Динамические структуры данных

Объект данных обладает динамической структурой, если его размер изменяется в процессе выполнения программы или он потенциально бесконечен.

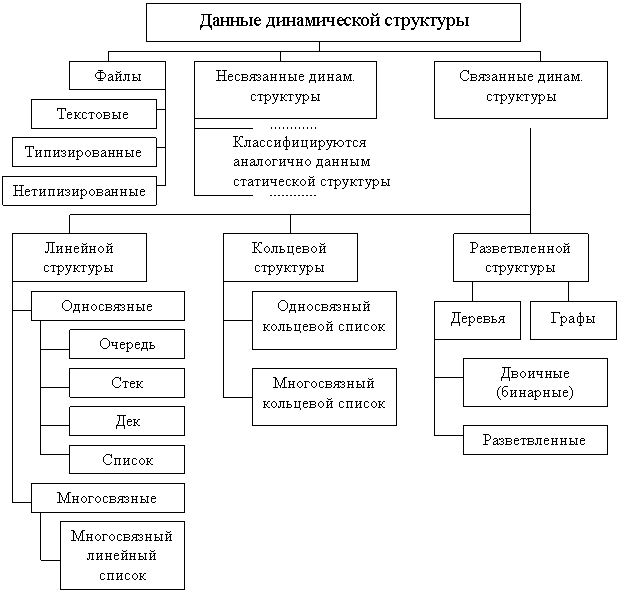
Классификация структур данных

Используемые в программировании данные можно разделить на две большие группы:



Данные статической структуры – это данные, взаиморасположение и взаимосвязи элементов которых всегда остаются постоянными.

Данные динамической структуры – это данные, внутреннее строение которых формируется по какому-либо закону, но количество элементов, их взаиморасположение и взаимосвязи могут динамически изменяться во время выполнения программы, согласно закону формирования.



Данные динамической структуры:

К данным динамической структуры относят файлы, несвязанные и связанные динамические данные.

Заметим, что файлы в данной классификации отнесены к динамическим структурам данных. Это сделано исходя из вышеприведенного определения. Хотя удаление и вставка элементов в середину файла не допускается, зато длина файла в процессе работы программы может изменяться – увеличиваться или уменьшаться до нуля. А это уже динамическое свойство файла как структуры данных.

Статические и динамические переменные в Паскале

В Паскале одной из задач описания типов является то, чтобы зафиксировать на время выполнения программы размер значений, а, следовательно, и размер выделяемой области памяти для них. Описанные таким образом переменные называются **статическими**.

Все переменные, объявленные в программе, размещаются в одной непрерывной области оперативной памяти – сегмент данных. Длина сегмента данных определяется архитектурой микропроцессора и составляет обычно 65536 байт.

Однако порой заранее не известны не только размеры значений, но и сам факт существования значения той или иной переменной. Для результата переменной приходится отводить память в расчете на самое большое значение, что приводит к нерациональному использованию памяти. Особенно это затруднительно при обработке больших массивов данных.

Предположим, например, что у вас есть программа, требующая массива в 400 строк по 100 символов каждая. Для этого массива требуется примерно 40К, что меньше максимума в 64К. Если остальные ваши переменные помещаются в оставшиеся 24К, массив такого объема проблемы не представляет. Но что если вам нужно два таких массива? Это потребовало бы 80К, и 64К сегмента данных не хватит. Другим общим примером является сортировка. Обычно когда вы сортируете большой объем данных, то делаете копию массива, сортируете копию, а затем записываете отсортированные данные обратно в исходный массив. Это сохраняет целостность ваших данных, но требует также наличия во время сортировки двух копий данных.

С другой стороны объем памяти компьютера достаточно велик для успешного решения задач с большой размерностью данных. Выходом из положения может служить использование так называемой динамической памяти.

Динамическая память (ДП) – это оперативная память ПК, предоставляемая программе при ее работе, за вычетом сегмента данных (64 Кб), стека (16 Кб) и собственно тела программы. Размер динамической памяти можно варьировать. По умолчанию ДП – вся доступная память ПК.

ДП – это фактически единственная возможность обработки массивов данных большой размерности. Многие практические задачи трудно или невозможно решить без использования ДП. Например, при разработке САПР статическое распределение памяти невозможно, т.к. размерность математических моделей в разных проектах может значительно различаться.

И статические и динамические переменные вызываются по их адресам. Без адреса не получить доступа к нужной ячейке памяти, но при использовании статических переменных, адрес непосредственно не указывается. Обращение осуществляется по имени. Компилятор размещает переменные в памяти и подставляет нужные адреса в коды команд.

Адресация динамических переменных осуществляется через **указатели**. Их значения определяют адрес объекта.

Для работы с динамическими переменными в программе должны быть выполнены следующие действия:

* **Выделение памяти под динамическую переменную;**
* **Инициализация указателя;**
* **Освобождение памяти после использования динамической переменной.**

Программист должен сам резервировать место, определять значение указателей, освобождать ДП.

Вместо любой статической переменной можно использовать динамическую, но без реальной необходимости этого делать не стоит.

Указатели

Для работы с динамическими программными объектами в Паскале предусмотрен ссылочный тип или тип указателей. В переменной ссылочного типа хранится ссылка на программный объект (адрес объекта).

**Указатель** – это переменная, которая в качестве своего значения содержит адрес байта памяти.

Объявление указателей

Указатель, связанный с некоторым определенным типом данных, называют типизированным указателем. Его описание имеет вид:

Имя\_переменной: ^ базовый-тип;

Например:

**Пример фрагмента программы объявления указателя**

Type A= array [1..100] of integer;   
   TA= ^ A ; {тип указатель на массив}  
Var  
   P1: ^ integer; {переменная типа указатель на целое число}  
   P2: ^ real; {переменная типа указатель на вещественное число}

Указатель, не связанный с каким-либо конкретным типом данных, называется **нетипизированным указателем**. Для описания нетипизированного указателя в Паскале существует стандартный тип **pointer**. Описание такого указателя имеет вид:

Имя-переменной: pointer;

С помощью нетипизированных указателей удобно динамически размещать данные, структура и тип которых меняются в ходе выполнения программы.

Значения указателей – адреса переменных в памяти. Адрес занимает четыре байта и хранится в виде двух слов, одно из которых определяет сегмент, второе – смещение.

Следовало бы ожидать, что значение одного указателя можно передать другому. На самом деле можно передавать значения только между указателями, связанными с одним типом данных. Указатели на различные типы данных имеют различный тип, причем эти типы несовместимы.

**Пример фрагмента программы объявления указателя различных типов**

Var p1,p2: ^integer;   
   p3: ^real;   
   pp: pointer;   
   ………   
   p1:= p2; {допустимое действие }   
   p1:= p3; {недопустимое действие}

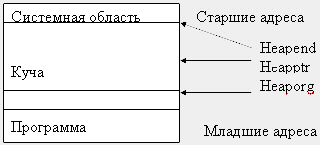
Однако это ограничение не распространяется на нетипизированный указатель. В программе допустимы будут следующие действия:

pp:= p3;  
p1:= pp;

Выделение и освобождение динамической памяти

Вся ДП рассматривается как сплошной массив байтов, который называется **кучей**.

Расположение кучи в памяти ПК.



Существуют стандартные переменные, в которых хранятся значения адресов начала, конца и текущей границы кучи:

* **Heaporg – начало кучи;**
* **Heapend – конец кучи;**
* **Heapptr – текущая граница незанятой ДП.**

Выделение памяти под динамическую переменную осуществляется процедурой:

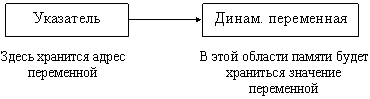
New (переменная\_типа\_указатель)

В результате обращения к этой процедуре указатель получает значение, соответствующее адресу в динамической памяти, начиная с которого можно разместить данные.

**Пример фрагмента программы объявления указателя различных типов**

Var i, j: ^integer;  
   r: ^real;  
begin  
   new( i); {после этого указатель i приобретает значение адреса Heapptr, а Heapptr смещается на 2 байта}  
   ……………  
   new( r) ; { r приобретает значение Heapptr, а Heapptr смещается на 6 байт}

Графически действие процедуры new можно изобразить так:



Освобождение динамической памяти осуществляется процедурой:

Dispose (переменная\_типа\_указатель)

**Пример фрагмента программы процедуры Dispose**

Dispose (i); {возвращает в кучу 2 байта}  
Dispose (r); {возвращает в кучу 6 байт}

Следует помнить, что повторное применение процедуры dispose к свободному указателю может привести к ошибке.

Процедура dispose освобождает память, занятую динамической переменной. При этом значение указателя становится неопределенным.

Процедура dispose

Любые действия над указателем в программе располагаются между процедурами new и dispose.

При использовании динамически распределяемых переменных часто возникает общая проблема, называемая утечкой динамической памяти. Утечка памяти – это ситуация, когда пространство выделяется в динамически распределяемой памяти и затем теряется – по каким-то причинам ваш указатель не указывает больше на распределенную область, так что вы не можете освободить пространство. Общей причиной утечек памяти является переприсваивание динамических переменных без освобождения предыдущих. Простейшим случаем является следующий:

**Пример фрагмента программы**

var IntPointer :^ Integer;   
begin   
    New (IntPointer);  
    New (IntPointer);   
end.

При первом вызове New в динамически распределяемой памяти выделяется 2 байта, и на них устанавливается указатель IntPointer. Второй вызов New выделяет другие 2 байта, и IntPointer устанавливается на них. Теперь у вас нет указателя, ссылающегося на первые 2 байта, поэтому вы не можете их освободить. В программе эти байты будут потеряны.

Присваивание значений указателю

Для инициализации указателей существует несколько возможностей.

* **процедура new отводит блок памяти в области динамических переменных и сохраняет адрес этой области в указателе;**
* **специальная операция @ ориентирует переменную-указатель на область памяти, содержащую уже существующую переменную. Эту операцию можно применять для ориентации на статическую и динамическую переменную.  
  Например,**

**type A=array[0..99] of char;  
var X: array [0..49] of integer;  
pA:^ A; {указатель на массив символов}**

**Объявлены переменные разных типов: массив из 50 целых чисел и указатель на массив символов. Чтобы указатель pA указывал на массив X, надо присвоить ему адрес X**

**pA:= @ X;**

* **Существует единственная константа ссылочного типа nil, которая обозначает «пустой» адрес. Ее можно присваивать любому указателю.**
* **Переменной-указателю можно присвоить значение другого указателя того же типа. Используя указательный тип pointer как промежуточный, можно присвоить значение одного указателя другому при несовпадении типов.**

Операции с указателями

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

Еще раз повторим правила присваивания указателей:

* **любому указателю можно присвоить стандартную константу nil, которая означает, что указатель не ссылается на какую-либо конкретную ячейку памяти;**
* **указатели стандартного типа pointer совместимы с указателями любого типа;**
* **указателю на конкретный тип данных можно присвоить только значение указателя того же или стандартного типа данных.**

Указатели можно сравнивать на равенство и неравенство, например:

If p1=p2 then …..  
If p1<>p2 then …..

В Паскале определены стандартные функции для работы с указателями:

* **addr( x) – тип результата pointer, возвращает адрес x (аналогично операции @), где x – имя переменной или подпрограммы;**
* **seg( x) – тип результата word, возвращает адрес сегмента для x;**
* **ofs( x) – тип результата word, возвращает смещение для x;**
* **ptr( seg, ofs) – тип результата pointer, по заданному сегменту и смещению формирует адрес типа pointer.**

Присваивание значений динамическим переменным

После того, как динамическая переменная объявлена, ей можно присваивать значения, изменять их, использовать в выражениях и т.д. Для этого используют следующее обращение: переменная\_указатель^. Такое обращение называется операция разадресации (разыменования). Таким образом происходит обращение к значению, на которое указывает указатель, т.е. к данным. Если же за переменной\_указателем значок ^ не стоит, то имеется в виду адрес, по которому расположены данные.

Динамически размещенные данные можно использовать в любом месте программы, где допустимо использование выражений соответствующего типа.

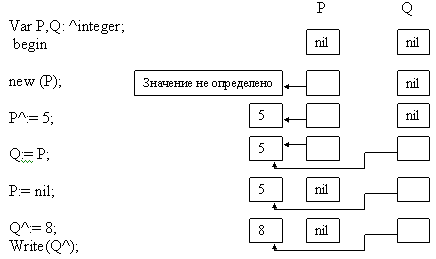
Например:

R^:= sqr( R^) + I^ -17;  
q^:= 2; inc(q^); writeln(q^)

Недопустимо использовать выражения, подобные следующим:

Адрес --->R:= sqr( R^) + I^ -17 <---вещественное выражение.  
Вещественная переменная --->R^:= sqr( R) <---аргумент – адрес.

Рассмотрим пример работы с указателями:



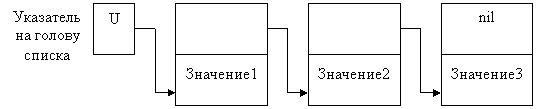
Динамические структуры

Линейные списки (однонаправленные цепочки).

**Списком** называется структура данных, каждый элемент которой посредством указателя связывается со следующим элементом.

Каждый элемент связанного списка, во-первых, хранит какую-либо информацию, во-вторых, указывает на следующий за ним элемент. Так как элемент списка хранит разнотипные части (хранимая информация и указатель), то его естественно представить записью, в которой в одном поле располагается объект, а в другом – указатель на следующую запись такого же типа. Такая запись называется **звеном**, а структура из таких записей называется списком или цепочкой.

Лишь на самый первый элемент списка (голову) имеется отдельный указатель. Последний элемент списка никуда не указывает.



Описание списка

**Пример описания списка**

Type ukazat= ^ S;  
   S= record  
      Inf: integer;  
      Next: ukazat;  
   End;

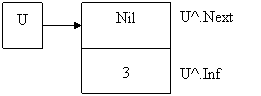
В Паскале существует основное правило: перед использованием какого-либо объекта он должен быть описан. Исключение сделано лишь для указателей, которые могут ссылаться на еще не объявленный тип.

Формирование списка

Чтобы список существовал, надо определить указатель на его начало.

**Пример описания списка**

Type ukazat= ^S;   
   S= record   
      Inf: integer;   
      Next: ukazat;   
   End;



Создадим первый элемент списка:

New (u); {выделяем место в памяти}  
u^. Next:= nil; {указатель пуст}  
u^. Inf:=3;

Продолжим формирование списка. Для этого нужно добавить элемент либо в конец списка, либо в голову.

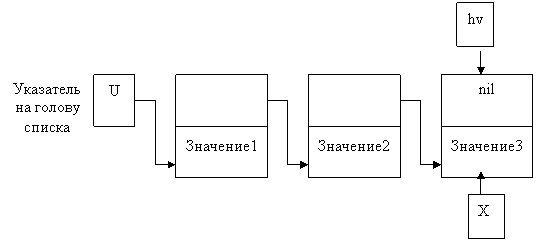
А) Добавим элемент в голову списка. Для этого необходимо выполнить последовательность действий:

* **получить память для нового элемента;**
* **поместить туда информацию;**
* **присоединить элемент к голове списка.**

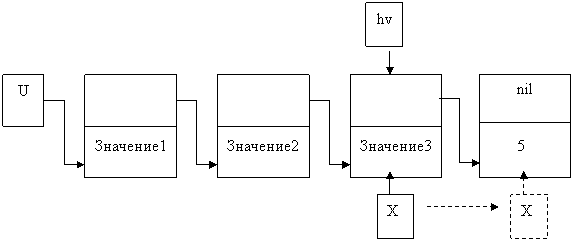
New(x);   
Readln(x^.Inf);   
x^. Next:= u;   
u:= x;

Б)Добавление элемента в конец списка. Для этого введем вспомогательную переменную, которая будет хранить адрес последнего элемента. Пусть это будет указатель с именем hv (хвост).

x:= hv;



New( x^. next); {выделяем память для следующего элемента}



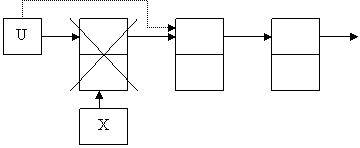
x:= x^.next;   
x^.next:= nil;   
x^. inf:= 5;   
hv:=x;

Просмотр списка

While u<> nil do  
Begin  
Writeln (u^.inf);  
u:= u^.next;>  
end;

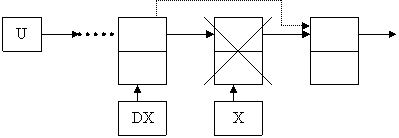
Удаление элемента из списка

А)Удаление первого элемента. Для этого во вспомогательном указателе запомним первый элемент, указатель на голову списка переключим на следующий элемент списка и освободим область динамической памяти, на которую указывает вспомогательный указатель.



x:= u;   
u:= u^.next;   
dispose(x);

Б) Удаление элемента из середины списка. Для этого нужно знать адреса удаляемого элемента и элемента, стоящего перед ним. Допустим, что digit – это значение удаляемого элемента.



x:= u;   
while ( x<> nil) and ( x^. inf<> digit) do   
begin   
dx:= x;   
x:= x^.next;   
end;   
dx:= x^.next:   
dispose(x);

В)Удаление из конца списка. Для этого нужно найти предпоследний элемент.

x:= u; dx:= u;   
while x^.next<> nil do   
begin   
dx:= x; x:= x^.next;   
end;   
dx^.next:= nil;   
dispose(x);

**Прохождение списка**. Важно уметь перебирать элементы списка, выполняя над ними какую-либо операцию. Пусть необходимо найти сумму элементов списка.

summa:= 0;   
x:= u;   
while x<> nil do   
begin   
summa:= summa+ x^.inf;   
x:= x^.next;   
end;

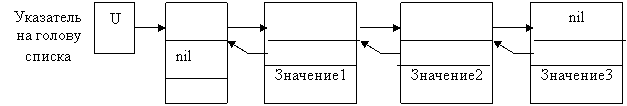
Динамические объекты сложной структуры

Использование однонаправленных списков при решении ряда задач может вызвать определенные трудности. Дело в том, что по однонаправленному списку можно двигаться только в одном направлении, от головы списка к последнему звену. Между тем нередко возникает необходимость произвести какую-либо операцию с элементом, предшествующим элементу с заданным свойством. Однако после нахождения элемента с данным свойством в однонаправленном списке у нас нет возможности получить удобный и быстрый способ доступа к предыдущему элементу.

Для устранения этого неудобства добавим в каждое звено списка еще одно поле, значением которого будет ссылка на предыдущее звено.

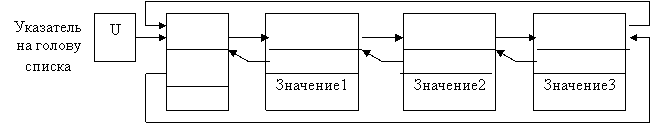
Type ukazat= ^S;   
S= record   
Inf: integer;   
Next: ukazat;   
Pred: ukazat;   
End;

Динамическая структура, состоящая из звеньев такого типа, называется **двунаправленным списком**, который схематично можно изобразить так:



Наличие в каждом звене двунаправленного списка ссылки как на следующее, так и на предыдущее звено позволяет от каждого звена двигаться по списку в любом направлении. По аналогии с однонаправленным списком здесь есть заглавное звено. В поле Pred этого звена фигурирует пустая ссылка nil, свидетельствующая, что у заглавного звена нет предыдущего (так же, как у последнего нет следующего).

В программировании двунаправленные списки часто обобщают следующим образом: в качестве значения поля Next последнего звена принимают ссылку на заглавное звено, а в качестве значения поля Pred заглавного звена – ссылку на последнее звено:



Как видно, здесь список замыкается в своеобразное «кольцо»: двигаясь по ссылкам, можно от последнего звена переходить к заглавному звену, а при движении в обратном направлении – от заглавного звена переходить к последнему. Списки подобного рода называют **кольцевыми списками**.

Существуют различные методы использования динамических списков:

* **Стек – особый вид списка, обращение к которому идет только через указатель на первый элемент. Если в стек нужно добавить элемент, то он добавляется впереди первого элемента, при этом указатель на начало стека переключается на новый элемент. Алгоритм работы со стеком характеризуется правилом: «последним пришел – первым вышел».**
* **Очередь– это вид списка, имеющего два указателя на первый и последний элемент цепочки. Новые элементы записываются вслед за последним, а выборка элементов идет с первого. Этот алгоритм типа «первым пришел – первым вышел».**
* **Возможно организовать списки с произвольным доступом к элементам. В этом случае необходим дополнительный указатель на текущий элемент.**