



МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра «Прикладная математика»

**Программирование в Delphi:
указатели и связанные списки**

Методические указания к лабораторной работе № 11
по курсам «Информатика», «Алгоритмические языки
и программирование»

Автор
Е.Н. Ладоса, Д.С. Цымбалов, О.В. Яценко,
А.П. Мул, В.И.Снигирева

Ростов-на-Дону, 2018



Аннотация

Описывается создание и обработка динамических, т.е. заранее не определенных структур в Object Pascal. Целью работы ставится рационализация обработки структурированной гетерогенной информации средствами Delphi. Предназначены для студентов всех специальностей факультета «Информатика и вычислительная техника».

Автор

Доцент, к.т.н.
Ладоша Е.Н.

Старший преподаватель кафедры
«Электроника и электротехника»
Цымбалов Д.С.

Доцент, к.ф.-м.н.
Яценко О.В.

Старший преподаватель кафедры
«Прикладная математика»
Мул А.П.

Студент ДГТУ
Снигирева В.И.



Цель работы

Цель работы – изучить возможности Delphi в части создания динамических объектов с формализуемой структурой. Рассмотрены основные средства работы с указателями и их практическое применение для создания связанных списков.

Указатели и связанные списки

Указатель – это переменная, в которой хранится адрес другой переменной. Иными словами, это переменная, указывающая на место в памяти, в котором хранится значение другой переменной.

С помощью указателей программист может создавать связанные списки и более сложные структуры данных и выделять для них память динамически, т.е. при выполнении программы. Используя указатели, можно легко добавлять или удалять данные из любой части структуры данных.

Связный список – это структура данных в виде списка, элементы которой связаны друг с другом с помощью указателей.

В *Pascal* указатели используются в любом типе данных, требующем динамического выделения больших блоков памяти. Например, длинные строки и объекты классов являются неявными указателями. Указатели всегда неявно присутствуют как бы «за кадром» исходного кода, даже если явно их в коде нет. Понимание указателей и операций с указателями необходимо для работы с *Pascal*. Более того: во многих приемах программирования указатели используются явно.

Использование указателей позволяет экономнее расходовать память, потому что с их помощью память для переменных можно выделять динамически. Однако динамическое выделение памяти в то же время является и недостатком использования указателей. Программист должен постоянно помнить о необходимости удаления неиспользуемых динамических объектов, иначе в программе произойдет *утечка памяти*. Интенсивная утечка памяти может вызвать крах программы, причем обнаружить источник утечки довольно трудно.

В *Pascal* символ «крышечка» (^) используется как для обозначения указателя, так и для его *разадресации*. Синтаксис определения типа указателя имеет вид

type

имя_типа_указателя = ^тип;

Например, оператор

type

```
IntPtr = ^Integer;
```

определяет тип указателя `IntPtr`. Переменная этого типа является указателем на целое значение, т.е. переменная типа `IntPtr` содержит адрес ячейки памяти, в которой хранится значение типа `Integer`.

Если определен тип указателя, то можно объявить указатель этого типа.

Например, оператор

```
intptr: IntPtr;
```

объявляет указатель `intPtr` типа `IntPtr`.

Если символ `^` расположен после указательной переменной, то он *разадресует* указатель, т.е. выражение возвращает значение, хранящееся по адресу, хранящемуся в указателе. Таким образом, выражение

имя_указателя[^]

разадресует указатель. Например, если переменная `intptr` имеет тип `IntPtr`, то выражение `intptr^` возвращает целое значение.

В *Pascal* есть зарезервированное ключевое слово `nil`, обозначающее специальную константу, значение которой можно присвоить любой указательной переменной. Если указателю присвоено значение `nil`, то это означает, что указатель не ссылается ни на какой объект. Поэтому рекомендуется всегда инициализировать указатели значением `nil`.

Для создания и уничтожения динамических переменных в *Pascal* используются встроенные процедуры `New()` и `Dispose()`. Процедура `New()` выделяет память для новой динамической переменной и присваивает указателю значение адреса выделенной области памяти. Когда созданная процедурой `New()` динамическая переменная больше не нужна, ее следует удалить с помощью процедуры `Dispose()`, т.е. освободить память, выделенную для этой переменной. Синтаксис процедур `New()` и `Dispose()` имеет вид

```
New(имя_указателя);
```

```
Dispose(имя_указателя);
```

После вызова процедуры `Dispose()` значение указателя становится неопределенным.

Внутри записей указатели используются для создания связных списков и других сложных структур данных. Например, приведенный ниже фрагмент кода создает, выводит на экран и уничтожает связный список целых значений от 1 до 10.

type

```
IntPtr = ^IntRecord;
```

```
IntRecord = record
```

```
    value: Integer;  
    next: IntPointer;  
end;
```

```
var
```

```
    top, temp: IntPointer;  
    count: Integer;
```

```
begin
```

```
    New(top);  
    temp := top;
```

```
    {Создание связанного списка}
```

```
    for count := 1 to 10 do begin
```

```
        temp^.value := count;
```

```
        if (count < 10) then begin
```

```
            New(temp^.next);
```

```
            temp := temp^.next;
```

```
        end
```

```
    else begin
```

```
        temp^.next := nil;
```

```
    end;
```

```
end;
```

```
    {Вывод}
```

```
    temp := top;
```

```
    while (temp <> nil) do begin
```

```
        Writeln(temp^.value);
```

```
        temp := temp^.next;
```

```
        if (temp <> nil) then begin
```

```
            writeln("");
```

```
            writeln('|');
```

```
            writeln('V');
```

```
            writeln("");
```

```
        end;
```

```
end;
```

```
    {Уничтожение}
```



```
while (top <> nil) do begin
    temp := top;
    top := top^.next;
    Dispose(temp);
end;
readln;

end.
```

В следующем примере указатели используются для создания связанного списка имен. Программа `LinkedList` содержит процедуры инициализации списка, добавления в список узлов, удаления узлов из списка, вывода содержимого списка на экран и уничтожения списка. Обратите внимание: цикл `while` в процедуре `Remove()` содержит булево выражение, требующее режима неполного вычисления, так как при `list`, равном `nil`, объекта `list^.name` не существует.

```
program LinkedList;

type
    NameType  = String[20];
    ListPointer = ^ListType;
    ListType   = record
        name: NameType;
        next: ListPointer;
    end;

{Инициализация связанного списка}
procedure InitializeList(var top: ListPointer);

begin
    top := nil;
end;

{Добавление в связанный список узла, содержащего имя}
procedure Add(var top: ListPointer; name: NameType);

var
    list: ListPointer;
```

```
begin
  if (top = nil) then begin
    New(top);
    list := top;
  end
  else begin
    list := top;
    while (list^.next <> nil) do begin
      list := list^.next;
    end;
    New(list^.next);
    list := list^.next;
  end;
  list^.name := name;
  list^.next := nil;
end;
```

{Удаления из связного списка узла, содержащего имя}
procedure Remove(var top: ListPointer; name: NameType);

```
var
  list, prev: ListPointer;
```

```
begin
  list := top;
  prev := list;
  while (list <> nil) and (list^.name <> name) do begin
    prev := list;
    list := list^.next;
  end;
  if (list <> nil) then begin
    if (prev = list) then begin {Удаление первого узла}
      prev := prev^.next;
      top := prev;
    end
    else begin {Удаление второго, или}
      prev^.next := list^.next; {последующего узла}
    end;
    Dispose(list);
  end;
```



```
end;  
end;
```

```
{Вывод связного списка}  
procedure DisplayList(top: ListPointer);
```

```
var  
    lineOut: String;  
  
begin  
    lineOut := "";  
    while (top <> nil) do begin  
        lineOut := lineOut + top^.name;  
        top := top^.next;  
        if (top <> nil) then begin  
            lineOut := lineOut + ' --> ';  
        end;  
    end;  
    writeln(lineOut);  
end;
```

```
{Уничтожение связного списка}  
procedure DestroyList(var top: ListPointer);
```

```
var  
    temp: ListPointer;  
  
begin  
    while (top <> nil) do begin  
        temp := top;  
        top := top^.next;  
        Dispose(temp);  
    end;  
end;  
var  
    list: ListPointer;
```

```
begin
```



```
InitializeList(list);  
Add(list, 'Jack');  
Add(list, 'Bill');  
Add(list, 'Tom');  
Add(list, 'Harry');  
DisplayList(list);  
Remove(list, 'Jack');  
DisplayList(list);  
Remove(list, 'Harry');  
DisplayList(list);  
Add(list, 'Sally');  
DisplayList(list);  
Remove(list, 'Tom');  
DisplayList(list);  
Add(list, 'Lucy');  
DisplayList(list);  
DestroyList(list);
```

```
readln;
```

```
end.
```

Операции с указателями и указатель **pointer**

Для указателей определены только операции присваивания и проверки на равенство и неравенство. В *Pascal* запрещается любые арифметические операции с указателями, их ввод-вывод и сравнение на больше-меньше.

Нетипизированный указатель не связан с каким-либо конкретным типом данных. Для нетипизированного указателя служит зарезервированное слово **Pointer**. Нетипизированные указатели используют в тех случаях, когда в процессе работы программы тип и структура данных изменяются. Для работы с не типизированными указателями существуют следующие процедуры:

Процедура **GetMem(P,Size)**, где **P** – переменная-указатель, **Size** – размер выделяемой памяти в байтах, позволяет выделить в динамической памяти область размера **Size**, при этом адрес выделенной области присваивается переменной **P**.

Процедура **FreeMem(P,Size)**, где **P** – переменная-указатель, **Size** – размер памяти в байтах, освобождает область в памяти, адрес которой задается указателем **P**, а размер равен **Size**

Указателям можно присваивать значения друг друга, если они указывают на

один и тот же тип. Нетипизированные указатели являются исключением из этого правила - они совместимы с любыми типизированными указателями. Для того чтобы записать значение по адресу, на который ссылается нетипизированный указатель необходимо использовать операции приведения типа.

В *Pascal* определены стандартные функции для работы с указателями:

- `addr(x):pointer` – возвращает адрес `x` (аналогично операции `@`).
- `ptr(x,y):pointer` – по заданному сегменту и смещению формирует адрес типа `pointer`.
- `seg(x):word` – возвращает адрес сегмента для `x`.
- `ofs(x):word` – возвращает смещение для `x`.

При работе с динамической памятью часто применяются вспомогательные функции:

- `Maxavail:longint` – возвращает длину в байтах самого длинного свободного участка динамической памяти.
- `Memavail:longint` – возвращает полный объем свободной динамической памяти в байтах.
- `Sizeof(x):word` – возвращает объем в байтах, занимаемый `x`.

Списки, стеки, очереди и очереди с двухсторонним доступом

Список – это упорядоченный набор данных. Первый узел списка называется называется *головой*, а последний – *хвостом*. Списки, стеки, очереди и очереди с двухсторонним доступом представляют собой специальные виды списков, добавление и удаление данных из которых регулируется определенными правилами. Реализация этих структур на основе статических массивов работает удовлетворительно только при относительно небольшом объеме данных. Динамические массивы более гибкие, с их помощью можно обрабатывать данные произвольного объема. Указатели позволяют создавать наиболее быстродействующие коды на основе оптимальных решений. Обычно указатели используются для обработки особенно больших объемов данных, однако при этом программисту приходится брать на себя дополнительную задачу управления памятью.

Наглядно стек можно представить себе как стопку тарелок, стоящих на столе. Когда рабочий в кафе моет посуду, он ставит каждую вымытую тарелку на самый верх стопки чистых тарелок. Когда в кафе заходит клиент, чистая тарелка для него снимается с верха стопки. Таким образом, последняя тарелка, поставленная в стопку, оказывается первой, извлеченной из стопки. Говорят, что

стек работает по принципу *LIFO* (last-in first-out, т.е. последним зашел – первым вышел).

Процессы добавления и чтения данных из стека называются *занесением в стек* и *извлечением из стека* (иногда используются термины «заталкивание в стек» и «выталкивание из стека»). Механизм работы стека напоминает магазин автомата, в который солдат заталкивает один патрон за другим. Во время стрельбы механизм автомата извлекает патроны тоже один за другим. Когда очередной патрон извлечен, на его место немедленно подается новый.

В компьютерном программировании используются термины *вершина* и *дно* стека. Вершиной стека является элемент данных, подлежащий извлечению. Между магазином автомата и стеком есть существенное отличие. В магазине автомата двигаются патроны, а в стеке элементы данных остаются неподвижными (данные не переписываются из ячейки в ячейку), вместо этого передвигается указатель на вершину стека, т.е. изменяется значение указателя. С этой точки зрения стек больше похож все же на стопку тарелок. Продемонстрируем работу стека с помощью следующего примера. Код программы StackEx реализует стек целых чисел на основе статического массива. Указателем на вершину стека является переменная `topOf Stack`. Как видно из программы, при инициализации указателю стека присваивается значение `nil`, т.е. если значение указателя равно `nil`, значит, стек пуст.

Для занесения и извлечения данных из стека предназначены процедуры `Pop()` и `Push()`. Статический массив стека `stack` типа `stackType` и относящиеся к нему переменные имеют глобальную область видимости.

```
program StackEx;
```

```
const
```

```
    STACKSIZE = 50; {Максимальный размер стека}
```

```
type
```

```
    StackType = array[1..STACKSIZE] of Integer; {Тип стека целых чисел}
```

```
var
```

```
    stack:    StackType;
```

```
    topOfStack: Integer;
```

```
{Инициализация стека}
```

```
procedure Initialize;
```

```
begin
  topOfStack := 0;
end;

{Функция IsEmpty возвращает True, если стек пуст}
function IsEmpty: Boolean;

begin
  IsEmpty := (topOfStack = 0);
end;

{Функция IsFull возвращает True, если стек заполнен}
function IsFull: Boolean;

begin
  IsFull := (topOfStack = STACKSIZE);
end;

{Извлечение целого числа с вершины стека}
procedure Pop(var value: Integer);

begin
  if IsEmpty then begin
    writeln('Недопустимая операция извлечения -- стек пуст!');
  end
  else begin
    value := stack[topOfStack];
    Dec(topOfStack);
  end;
end;

{Занесение целого числа в стек}
procedure Push(value: Integer);

begin
  if IsFull then begin
    writeln('Недопустимая операция занесения -- стек заполнен!');
  end
  else begin
```

```
Inc(topOfStack);
stack[topOfStack] := value;
end;
end;

{Тестирование стека}

var
  value, count : Integer;
  outstr: string;

begin

  Initialize;
  for count := 1 to 10 do begin
    Push(count);
    Str(count, outstr);
    Writeln('Занесено: ' + outstr);
  end;
  for count := 1 to 5 do begin
    Pop(value);
    Str(value, outstr);
    Writeln('Извлечено: ' + outstr);
  end;
  for count := 30 to 33 do begin
    Push(count);
    Str(count, outstr);
    Writeln('Занесено: ' + outstr);
  end;
  for count := 1 to 9 do begin
    Pop(value);
    Str(value, outstr);
    Writeln('Извлечено: ' + outstr);
  end;

  readln;
end.
```

Принцип действия очереди противоположен принципу действия стека. Стек похож на стопку тарелок, стоящую на столе, в то время как очередь напо-

минает живую очередь клиентов кафе. Первый клиент в очереди всегда обслуживается первым. Таким образом, очередь работает по принципу *FIFO* (first-in first-out, первым вошел – первым вышел). Когда данные добавляются в конец очереди, говорят, что они *ставятся очередь*. Термины «добавление» или «занесение в очередь» – синонимы термину «постановка в очередь». В программировании используются также термины *начало* и *конец* очереди. В начале очереди находится элемент данных, предназначенный для извлечения из очереди на следующем шаге.

Очередь с двусторонним доступом (двусторонняя очередь) обладает свойствами как стека, так и простой очереди. Данные в нее могут добавляться в любой конец и извлекаться из любого конца. Сравните: как в стек, так и в одностороннюю очередь данные могут добавляться только в конец. Обычно в программах, реализующих двустороннюю очередь, предусмотрены процедуры добавления данных в начало и в конец очереди, а также извлечения из начала и из конца.

Рассмотрим пример реализации на *Pascal* двусторонней очереди целых чисел. В коде программы `DequeEx`, реализующем двустороннюю очередь, используется двусвязный список, т.е. связный список, в котором каждый узел содержит два указателя: одни из них указывает на предыдущий узел, а другой – на следующий.

```
program DequeEx;

type
  NodePointer = ^NodeType;
  NodeType    = record
    value: Integer;
    next: NodePointer;
    last: NodePointer;
  end;
  DequeType   = record
    front: NodePointer;
    rear: NodePointer;
  end;

{Инициализация двусторонней очереди}
procedure Initialize(var deque: DequeType);
```

```
begin
  deque.front := nil;
  deque.rear := nil;
end;

{Функция IsEmpty возвращает True, если очередь пустая}
function IsEmpty(deque: DequeType): Boolean;

begin
  IsEmpty := (deque.front = nil) or (deque.rear = nil);
end;

{Добавление целого значения в начало двусторонней очереди}
procedure AddFront(var deque: DequeType; value: Integer);

var
  temp: NodePointer;

begin
  New(temp);
  temp^.value := value;
  temp^.next := deque.front;
  temp^.last := nil;
  if IsEmpty(deque) then begin
    deque.rear := temp;
  end
  else begin
    deque.front^.last := temp;
  end;
  deque.front := temp;
end;

{Добавление целого значения в конец двусторонней очереди}
procedure AddRear(var deque: DequeType; value: Integer);

var
  temp: NodePointer;

begin
```

```
New(temp);  
temp^.value := value;  
temp^.last := deque.rear;  
temp^.next := nil;  
if IsEmpty(deque) then begin  
    deque.front := temp;  
end  
else begin  
    deque.rear^.next := temp;  
end;  
deque.rear := temp;  
end;
```

{Извлечение целого значения из конца двусторонней очереди}

```
procedure RemoveFront(var deque: DequeType;  
    var value: Integer);
```

```
var  
    temp: NodePointer;
```

```
begin  
    if IsEmpty(deque) then begin  
        value := 0;  
        Writeln('Недопустимый вызов функции RemoveFront() -- двусторонняя оче-  
редь пустая!');  
    end  
    else begin  
        temp := deque.front;  
        value := temp^.value;  
        deque.front := temp^.next;  
        Dispose(temp);  
        if IsEmpty(deque) then begin  
            deque.rear := nil;  
        end  
        else begin  
            deque.front^.last := nil;  
        end;  
    end;  
end;  
end;
```


{Извлечение целого значения из конца двусторонней очереди}

```
procedure RemoveRear(var deque: DequeType;  
    var value: Integer);
```

```
var
```

```
    temp: NodePointer;
```

```
begin
```

```
    if IsEmpty(deque) then begin
```

```
        value := 0;
```

```
        Writeln('Недопустимый вызов функции RemoveRear() -- Двусторонняя очередь  
пустая!');
```

```
    end
```

```
    else begin
```

```
        temp := deque.rear;
```

```
        value := temp^.value;
```

```
        deque.rear := temp^.last;
```

```
        Dispose(temp);
```

```
        if IsEmpty(deque) then begin
```

```
            deque.front := nil;
```

```
        end
```

```
        else begin
```

```
            deque.rear^.next := nil;
```

```
        end;
```

```
    end;
```

```
end;
```

{Тестирование двусторонней очереди}

```
var
```

```
    deque:    DequeType;
```

```
    count, value: Integer;
```

```
    outstr: string;
```

```
begin
```

```
    Initialize(deque);
```

```
    for count := 1 to 10 do begin
```

```
        AddRear(deque, count);
```

```
Str(count, outstr);
Writeln('Добавлено в конец: ' + outstr);
end;
for count := 5 downto 1 do begin
  AddFront(deque, count);
  Str(count, outstr);
  Writeln('Добавлено в начало: ' + outstr);
end;
for count := 1 to 10 do begin
  RemoveFront(deque, value);
  Str(value, outstr);
  Writeln('Извлечено из начала: ' + outstr);
end;
for count := 1 to 5 do begin
  RemoveRear(deque, value);
  Str(value, outstr);
  Writeln('Извлечено из конца: ') + outstr);
end;
readln
end.
```

Контрольные вопросы.

1. Что называется указателем?
2. Для чего необходимо значение nil при работе с указателями?
3. Назовите операции, которые допускаются над значениями ссылочного типа?
4. Какие стандартные процедура реализуют основные действия над динамическими переменными?
5. К чему приводит «потеря» указателя на данные, хранимые в динамической памяти?
6. Ограничена ли динамическая память?

Задания

1. Реализуйте операции работы со стеком, построенным на основе динамических структур.
2. Реализуйте операции работы с очередью с односторонним доступом, построенной на основе динамических структур.



Список использованной литературы

1. *Фаронов В.В.* Delphi 3. Учебный курс. М.: «Нолидж», 1998. 400 с.
2. *Галисеев Г.В.* Программирование в среде Delphi 8 for .NET. М.: Издательский дом «Вильямс», 2004. 304 с.
3. *Павловска Т.А.* Паскаль. Программирование на языке высокого уровня. СПб.: Питер, 2003. 393 с.
4. *Абрамов С.А. и др.* Задачи по программированию. М.: Наука, 1988. 224 с.