Конструкторы и Деструкторы

Полиморфизм является необходимым средством при обеспечении расширяемости типов в приложениях.

Полиморфизм позволяет не переделывать структуру программы

- *Если возникает необходимость определения нового типа данных, например нового типа геометрической фигуры*
- *Использование полиморфизма необходимо, если процедура построения фигур работает с фиксированным набором объектов, описанных в модуле, а пользователь, работающий с модулем, хочет определить новый тип фигуры*

Для этого применяются виртуальные методы.

Метод становится виртуальным, если за его объявлением в типе объекта стоит зарезервированное слово **VIRTUAL**.

Виртуальные методы

Необходимо помнить, что если метод объявлен в родительском типе как ***VIRTUAL****, то все методы с аналогичными именами в дочерних типах также должны объявляться виртуальными, во избежание ошибки компилятора.*

Например:

```
UNIT New_persons;
INTERFACE
TYPE
  PPerson = ^TPerson;
  TPerson = OBJECT
    Name : STRING[30];
    Date   : STRING[10];
    Rate   : REAL;
    CONSTRUCTOR Init(Nm,Dt:STRING;                    Rt:REAL);
    DESTRUCTOR Done;       VIRTUAL;
    PROCEDURE ShowAll;     VIRTUAL;
  END;

  PStudent = ^TStudent;
  TStudent = OBJECT(TPerson)
    Ball    : REAL;
    CONSTRUCTOR Init(Nm,Dt:STRING;                Rt,Bl:REAL);
    DESTRUCTOR Done;          VIRTUAL;
    FUNCTION GetSum : REAL; VIRTUAL;
    PROCEDURE ShowSum;      VIRTUAL;
    PROCEDURE ShowAll;        VIRTUAL;
  END;
PStaff = ^TStaff;
TStaff = OBJECT(TPerson)
  Bonus   : REAL;
  CONSTRUCTOR Init(Nm,Dt:STRING;                  Rt,Bn:REAL);
  DESTRUCTOR Done;    VIRTUAL;
  FUNCTION GetSum : REAL; VIRTUAL;
  PROCEDURE ShowSum;      VIRTUAL;
```

Обратите внимание, что метод ***ShowSum***, показанный для типа ***TTeacher***, теперь удален из его определения.

- Типу ***TTeacher*** уже не нужно переопределять метод ***ShowSum*** типа ***TStaff***.
- Вместо этого ***ShowSum*** может просто наследоваться от ***TStaff*** со всеми вложенными в него вызовами, которые в этом случае уже будут вызывать методы из ***TTeacher***, а не из ***TStaff***, как это происходило в полностью статической иерархии объектов.

*Обратите внимание на зарезервированное слово **CONSTRUCTOR** (конструктор), заменившее зарезервированное слово **PROCEDURE** для процедур **Init**.*

- Конструктор является специальным типом процедуры, которая выполняет некоторую установочную работу для механизма виртуальных методов.

Конструктор должен вызываться перед вызовом любого виртуального метода.

- Вызов виртуального метода без предварительного вызова конструктора может привести к блокированию системы.
- У компилятора нет способа проверить порядок вызова методов.

Виртуальные методы

Каждый тип объекта, имеющий виртуальные методы, обязан иметь конструктор.

- Понятие **DESTRUCTOR** (деструктор), обратное понятию **CONSTRUCTOR**, будет объяснено далее.

Каждый экземпляр объекта должен инициализироваться отдельным вызовом конструктора.

- Недостаточно инициализировать один экземпляр объекта и затем присваивать этот экземпляр другим.
- Другие экземпляры, даже если они содержат правильные данные, не будут инициализированы оператором присваивания и заблокируют систему при любых вызовах их виртуальных методов.

Виртуальные методы

Например:

```
VAR
    One,Two:   TPerson;
BEGIN
    One.Init('Петр Петров','25-06-1995' ,400000);
    Two  := One;   {Неправильный вызов!}
END;
```

Каждый тип объекта, содержащий виртуальные методы, имеет таблицу виртуальных методов (ТВМ), хранящуюся в сегменте данных.

- ТВМ содержит размер типа объекта и для каждого виртуального метода указатель кода, исполняющий данный метод.
- Конструктор устанавливает связь между

Необходимо учесть, что имеется только одна ТВМ для каждого типа объекта.

- Отдельные экземпляры объекта (т.е. переменные Данного типа) содержат только адрес ТВМ, но не саму ТВМ.

Конструктор устанавливает значение адреса ТВМ.

- Вызов виртуального метода до вызова конструктора приведет к ошибке т.к. к этому моменту поле адреса ТВМ еще не инициализировано и содержит неопределенный адрес.

Виртуальные методы

При отладке программы для контроля правильности вызовов виртуальных методов можно использовать директиву компилятора ***$R***.

- Если директива **$R** находится во включенном состоянии **{$R+};** то все вызовы виртуальных методов будут проверяться на состояние инициализации объекта, выполняющего вызов метода.
- Если выполняющий вызов объект еще не был

Отрицательной стороной использования данной директивы является замедление работы программы.

- Это происходит из-за того, что при каждом вызове виртуального метода дополнительно выполняется процедура проверки правильности инициализации.

Если скорость работы программы является критическим параметром, то рекомендуется использовать проверку виртуальных методов только на этапе отладки.

Виртуальные методы

Как только родительский тип объекта объявит метод виртуальным, все его потомки также должны объявить этот метод виртуальным

Статический метод никогда не может переопределить виртуальный метод.

- Если Вы попытаетесь сделать это, компилятор выдаст сообщение об ошибке.

После того, как метод стал виртуальным, его заголовок не может изменяться в объектах более низкого уровня иерархии.

- Определение виртуального метода можно представить как шаблон для всех родственных ему методов.

Одноименный метод, отличающийся от виртуального метода предка, может иметь другое число параметров и другие типы при

идентичными, включая число параметров и их типы.

Виртуальные методы

Преимуществом использования виртуальных методов является то, что типы объектов и методы, определенные в модуле, могут поставляться пользователю в виде TPU-файла, т.е. без исходного кода.

- Для работы с объектами модуля необходимо знать содержание только интерфейсной части модуля.
- Используя полиморфные объекты и виртуальные методы, пользователь TPU-файла может свободно добавлять новые методы к уже существующим.

Новое понятие, связанное с добавлением новых функциональных характеристик в программу без модификации ее исходного кода, называется способностью к расширению.

Способность к расширению является естественным продолжением наследования

- Наследуются все свойства, которыми обладают порождающие типы, а затем добавляются новые по мере необходимости.
- Позднее связывание позволяет новые методы связать с уже существующими во время выполнения программы, благодаря чему расширение существующего кода выглядит как бы невидимым, требуя лишь небольшого увеличения таблицы виртуальных методов.

Расширяемость объектов

Преимущества и недостатки виртуальных методов

Рекомендуется делать методы виртуальными.

- В таких случаях выполнения и использования памяти. Однако
- Каждый вызов виртуального метода должен проходить через ТВМ, тогда как статические
- Дополнительная скорость и эффективное
- Хотя просмотр ТВМ весьма эффективен, вызов

получить оптимальную эффективность скорости

лучше определить его виртуальным, особенно, если имеется предположение, что метод будет перекрываться кем-либо из потомков, а его методы непосредственно

Иногда не известно, является метод виртуальным или нет.

Если у объекта есть виртуальные методы, то для него будет создана ТВМ и любой экземпляр этого объекта будет с ней связан.

Всегда необходимо учитывать возможность последующей модификации программы.

методов должны уравновешивать ся гибкостью, которая присуща виртуальным методам, ведь имеющийся код можно расширить спустя большой промежуток времени с

Точно так же, как и любые типы данных в Паскале, объекты можно размещать в динамической памяти и работать с ними, применяя указатели.

Одним из самых простых способов размещения объектов в памяти является использование процедуры ***New****, традиционно применяемой для работы с указателями:*

```
VAR
    Sum    :   REAL;
    P      : ^TPerson;
...
New(P);
```

Если динамический объект содержит виртуальные методы, он должен инициализироваться с помощью вызова конструктора, перед тем как будет вызван любой из его методов:

```
P^.Init('Иван Петров','25-06-1995' ,40000);
```

Затем вызовы методов могут происходить в обычном порядке, с использованием имени указателя и ссылочного символа ^ вместо имени экземпляра объекта, которые использовались бы при обращении к статически размещенному объекту:

```
Sum  := P^.GetSum;
```

Специально для работы с динамическими объектами Турбо Паскаль включает несколько усовершенствованных процедур для размещения и удаления объектов из памяти наиболее эффективными способами.

*Процедура **New** может вызываться с двумя параметрами: имя указателя используется в качестве первого параметра, а имя конструктора — в качестве второго параметра:*

```
New(P, Init ('Иван Петров','25-06-1995',40000));
```

Расширенное использование оператора New.

*При использовании расширенного синтаксиса процедуры **New** конструктор **Init** выполняет динамическое размещение объекта, используя специальный сгенерированный код, вызываемый оператором **CONSTRUCTOR** и выполняемый до основного кода конструктора.*

*Имя экземпляра объекта не может использоваться в качестве первого параметра процедуры, т.к. во время вызова процедуры **New** экземпляр, инициализируемый с помощью **Init**, еще не существует.*

Компилятор определяет правильность вызова конструктора, проверяя тип указателя, передаваемого в качестве первого параметра.

Расширенное использование оператора New.

Процедура **New** также может использоваться в качестве функции, которая возвращает значение указателя.

*Передаваемый **New** параметр на этот раз должен быть типом указателя на объект, а не самим указателем:*

```
TYPE
     PPerson = ^TPerson;
VAR
     P : PPerson;
. . . .
P:= New(PPerson);
```

*В качестве второго параметра процедура **New** может содержать конструктор объектного типа:*

```
P:=New(PPerson,Init('Иван Петров','25-06-1995',40000));
```

Расширенное использование оператора New.

Турбо Паскаль позволяет установить пользовательскую функцию обработки ошибок динамической памяти с помощью переменной **HeapError**, которая является стандартной и не требует описания в разделе переменных.
Она содержит адрес стандартной функции обработки ошибок в Паскале, которая может быть замещена.

Пользовательская функция обработки ошибок должна иметь формат:

```
FUNCTION HeapFunc(Size:Word): INTEGER;   FAR;
```

Наличие директивы **FAR** обязательно

Новая функция обработки ошибок устанавливается путем присваивания ее адреса переменной ***HeapError*** *следующим образом:*

```
HeapError:= @HeapFunc;
```

Такая возможность может оказаться полезной при использовании конструкторов.

*По умолчанию, если не хватает памяти для размещения экземпляра динамического объекта, вызов конструктора, использующий расширенный синтаксис стандартной процедуры **New**, генерирует фатальную ошибку выполнения с кодом 203.*

- Программа прекращает свою работу.

*Если устанавливается пользовательская функция обработки ошибок динамической памяти, которая возвращает 1, а не стандартный результат 0, то вызов конструктора через **New** будет возвращать **NIL** в том случае, если конструктор не сможет завершить запрос .*

- Программа продолжает работать.

Обнаружение ошибок конструктора

Обнаружение ошибок конструктора

Обнаружение ошибок конструктора

*Для реализации изложенного выше механизма, Турбо Паскаль предоставляет новую стандартную процедуру **Fail**, которая не имеет параметров и которая может вызываться только изнутри конструктора.*

- Вызов **Fail** заставляет конструктор удалить динамический экземпляр, который был размещен при входе в конструктор, и приводит к возврату указателя **NIL** для индикации неудачной попытки.

Обнаружение ошибок конструктора

*Если динамические экземпляры размещаются с помощью расширенного синтаксиса **New**, то результирующее значение **NIL**, передаваемое указателю, свидетельствует о неудачной операции.*

- Но нет таких переменных типа указатель, которые можно было бы проверить после создания статического экземпляра или после вызова унаследованного конструктора.

*Турбо Паскаль в качестве функций позволяет использовать конструкторы, которые возвращают результат типа **BOOLEAN**.*

- Возвращаемое значение **TRUE** означает успех, а **FALSE** — неудачу, благодаря вызову **Fail** внутри конструктора.

Обнаружение ошибок конструктора

Рассмотрим, как можно описать последовательный вызов конструкторов типа ***TPerson***, ***TStaff*** *и* ***TTeacher***.

пример конструкторов не использующих обнаружение ошибок:

```
CONSTRUCTOR TPerson.Init(Nm,Dt : STRING; Rt : REAL);
BEGIN
      Name:= Nm;
      Date:= Dt;
      Rate:= Rt;
END;

CONSTRUCTOR TStaff.Init(Nm,Dt : STRING; Rt,Bn : REAL);
BEGIN
      TPerson.Init(Nm,Dt,Rt) ;
      Bonus:= Bn;
END;

CONSTRUCTOR TTeacher.Init(Nm,Dt:STRING; Rt,Bn,Hrt:REAL; Hr:WORD);
BEGIN
      TStaff.Init(Nm,Dt,Rt,Bn);
      Hours:= Hr;
      HourRate:= Hrt;
END;
```

*В случае инициализации экземпляр типа **TTeacher**, он вызывает конструктор типа **TStaff**, который в свою очередь вызывает конструктор типа **TPerson**. Если произойдет ошибка при вызове последнего конструктора, придется отменить вызовы всех трех конструкторов.*

можно переписать конструкторы типа ***TStaff*** *и* ***TTeacher*** *с учетом обнаружения ошибок:*

```
CONSTRUCTOR TStaff.Init(Nm,Dt : STRING; Rt,Bn : REAL);
BEGIN
      IF NOT TPerson.Init(Nm,Dt,Rt) THEN Fail;
      Bonus := Bn;
END;

CONSTRUCTOR TTeacher.Init (Nm,Dt: STRING; Rt,Bn,Hrt:REAL; Hr:WORD);
BEGIN
      IF NOT TStaff.Init(Nm,Dt,Rt,Bn) THEN Fail;
      Hours:= Hr;
      HourRate:= Hrt;
END;

FUNCTION HeapFunc(Size: WORD): INTEGER; FAR;
BEGIN
      HeapFunc:= 1;
END;
```

Обратите внимание на то, что функция ***HeapFunc*** *выполняет одну единственную операцию: при любом вызове возвращает* ***1****.*
В выполнении других операций в данном случае нет необходимости.

В приведенных примерах вложенные вызовы конструкторов осуществляются путем указания имени предка, за которым следует точка и имя конструктора.

*Турбо Паскаль предоставляет специальное слово **INHERITED**, используя которое, можно вызывать методы предка без указания имени типа предка и точки*

- Например:

```
CONSTRUCTOR TStaff.Init(Nm,Dt:STRING; Rt,Bn:REAL);
BEGIN
     IF NOT INHERITED Init(Nm,Dt,Rt)   THEN Fail;
     Bonus:= Bn;
END;
```

*Это может
когда запомн*

Подобно другим типам данных, размещаемые в динамической памяти объекты могут удаляться в случае необходимости с помощью процедуры ***Dispose***

Однако при удалении ненужного объекта может понадобиться нечто большее, чем простое освобождение занимаемой им динамической памяти.

- Объект может содержать указатели на динамические структуры или объекты, которые нужно освободить или очистить в определенном порядке, особенно если используется сложная динамическая структура данных.

Все операции, необходимые для очистки динамического объекта, должны объединяться в один метод таким образом, чтобы объект мог быть уничтожен с помощью единственного вызова: ***Person.Done****;*

Деструкторы

Метод ***Done*** должен инкапсулировать все детали очистки своего объекта, а также всех структур данных и вложенных объектов.

- Для обозначения таких методов обычно рекомендуется использовать идентификатор ***Done***.

Часто полезно и допустимо определять несколько методов очистки для данного типа объекта.

- В зависимости от их размещения или использования, или в зависимости от состояния и режима на момент очистки сложные объекты могут потребовать очистки несколькими различными

Турбо Паскаль предоставляет специальный тип метода, называемый "сборщиком мусора" или деструктором, для очистки и удаления динамических объектов.

- Деструктор объединяет этап удаления объекта с другими действиями или задачами, необходимыми для данного типа объекта.
- Для одного типа объекта можно определить несколько деструкторов.

Деструкторы

Деструктор размещается вместе с другими методами объекта в определении типа объекта:

```
TYPE
    TPerson = OBJECT
        Name: STRING[30];
        Date: STRING[10];
        Rate: REAL;
        CONSTRUCTOR Init(Nm,Dt:STRING; Rt:REAL);
        DESTRUCTOR Done; VIRTUAL;
        PROCEDURE ShowAll; VIRTUAL;
    END;
```

Деструкторы можно наследовать, и они могут быть либо статическими, либо виртуальными.

- Поскольку различные программы завершения обычно требуют различных типов объектов, рекомендуется всегда определять деструкторы виртуальными, чтобы для каждого типа объекта выполнялся правильный деструктор.

Нет необходимости указывать зарезервированное слово **DESTRUCTOR** для каждого метода очистки, даже если определение типа объекта содержит виртуальные методы.

- Деструкторы, в действительности, работают только с динамическими объектами.
- При очистке таких объектов деструктор осуществляет некоторые специальные функции, гарантируя, что в динамической памяти всегда будет освобождаться правильное

Применение деструктора к статическим объектам также не является ошибкой и не может привести к некорректной работе программы.

Основное преимущество использования деструктора заключается в удалении из памяти полиморфных объектов.

Полиморфными являются те объекты, которые были присвоены экземпляру родительского типа, благодаря правилам совместимости типов.

- Экземпляр объекта типа ***TStudent***, присвоенный переменной типа ***TPerson***, является примером полиморфного объекта.

Эти правила могут применяться и к указателям на объекты

- Указатель на ***TStudent*** может свободно присваиваться указателю на ***TPerson***, а адресуемый им объект также будет полиморфным.

Термин "полиморфный" означает, что компилятор, строя код объекта, во время компиляции точно не знает, какой тип объекта будет в действительности использован.

- Единственное, что он знает, это то, что объект принадлежит иерархии объектов, являющихся потомками указанного типа предка.

Очевидно, что размеры типов объектов отличаются.

- Во время компиляции из полиморфного объекта нельзя извлечь какую-либо информацию о его размере.

Информация о размере удаляемого объекта становится доступной для деструктора в момент удаления, благодаря обращению к тому месту, где эта информация записана, т.е. к таблице виртуальных методов экземпляров объектов определенного типа.

- В каждой ТВМ содержится размер в байтах данного типа объекта.

*Таблица виртуальных методов любого объекта доступна через скрытый параметр **Self**, содержащий адрес ТВМ, который передается методу при вызове.*

- Деструктор также является методом, поэтому, когда объект вызывает его, деструктор получает копию ***Self*** через стек.

Таким образом, если объект является полиморфным во время компиляции, он никогда не будет полиморфным во время выполнения, благодаря позднему связыванию.

Для выполнения освобождения памяти при позднем связывании деструктор нужно вызывать как часть расширенного синтаксиса процедуры ***Dispose:***

Dispose(P,Done);

Деструкторы

Вызов деструктора вне процедуры ***Dispose*** *не приведет к автоматическому освобождению памяти.*

Сам по себе метод деструктора может быть пустым и выполнять только функцию связи с процедурой ***Dispose,*** *поскольку основная информация содержится не в теле деструктора, а связана с его заголовком, содержащим слово* ***DESTRUCTOR****.*

```
DESTRUCTOR TPerson.Done;
BEGIN
END;
```

Деструктор дочернего типа должен последним действием вызывать соответствующий деструктор своего непосредственного предка, чтобы освободить поля всех наследуемых указателей объекта

```
DESTRUCTOR TStaff.Done;
BEGIN
        INHERITED Done;
END;
```

Полезная информация

В Турбо Паскале имеется дополнительный класс методов позднего связывания, которые называются динамическими.

- Фактически, динамические методы являются подклассом виртуальных методов и отличаются от них только способом вызова на этапе выполнения.
- Во всех других отношениях динамический метод можно рассматривать как эквивалентный виртуальному.

*Описание динамического метода аналогично описанию виртуального за исключением того, что оно должно включать в себя индекс динамического метода, который указывается сразу за ключевым словом **VIRTUAL**.*

- Индекс динамического метода должен задаваться целочисленной константой в диапазоне значений от 1 до 65535 и представлять собой уникальное значение среди индексов других динамических методов, содержащихся в объектном типе или его предках.

Переопределение динамического метода должно точно соответствовать порядку, типу и именам параметров, а также типу результата функции, если метод является функцией.

- Переопределение должно включать в себя директиву **VIRTUAL**, за которой следует тот же индекс динамического метода, что и заданный в объектном типе предка.

Использование динамических методов целесообразно при создании длинной иерархии объектов с большим количеством виртуальных методов.

- Для виртуальных методов в иерархии будут создаваться очень длинные ТВМ с указанием всех виртуальных методов предков, хотя переопределяться может только часть из них.
- Это требует большого объема используемой памяти для хранения ТВМ.

При использовании динамических методов создается альтернативная таблице виртуальных методов таблица динамических методов (ТДМ).

- В которой указываются только те виртуальные методы, которые переопределяются, что позволяет экономить память.