных-указателей на окружности
- счетчик текущего количества объектов-окружностей в контейнере

Минимально необходимый набор **открытых** методов:

- конструктор без параметров для создания пустого контейнера
- Get-метод для свойства-счетчика (а вот Set-метод не нужен, т.к. изменение счетчика должно происходить только при добавлении или удалении объектов)
- Get-метод для получения адреса окружности с заданным номером
- метод-функция для добавления нового объекта (для простоты будем добавлять новый объект только в конец имеющегося набора):
  входной параметр – адрес добавляемой окружности
  результат – успешность выполнения операции (массив, однако!)
- метод-функция для удаления объекта:
  входной параметр – номер удаляемой окружности
  результат – адрес удаленного объекта (нулевой в случае неудачи)
- метод-функция поиска объекта:
  входной параметр – радиус окружности
  результат – номер объекта в массиве и его адрес
- методы-итераторы для циклической обработки объектов в контейнере (например – показ или перемещение всех окружностей)

Этап 2. Формальное описание в виде класса

Описание контейнерного класса на языке Delphi/Free Pascal

```
TArrayCircleContainer = class
 private
  count : integer;                    // текущее число окружностей в контейнере
```

```
  Circs :  array  [1..100]  of  TCircle;  // массив указателей на окружности
public
  constructor  Create;   //  создаем пустой контейнер
  function  GetCount : integer;
  function  GetCirc (nom : integer) : TCircle;   // получение адреса окр-ти
  function  Add (aCirc : TCircle) : boolean;  // добавляем в конец набора
  function  Delete (ai : integer) : TCircle; //  удаление объекта по номеру
  function  Search (aRad : integer;  var  Nom : integer) : TCircle;  // поиск
  procedure  ShowAll;    //  метод-итератор для показа всех окружностей
  procedure  MoveAll (dx, dy : integer);  // еще один итератор
end;
```

Этап 3. Реализация методов

Далее приводится реализация только некоторых (основных) методов, тогда как другие (очевидные) оставлены для самостоятельной работы. Кроме того, данная реализация, как это часто бывает в программировании, является лишь одной из возможных.

```
constructor  TArrayCircleContainer.Create;
  var   i : integer;
  begin
    for  i := 1  to  100  do  Circs [ i ] := nil;   // массив пустых указателей
    count := 0;
  end;

function  TArrayCircleContainer.Add (aCirc : TCircle) : boolean;
  begin
    result := false;   // если добавление будет невозможно
    if   ( count < 100 )   then
          begin
            count := count + 1;    // а лучше так:   inc(count);
```

```
            Circs [ count ] := aCirc;
            result := true;  //  добавление выполнено
          end;
  end;

function  TArrayCircleContainer.Delete (ai : integer) : TCircle;
  begin
    result := nil;   //  если удаление не будет выполнено
     if   ( count > 0 ) and ( ai <= count )   then
          begin
            result := Circs [ ai ];   // возвращаем адрес удаляемого объекта
            count := count – 1;   // а лучше так:   dec (count);
            «сдвиг хвостовой части массива влево»
          end;
  end;

function TArrayCircleContainer.Search (aRad : integer;  var  Nom : integer) :
TCircle;
var   i : integer;
begin
     result := nil;
     if   ( count > 0 )   then
       for   i := 1   to  Count   do
          if  ( Circs [ i ].GetR = aRad )    // радиус - через метод доступа
          then   begin
                   Nom := i;
                   result := Circs [ i ];
                   break;
                 end;
end;
```

```
function  TArrayCircleContainer.GetCirc (nom : integer) : TCircle;
  begin
    result := nil;
    if   ( count > 0 ) and (nom <= count)  then
       result :=  Circs [nom];
  end;

procedure TArrayCircleContainer.ShowAll;
var   i : integer;
begin
    for   i := 1   to  count   do
      if  ( Circs [ i ] <> nil )    then  Circs [ i ].Show;
end;
```

Этап 4. Демонстрация использования

1. **Объявить** объектную переменную контейнерного типа:

```
var  MyCont : TArrayCircleContainer;
```

2. **Создать** пустой контейнер с помощью конструктора:

```
MyCont := TArrayCircleContainer.Create;
```

3. **Добавить** в контейнер необходимые объекты-окружности за счет повторения следующих операций:

```
MyCirc := TCircle.Create(random(…), random(…), random(…));
if   MyCont.Add (MyCirc)   then  «добавили»   else  «нет места» ;
```

4. Циклически **обработать** контейнер:

```
MyCont.ShowAll;
MyCont.MoveTo (…);
```

5. **Удалить** ненужные окружности

if (MyCont.Delete(2) = **nil**) **then** «удаление невозможно»

else «удалили окружность 2»

6. **Найти** окружность по радиусу (например – 50):

MyCirc := MyCont.Search(50, N);

if (MyCirc = **nil**) **then** «не нашли» **else** «нашли под номером N»;

7. **Изменить** радиус окружности с заданным номером:

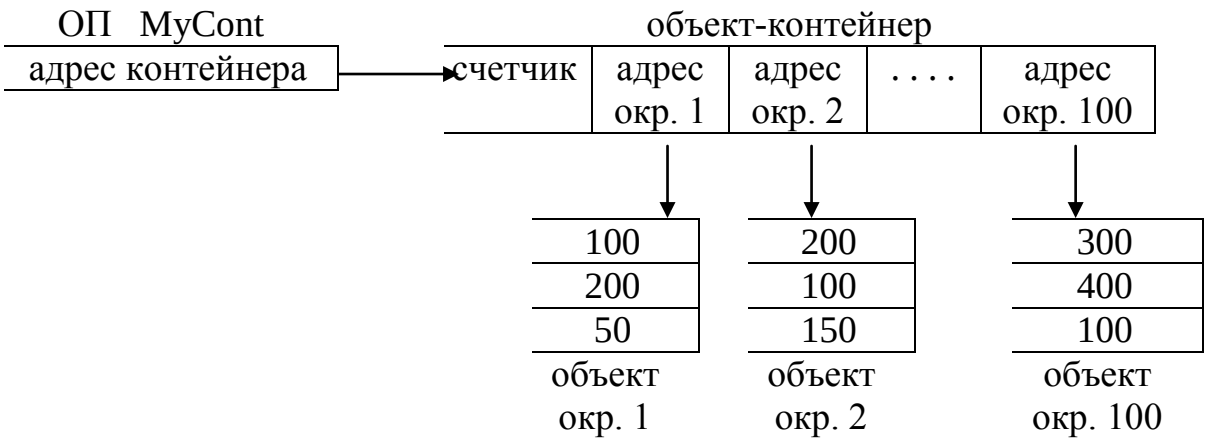
MyCirc := MyCont.GetCirc(Nomer);

if (MyCirc <> **nil**) **then** MyCirc.SetR(100);

При выполнении этой программы будут выделены следующие области памяти:

- для объектной переменной контейнерного типа
- для самого объекта-контейнера
- для включенных в контейнер объектов-окружностей

Схематично взаимосвязи между этими областями можно представить так:



Одно из важнейших достоинств объектной реализации состоит в том, что разработчик классов может вносить изменения в свои классы «**незаметно**» от пользователей, т.е. без необходимости переписывания прикладных программ. В частности, простейший контейнер-массив можно улучшить, заменив обычный массив его **динамической** версией.

Особенность динамических массивов – это возможность **изменять размер** во время **выполнения** программы.

Переход на новую версию контейнера требует небольших изменений в структуре классов и реализации методов:

- изменяется **объявление** массива
- вводится новое закрытое свойство – **текущий размер** динамического массива и Get-метод доступа к нему
- старый конструктор без параметров: изменяется его реализация с целью создания массива с установленным **по умолчанию** начальным размером
- вводится **второй** конструктор с одним **входным** параметром, определяющим **начальный** размер массива
- изменяется реализация метода **добавления**: при отсутствии свободного места в текущем массиве происходит его **динамическое расширение**; в этом случае отказ в добавлении происходит только при отсутствии необходимой свободной динамической памяти
- немного изменяется реализация **циклов** при обработке массивов, поскольку динамические массивы адресуются с **нуля**

При работе с динамическими массивами надо учитывать следующие **особенности** их использования. **Во-первых,** переход к новому массиву другого размера – это **выделение новой области** памяти и **занесение** в нее данных из старого массива. Выполнение этих операций требует некоторого **времени**. С одной стороны, наиболее эффективное использование памяти достигается при изменении размера на один элемент, но с другой стороны **многократное** повторение таких операций может существенно **замедлить** выполнение программы. Поэтому приходится идти на «вечный» **компромисс** память-время и ради скорости выполнения немного пожертвовать памятью. На практике хорошо показала себя следующая **рекомендация**: увеличить размер массива на 50-100 % от текущего значения.

Во-вторых, в отличие от классических массивов, переменная, определяющая имя массива, является **указателем** на область памяти с элементами этого массива. Для создания нового массива в Delphi/Free Pascal используется стандартный вызов

SetLength (имя_массива, размер_массива);

Формальное объявление класса:

```
TDynArrayCircleContainer = class
private
  Circs :  array  of  TCircle;     // динамический (!) массив указателей
  count : integer;              // не изменяется
  size : integer;          //  новое свойство – текущий размер массива
public
  constructor  Create; overload;  //  старая форма – новое содержание
  constructor  Create (aSize: integer);  overload; //  новый конструктор
  function  GetCount : integer;   //  не изменяется
  function  GetSize : integer;   // новый метод доступа
  function  GetCirc (nom : integer) : TCircle;   // не изменяем
  function  Add (aCirc : TCircle) : boolean;  // измененный метод
  function  Delete (ai : integer) : TCircle;    // можно не изменять
  function  Search (aRad : integer;  var  Nom : integer) : TCircle;  // немного
  procedure  ShowAll;                                                // изменяем
  procedure  MoveAll (dx, dy : integer);                             // циклы
end;
```

Программная реализация нового конструктора:

```
constructor  TDynArrayCircleContainer.Create (aSize : integer);
  var   i : integer;
  begin
    SetLength (Circs, aSize);   // динамическое создание массива размера aSize
```

```
    size := aSize;                  // запомнили начальный размер массива
    for  i := 0  to  (size-1)  do  Circs [ i ] := nil;  // заполнили нулями
    count := 0;
  end;
```

Измененный метод добавления:

```
function  TDynArrayCircleContainer.Add (aCirc : TCircle) : boolean;
  begin
    result := true;
    if   ( count = size )   then                // места нет, надо расширить массив
    begin
      size := size * 2;                              // увеличиваем размер в два раза
      SetLength (Circs, size);       // создаем новый массив большего размера
          // если памяти для нового массива нет, установить  result := false;
    end;
    count := count + 1;          // лучше так:   inc (count);
    Circs [ count ] := aCirc;
  end;
```

Использовать новую версию контейнера можно практически без каких-либо изменений в прикладной программе, поскольку открытый интерфейс класса почти не изменился.

Контейнер на основе **динамического** массива для **объектов-студентов**:

```
TDynArrayStudContainer = class
 private
  Name : string;                        // информационное имя контейнера
  Studs :  array   of  TStudent;        // дин. массив ссылок на студентов
```

```
  count : integer;                    // число студентов
  size : integer;                     //  текущий размер дин. массива
public
  constructor  Create (aSize: integer); // создание контейнера заданн. размера
  function  GetCount : integer;         // запрос текущего числа студентов
  function  GetName : string;           // запрос имени
  procedure  SetName (newName : string);        // изменение имени
  procedure  Add (newStud : TStudent);          // добавление студента
  function  Delete (aFam: string) : boolean;    // удаление студента по фамилии
  function  GetStud (nom : integer) : TStudent;  // получение ссылки на студ-та
  function  Search (aFam : string) : integer;     // поиск студента по фамилии
  procedure  SortByFam;                          //  сортировка по фамилии
  procedure  SortBySredBall;                    // сортировка по среднему баллу
end;
```

Фрагменты тестовой (пользовательской) программы:

1. **Объявить** объектную переменную контейнерного типа:

```
var   Gruppa : TDynArrayStudContainer;
```

2. **Создать** пустой контейнер заданного размера с помощью конструктора:

```
Gruppa := TDynArrayStudContainer.Create (20);
```

3. **Добавить** в контейнер объекты-студенты:

```
Stud := TStudent.Create('Иванов');
Gruppa.Add (Stud);
```

4. Циклически **обработать** контейнер:

```
Gruppa.SortBySredBall;
```

5. **Удалить** студента:

if (Gruppa.Delete('Петров')) **then** «удален» **else** «нет такого»

6. **Найти** студента по фамилии:

Nomer := Gruppa.Search('Козин');

if (Nomer = 0) **then** «не нашли» **else** «нашли под номером Nomer» ;

7. **Изменить** фамилию студента с заданным номером:

Stud := Gruppa.GetStud(Nomer); // сначала получаем ссылку на студента

if (Stud <> **nil**) **then** Stud.SetFam('Сидоров'); // а теперь используем ее

8. **Вывести** средний балл студента с заданным номером:

if (Stud <> **nil**) **then** **Writeln**(Stud.SredBall); // аналогично

Для сравнения приведем **описание контейнерного класса на языке C#**, отметив заодно, что практически также выполняется описание на языке **Java**. Напомним, что в этих языках массивы являются **объектами** и поэтому **всегда** создаются динамически.

```
class ArrayCircleContainer {
  private Circle[ ] Circs;     // будущий массив указателей
  private int Count;           // счетчик числа объектов в контейнере

  public ArrayCircleContainer (int aSize) {
          Circs = new Circle[aSize];  // создание массива
          Count = 0; }
  public int GetCount { return Count;}
  public int GetSize { return Circs.Length; }
  public bool Add (Circle aCirc) { // код добавления новой окружности}
  public Circle Delete (int ai) { // код удаления окружности по ее номеру}
  public Circle Search (int aRad) { // код поиска по радиусу }
  public void ShowAll( ) { // код отображения всех окружностей }
  public void MoveAll (int dx, int dy) { // код перемещения }
```

```
}          //  конец описания класса
```

Фрагмент демонстрационной программы:

```
class   ContDemo {
  static  void  Main ( )  {   //   точка входа в программу
    ArrayCircleContainer   MyCont = new  ArrayCircleContainer (100);
    Circle  MyCirc = new  Circle (. . . . .);
    MyCont.Add (MyCirc);
    //  повторить два последних действия необходимое число раз
    MyCont.ShowAll ( );
    MyCont.MoveAll (50, 50);
    . . . . . .
    }
}
```

Пример реализации контейнерного объекта интересен еще и тем, что объектная программа использует объекты **двух** типов — сами хранящиеся в контейнере объекты и собственно объект-контейнер. Тем самым в программе реализуется **взаимодействие** объектов, что является предметом детального обсуждения в последующих темах пособия. Но уже сейчас можно привести **схематичное** представление такого взаимодействия, и для этих целей удобно воспользоваться средствами **языка моделирования UML**, в частности — **диаграммой классов**. Подробное рассмотрение этого языка выходит за рамки пособия, поэтому ограничимся **упрощенной** формой диаграммы классов, причем в качестве примера возьмем классы «Студент» и «Группа».

Каждый класс представим в виде прямоугольника, содержащего три области — название класса, перечень его свойств, перечень методов. Для обозначения связи этих классов используется стрелка от класса «Студент» к классу «Группа» с указанием возможного числа студентов в группе.

Этот простой пример показывает одну из самых замечательных особенностей объектного подхода — четкое **распределение** функциональных **обязанностей** между объектами. Объекты класса «Студент» отвечают за информацию, связанную только с **одним** конкретным студентом (его фамилия и оценки), а объекты класса «Группа» реализуют **другую** функциональность, такую как добавление и удаление студентов, обработку сводных данных в разрезе группы.

Упрощенная диаграмма классов для задачи «Студенты — Группа» имеет следующий вид: