**27.05.20**

Уважаемые студенты группы ИС21. Сегодня мы изучаем теорию. Вам необходимо будет выполнить следующие задания: внимательно изучите материал урока; для подготовки к экзамену в тетрадь выпишите: основные понятия и определения, описание указателей, стандартные процедуры и функции для работы с указателями, типовые дейст­вия со списками.

Отчет о работе отправьте его по электронной почте на yun707@yandex.ru. При отправлении выполненных заданий укажите ФАМИЛИЮ СТУДЕНТА, в Теме НАИМЕНОВАНИЕ ДИСЦИПЛИНЫ и НАЗВАНИЕ ВЫПОЛНЕННОЙ РАБОТЫ.

**Тема: Динамические переменные. Переменные указателя. Линейные структуры данных: стеки, очереди, списки**

**Указатели**

При выполнении программы каждая используемая в ней переменная получает свой адрес в оперативной памяти. Программисту не нужно заботиться о механизме распределения адресов, это делается автоматически. Машина сама ищет свободное место в памяти и выделяет в ней место для переменных. В Турбо-Паскале есть два способа распределения памяти: *статический* и *динамический*.

При ***статическом распределении*** всем объявленным в программе переменным в сегменте данных выделяются фиксированные участки оперативной памяти. В связи с этим использование заранее не объявленных переменных не допускается.

При ***динамическом распределении*** памяти имеется возможность создавать новые, не объявленные заранее, переменные и размещать их на свободные участки в динамической области оперативной памяти. Динамическая переменная не указывается явно в описаниях переменных и к ней нельзя обратиться по имени. Это достигается за счет использования указателей.

***Указатель*** – это элемент данных, представляющий собой ссылку на определенную ячейку оперативной памяти (т.е. адрес ячейки), начиная с которой записывается значение переменной.

Переменные, которые размещаются в оперативной памяти динамическим способом с помощью указателей, называются ***динамическими переменными***.

Указатель может принимать значения, равные всем адресам оперативной памяти, по которым возможна запись данных. Указатель может иметь стандартное значение **Nil** (пусто), которая говорит, что соответствующая переменная в динамической памяти отсутствует (в указателе не содержится никакого адреса).

**Описание указателей**

Указатель объявляется с помощью символа каре (**^**), за которым записывается идентификатор типа динамической переменной:

**1)**Type имя\_типа = ^тип;

Var имя\_переменной = имя\_типа;

Или

**2)**Var имя\_переменной: ^имя\_типа;

 Например: Var A, B, C: ^Integer;

В этом случае переменные *A, B, C* являются указателями на переменные типа ***Integer***. Для обращения к значениям этих переменных служат идентификаторы **A^**, **B^**, **C^**, т.е. имя переменной и знак каре.

Кроме того, указатель может быть объявлен явно следующим образом:

**3)**Var p: Pointer;

Над указателями не определено никаких операций, кроме проверки на равенство и неравенство.

Допустимо также использование оператора присваивания, при этом в правой части может находиться либо функция определения адреса **Addr(X)**, либо выражение **@X**, где **@** — унарная операция взятия адреса, **X** – имя переменной любого типа, в том числе процедурного.

Для динамических переменных (**A^**, **B^**, **C^**) допустимы все те же операции, что и над обычными переменными данного типа.

Переменные типа указатель не могут быть элементами списка ввода-вывода.

**Стандартные процедуры и функции для работы с указателями**

Работа с динамической памятью осуществляется с помощью процедур и функций. ***Процедуры:***

1. Любым действиям с динамической переменной должна предшествовать процедура ее размещения в ОЗУ. Эта процедура имеет вид: **New (Var p: Pointer)**. Она создает новую динамическую переменную, присваивая указателю ***p*** значение ее адреса в ОЗУ. При этом динамической переменной отводится блок памяти, соответствующий размеру типа, с которым объявлен указатель ***p***.
2. Когда в ходе вычислительного процесса переменная становится ненужной, ее следует удалить. Это осуществляется процедурой **Dispose (Var p: Pointer)**. Данная процедура освобождает память, занятую динамической переменной, делая значение ее указателя неопределенным.

Type Str = string[6];

Var P = ^Str;

Begin

New(P);

P^:= ?Hellow?;

Dispose(P)

End.

1. Процедура **GetMem (Var p: Pointer, Size: Word)**, где **Word** — 2 байта (целые числа от 0 до 65535), создает новую динамическую переменную размером **Size** байт,  устанавливая значения указателя  на начало выделяемой ей динамической области оперативной памяти.
2. Процедура **FreeMem (Var p: Pointer, Size: Word)** уничтожает динамическую переменную, освобождая **Size** байт. После выполнения процедуры **FreeMem** значение ***p*** становится неопределенным.
3. Процедура **Mark (Var p: Pointer)** записывает в указатель ***p*** адрес начала участка свободной динамической памяти на момент ее вызова.
4. Процедура **Release (Var p: Pointer)** освобождает участок динамической памяти, начиная с адреса, записанного в указатель ***p*** процедурой **Mark**, т.е. очищает ту динамическую память, которая была занята после вызова процедуры **Mark**.

Пример:

Var

p: pointer;

p1, p2, p3: ^Integer;

Begin

New(p1); {Создает указатель на переменную типа Integer}

Mark (p); {Запоминает состояние (все дальнейшие указатели будут размещаться за указателем p)}

New(p2); {Создает указатели еще на две переменные типа Integer}

New(p3);

Release (p) {Память, выделенная для p2^, p3^ освобождена, но p1^ все еще можно использовать}

End.

***Функции:***

1. **MaxAvail: LongInt** – возвращает длину в байтах самого длинного свободного участка динамической памяти.
2. **MemAvail: LongInt –** возвращает размер свободной области динамической памяти (в байтах).
3. **Addr(X):** **Pointer –** возвращает адрес объекта, где ***X*** — любая переменная, имя процедуры или функции.
4. **SizeOf(X): Word** возвращает объем в байтах, занимаемый ***X***, причем ***X*** может быть либо именем переменной любого типа, либо именем типа.

Все изложенное пока не дает ответа на вопрос: «А зачем это нужно?» Отвечаем. Динамические переменные используются в основном в двух ситуациях:

* для работы с массивами больших размеров;
* для работы с особыми структурами переменных размеров, которые получили название динамических структур данных.

**Массив указателей**

В чем же преимущество динамического выделения памяти, если приходится выполнять так много дополнительных действий? Дело в том, что транслятор ограничивает суммарный объем статической памяти одним сегментом, т.е. 64 Кб. Объем динамической памяти ограничен лишь физическим ресурсом компьютера, а это гораздо больше.

Предположим, мы хотим запомнить как можно больше длинных строк символов. Размер строки – 256 байт, поэтому их предельное количество в статической памяти равно 64\*1024/256=256.

Для хранения большего числа строк разместим в статической памяти лишь указатели на них, а сами строки перенесем в динамическую память (размер одного указателя – 4 байта)

Указатели чаще всего используются для работы с ***динамическими массивами памяти***, которые представляют собой массивы переменной длины, память под которые может выделяться и изменяться в процессе выполнения программы, как при каждом новом запуске программы, так и в разных ее частях. Обращение к **i**-му элементу динамического массива **x** имеет вид **x^[i]**.

При работе с динамическими переменными необходимо соблюдать следующий порядок работы:

1. описать указатели;
2. распределить память;
3. обработать динамический массив;
4. освободить память.

Понятие динамического массива можно распространить и на матрицы. Динамическая матрица представляет собой массив указателей, каждый из которых адресует одну строку (или столбец). Рассмотрим описание динамической матрицы. Пусть есть типы данных **massiv** и указатель на него **din\_massiv**:

type massiv=array [1..1000] of real;

din\_massiv=^massiv;

Динамическая матрица X будет представлять собой массив указателей:

var X: array[1..1000] of din\_massiv;

 Работать с матрицей необходимо следующим образом:

1. определить ее размеры (пусть N – число строк, M – число столбцов);
2. выделить память под матрицу:
	* for i:=1 to N do
	* getmem(X[i], M\*sizeof(real));

Каждый элемент статического массива X[i] – указатель на динамический массив, состоящий из M элементов типа real. В статическом массиве X находится N указателей.

1. для обращения к элементу динамической матрицы, расположенному в i-той строке и j-м столбце, следует использовать следующую конструкцию: **X[i]^[j]**;
2. после завершения работы с матрицей необходимо освободить память:
	* for i:=1 to N do
	* freemem(X[i], M\*sizeof(real));

Пример:

Const

MaxItem=2000;

Type

Pstring=^String;

TDinMas=Array[1.. MaxItem] Of Pstring;

Var

p: TDinMas;

i: Ineger;

Begin

For i:=1 To MaxItem Do New(p[i]);

p[1]^:= ?Hallo!?

End.

При хранении большого количества однотипных данных можно использовать динамические массивы, но у них есть существенные ***недостатки***:

* размер динамического массива постоянен;
* невозможно добавление элементов в середину массива.

Если же количество данных непостоянно и изменяется в процессе выполнения программы, либо существует необходимость добавления элементов в середину набора данных, что особенно актуально при работе с упорядоченными наборами данных, то целесообразно использовать динамические структуры данных.

**Динамические структуры данных**

Структурированные типы данных, такие, как массивы, множества, записи, представляют собой ***статические структуры***, т.к. их размеры неизменны в течение всего времени выполнения программы.

Часто требуется, чтобы структуры данных меняли свои размеры в ходе решения задачи. Такие структуры называются ***динамическими***, к ним относятся *стеки, очереди, списки, деревья* и др. Описание динамических структур с помощью массивов, записей и файлов приводит к неэкономному использованию памяти ЭВМ и увеличивает время решения задач.

В поисках решения проблемы быстрой обработки больших объемов данных были предложены динамические структуры данных. Они характеризуются следующими особенностями:

* для отдельных элементов структуры память выделяется в тот момент, ко­гда в них появляется необходимость (а не сразу и одним блоком как для массивов);
* число элементов динамической структуры заранее не объявляется и мо­жет изменяться от нуля до некоторого значения, определяемого специ­фикой соответствующей задачи или доступным объемом памяти;
* память, занимаемая структурой, не представляет собой непрерывную об­ласть, т. е. элементы могут быть разбросаны в памяти хаотическим об­разом;
* логическая последовательность элементов задается в явном виде с по­мощью одного или нескольких указателей, хранящихся в самих элемен­тах. Как правило, каждый элемент, кроме своего значения, хранит указа­тель на следующий элемент или на два соседних с ним элемента.

Для того чтобы идея стала совсем понятной, проведем такую аналогию. При записи на прием к врачу каждый пациент получает свой порядковый номер в списке пациентов, и ему сообщается время приема. Если очередь к врачу движется строго по номерам, то никто не обязан запоминать своих соседей по очереди.

Другое дело – живая очередь (в магазине, железнодорожной кассе). Коли­чество человек в очереди постоянно меняется, люди приходят и уходят, но никакой неразберихи не происходит – каждый знает, за кем он стоит. Об­разуется связанная цепочка людей. Очевидно, кому-то из стоящих в очереди и пришла идея создания динамических структур данных.

Простая по своей сути идея оказалась не столь простой в реализации, по­этому дальнейший материал потребует повышенного внимания. Рассмотрим наиболее распространенную динамическую структуру, которая называется *связанным списком*.

***Связанный список*** – это такая структура данных, эле­ментами которой служат записи одного и того же формата, связанные друг с другом с помощью указателей, расположенных в самих элементах. Эле­менты списка часто называются его *узлами.*

Использование записей для представления элементов связанного списка вполне закономерно. Это единственный комбинированный тип данных, который допускает группирование данных различных типов. Каждый элемент списка состоит из двух различных по значению частей: *содержательной* (информационной) и *указующей* (см. рис. ниже). В минимальном варианте содержательная и указующая части занимают по одному полю записи, но могут представлять собой и более одного поля.

|  |
| --- |
| **Элемент (узел) связанного** **списка — запись**  |
| Содержательная часть | Указующая часть |
| Данные любого типа | Один или два указателя на соседний элемент (элементы) |

В содержательной части хранятся данные, ради которых и создается список.

Если указующая часть хранит адрес одного соседнего элемента списка, то
такой список называется ***односвязны****м*, или *однонаправленным*. Поле указате­ля последнего элемента содержит специальный признак ***Nil*** (признак конца     списка). Для каждого элемента (кроме последнего) имеется единственный следующий элемент, поэтому связанные списки иногда называют *линейными*.  Логическая структура односвязного списка представлена ниже.

В случае, когда в указующей части хранятся адреса и предыдущего, и следующего элементов, список называется ***двусвязным****,* или *двунаправленным.* Двунаправленный список более универсален, т. к. по такой цепочке можно двигаться в двух направлениях: прямом и обратном. В двусвязном списке можно продвигаться от элемента к элементу двумя противоположными пу­тями, поэтому в конце каждого из них в поле соответствующего указателя находится признак пустого указателя (***Nil***).

Из односвязного и двусвязного списков можно получить кольцевой список, который вообще не содержит пустых указателей. Кстати, буфер клавиатуры ПК реализован именно как кольцевой список.

С линейными списками можно выполнять те же действия, что и с одномер­ными массивами, поскольку назначение списков и массивов одно и то же — обработка данных в оперативной памяти.

Перечислим ***типовые дейст­вия со списками***:

* добавить новый узел непосредственно перед заданным узлом;
* удалить заданный узел;
* объединить два (или более) линейных списка в один список;
* разбить линейный список на два (или более) списка;
* сделать копию списка;
* выполнить сортировку узлов списка в возрастающем порядке по некото­рым полям в узлах;
* найти в списке узел с заданным значением в некотором поле.

Очень часто встречаются линейные списки, в которых добавление и удале­ние производятся только в первом или последнем узлах. Назовем эти списки.

* ***Стек*** — линейный   список,   в   котором   все   добавления   и   удаления (и обычно всякий доступ) делаются в одном конце списка.
* ***Очередь*** — линейный список, в котором все добавления производятся на одном конце списка, а все удаления делаются на его другом конце.
* ***Дек*** (очередь с двумя концами) — линейный список, в котором все до­бавления и удаления делаются на обоих концах списка.

Прежде чем рассматривать действия со связанными списками, введем обо­значения переменных, которыми будем пользоваться при описании соответ­ствующих алгоритмов и структур данных (см. в табл.).

|  |  |
| --- | --- |
| **Элемент** | **Описание** |
| item, pitem | Тип для одного элемента списка (запись) и указателя на него |
| data | Поле данных (информационная часть элементов списка) |
| next,prev | Указатели следующего, предыдущего элементов (указующая часть) |
| head, top | Указатели на первыйи последний элемент списка |
| pl,p2 | Рабочие указатели |

Опишем тип одного элемента односвязного списка и указателя на этот эле­мент:

type pitem=^item;

item=record

data:... {простой или определяемый пользователем тип}

next:pitem;{или prev:pitem;}

end;

*Обратите внимание на описание* – это единственный разрешенный случай, когда тип используется до объявления (***item***). Очевидно, разработчики ком­пилятора сделали исключение ввиду особой важности списковых структур.

Перейдем к примерам. Наиболее просто реализуются действия со *стеком*, поэтому первый пример демонстрирует использование стека. Все действия со стеком выполняются только на одном конце, который на­зывается *вершиной стека*. Стеки как структуры данных имеют широкое при­менение в системном программировании (например, при разработке ком­пиляторов). В частности, область памяти, в которую помещаются парамет­ры и локальные переменные подпрограмм, имеет структуру стека. Именно благодаря устройству стека возможны такие приемы программирования, как вложенные подпрограммы и рекурсия.

В примере реализуется учебный стек, содержащий целые числа.

type pitem=^item;

item=record {элемент стека}

data:integer; {значение элемента}

prev:pitem; {адрес предыдущего элемента}

end;

var top,p:pitem;

n,k:integer;

procedure add(x:integer); {добавляет элемент на вершину стека}

begin

new(p); {создаем произвольный элемент р}

p^.data:=x; p^.prev:=top;

top:=p; {}

end;

 procedure deltop; {удаляет узел с вершины стека}

begin

if top < >nil then begin {если стек не пустой}

p:=top^ . prev; {запоминаем предшествующий вершине элемент}

dispose(top); top:=p; {устанавливаем p вершиной стека}

end;

end;

  procedure writestack; {выводит стек на экран}

begin

writeln(‘Содержимое стека (начиная с вершины):’);

p:=top;

while p<>nil do begin

write (p^.data, ‘ ’ ); p:= p^ . prev;

end;

writeln;

end;

  begin {начало программы}

top:=nil;

for k:=1 to 10 do add(k); {заполняем стек числами от 1 до 10}

writestack;

writeln(‘Введите значение элемента для добавления в стек:’);

readln(n); add(n);

writestack;

writeln(‘Сколько элементов стека нужно удалить?’); readln(n);

for k:=1 to n do deltop;

writestack;

readln

end.

В примере реализованы две основные операции со стеком: добавление и удаление элементов. Для решения задачи потребовалось всего два указателя типа pitem. Один из них (top) всегда указывает на вершину стека, второй (p) – рабочий указатель, предназначенный для временного хранения адресов различных элементов. Обратите внимание на типовую процедуру вывода списка при помощи цикла while. Стандартное действие p:= p^ . prev; означает переход к следующему элементу стека (для стека правильнее назвать этот элемент не следующим, а предыдущим, т.к. он был помещен в стек раньше). Поэтому элементы стека можно вывести только в порядке, обратном тому, в котором они выводились.

Подведем итоги. Итак, связные списки и массивы – две основные структуры в оперативной памяти, которые используются для обработки данных. Сравним их между собой, выделив главные моменты:

* организация данных в памяти в виде связных списков обеспечивает более экономное использование памяти по сравнению с массивами;
* другим преимуществом связных списков является удобство вставки и удаления элементов. В массиве для этих целей приходилось раздвигать или сдвигать элементы. В списке для удаления и вставки достаточно только поменять значения указующих полей соседних элементов;
* явным достоинством массивов является простота их использования по сравнению со списками;
* еще более существенным преимуществом массива является высокая скорость доступа к элементу массива по его индексу. А для получения доступа к последнему элементу односвязного списка необходимо последовательно обойти всю цепочку, начиная с самого первого элемента.

Из сказанного можно сделать вывод – при решении задач обработки данных во многих случаях выбор между массивом и списком сделать непросто. Необходимо тщательно взвесить все плюсы и минусы обоих вариантов, а далее действовать по обстоятельствам.

**ОБРАЩАЮ ВАШЕ ВНИМАНИЕ НА ТО, ЧТО ДО ОКОНЧАНИЯ ИЗУЧЕНИЯ ДИСЦИПЛИНЫ «ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ» ОСТАЛОСЬ ЧЕТЫРЕ ЗАНЯТИЯ, А ПОТОМ ЭКЗАМЕН. ПОЭТОМУ, СДАЕМ ВСЕ ПРАКТИЧЕСКИЕ РАБОТЫ, У КОГО ЕСТЬ ЗАДОЛЖЕННОСТЬ.**