УНИВЕРСИТЕТ СИНЕРГИЯ | БОЛЬШЕ ЧЕМ ОБРАЗОВАНИЕ

Основа алгоритмизации и программирования «Рекурсия»

Практическое занятие
Лекция

Кафедра цифровой экономики

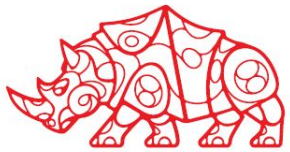
Гринева Е.С., преподаватель

21 октября 2022 г.

УНИВЕРСИТЕТ СИНЕРГИЯ

Цель занятия:

- **Изучить:**
- Динамические структуры данных: стек, очередь, дек.
- Рекурсивные процедуры и функции.
- Сравнить рекурсивные и нерекурсивные алгоритмоы



УНИВЕРСИТЕТ СИНЕРГИЯ

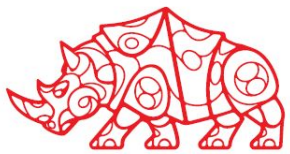
Глобальные и локальные переменные (константы, типы данных)

Переменные (константы, типы данных), которые описаны в основной программе называются **глобальными**. Переменные, описаные внутри подпрограммы – **локальными**.

```
Program GLOB;
  var
        A: <тип>;
  ......
  procedure lok;
  var
        B: < тип >;
  .......
```

Переменная А – глобальная, так как описана в основной программе.

Переменная В – локальная; она описана в процедуре.



Обратим внимание на отличие формата описания функции. Кроме имени функции и формальных параметров с описанием их типов

!!! В разделе операторов должен находиться по крайней мере один оператор, который имени функции присваивает значение

Замечание. Если таких операторов несколько, то в точку вызова возвращается результат последнего присваиванивания.

!!! Вызываемый результат может иметь любой скалярный тип, типы string и указатель. Обратим внимание: результатом функции не может быть массив, множество или запись. Это очевидно, так как результатом функции должно быть одно значение, а массив, множество, запись – сложные типы, состоящие из множества элементов.

Обращение к функции (вызов функции) также, как и вызов процедуры, осуществляется по имени с указанием фактических параметров:

<имя функции> (<фактические параметры>);

Примеры стандартных функций

Арифметические функции находятся в модуле System.

Напомним, что модуль System подключается автоматически к каждой программе.

Abs (x: real): real;	Модуль
abs (x: integer): integer;	
arctan (x: real): real;	Арктангенс
cos (x: real): real;	Косинус
sin (x: real): real;	Синус
exp (x: real): real;	e^x
ln (x: real): real;	Ln x
int (x: real): real;	Вычисляет целую часть числа
flag (x: real): real;	Вычисляет дробную часть числа
sqr (x: real): real;	Квадрат числа x
sqr (x: integer): integer;	
sqrt (x: real): real;	Кореньквадратный из x
odd (x: integer): oolean;	X (нечетное)=true, X(четное)=false
random (x: word): word.	Генерирует псевдослучайное число из диапазона 0…x
random: real	Генерирует псевдослучайное число из диапазона 0…0,99
pi	Число ∏

Примеры функций из модуля CRT. Wherex: byte; Wherey: byte; Keypressed: Boolean;

Readkey: char; Это функции БЕЗ параметров.

Функции для работы со строками:

Copy (St, Poz, n) : string; Concat (st1, …, Stn: string) : string; И другие. Это функции С параметрами.

Пример 1.Возведение вещественного числа в вещественную степень x^y

Для примера вычислим значения x^3, x^4, x^5+x^6

```
program DemoPower;
        var
                x, sum: real;
        function Pow (x,y: real): real;
        {функция вычисляет x в степени y}
        begin
                Pow:=exp(y*ln(x));
        end;
Begin
        {начало основной программы}
        writeln ('Введите x');
        readln (x);
        writeln (Pow(x,3)); {вычисление и одновременный вывод на экран x^3 }
        writeln (Pow(x,4)); {вычисление и одновременный вывод на экран x^4 }
        {Внимание. Далее имя функции используется в выражении как операнд}
        sum:= Pow (x,5)+Pow(x,6); )); {вычисление x^5+x^6}
        writeln (sum);
End.
```

Иногда встречаются такие случаи, когда задача разбивается на подзадачи, которые имеют ту же структуру, что и основная задача.

В таких случаях используют механизм, который называется **рекурсией**.

Способ вызова подпрограммы, в котором подпрограмма вызывает сама себя, называют рекурсией.

Подпрограммы, реализующие рекурсию, называются **рекурсивными подпрограммами**.

Поясним механизм рекурсивных подпрограмм с помощью классического примера использования рекурсии.

Пример 1.Вычисление факториала числа.

Обоснование выбора способа реализации.

Обратим внимание на то, что вычислить факториал числа N можно следующим образом: **N! = N * (N-1)! = N * (N-1) * (N-2)! и так далее**

То есть для вычисления факториала числа N требуется вычислить факториал числа (N-1), для вычисления факториала числа (N-1) необходимо вычислить факториал числа (N-2) и так далее.

Заметим, что вычисление факториала числа сводится к вычислению факториала числа, на единицу меньшего самого числа.

Реализуем такой алгоритм с использованием механизма рекурсии.

Так как подпрограмма будет производить вычисление значения, то реализовывать ее будем в виде функции.

```
function Fact (n: byte) : integer;
begin
        if n = 0  then   Fact := 1
                 else   Fact := n * Fact(n-1);
end;
```

Здесь имени функции сразу присваивается результат вычисления.

При вызове функции Fact(n-1) согласно оператору Fact := n * Fact(n-1), где т – параметр функции, вместо n подставится параметр (n-1) и, следовательно, вычислится строка

n * (n-1) * Fact ((n-1) -1) и так далее.

Рекурсивное обращение к функции Fact будет продолжаться до тех пор, пока n не станет равным 0.

Косвенная рекурсия

Описанный выше механизм часто называют прямой рекурсии.

Существет еще один рекурсивный механизм - **<u>косвенная рекурсия</u>**.

Механизм применяется для реализации следующей ситуации.

Первая подпрограмма вызывает вторую, еще не описанную.

Продемонстрируем ситуацию на примере.

```
Program Kosv_Rec;
    ……
    procedure A (var x: real);
    begin
            ….
            B(z);
            …
    end;
```

```
procedure B (var y: real);
begin
        …
        A(z);
        …
end;
```

Здесь процедура A обращается к процедуре B и наоборот. Какую процедуру первой мы бы ни описали, в любом случае будет ошибка – обращение к еще не описанной процедуре. Для того, чтобы избежать такой ситуации, используется предварительное описание подпрограмм. Предварительное описание состоит из заголовка подпрограммы и директивы (указания компилятору) предварительного описания подпрограмм FORWARD. Позже подпрограмма описывается без повторения списка параметров и типа возвращаемого результата. Правильная реализация вышеприведенного примера будет выглядеть следующим образом.

```
Program Kosv_Rec;
……
procedure B (var y: real); Forward;
procedure A (var x: real);
begin
        ….
        B(z);
        …
end;
```

```
procedure B;
begin
        …
        A(z);
        …
end;
```

Стек

Стеком называется *динамическая структура данных*, у которой в каждый момент времени доступен только верхний (последний) элемент. Лучше всего понятие *стека* можно проиллюстрировать примером стопки книг: в стопку можно добавить еще одну книжку, положив ее сверху; из стопки можно взять, не разрушив ее, только верхнюю книгу. А для того, чтобы достать книжку из середины стопки, необходимо сначала убрать по одной все лежащие выше нее.

Последовательность обработки элементов *стека* хорошо отражают аббревиатуры **LIFO** (**L** *ast* **I** n **F** irst **O** ut - "последним вошел, первым вышел") и **FILO** (**F** irst **I** n **L** *ast* **O** ut - "первым вошел, последним вышел").

Реализовать *стек* можно любым удобным для программиста способом: например, массивом. Тогда началом *стека* (его "верхним" элементом) будет последний компонент массива, а освобождение *стека* будет происходить в направлении от конца массива к его началу. При такой реализации нет необходимости в постоянном перемещении компонент массива.

Операции

Для *стека* должны быть определены следующие операции:

`empty(<нач_стека>):boolean`	- проверка *стека* на пустоту;
`add(<нач_стека>,<новый_элемент>):<нач_стека>`	- добавление элемента в *стек* ;
`take(<нач_стека>):<тип_элементов_стека>`	- считывание значения верхнего элемента;
`del(<нач_стека>):<нач_стека>` .	- удаление верхнего элемента из *стека*.

- **Очередь**
- ***Очередью*** называется *динамическая структура*, у которой в каждый момент времени доступны два элемента: первый и последний. Из начала *очереди* элементы можно удалять, а к концу - добавлять. Примером *очереди* программистской вполне может служить *очередь* обывательская: скажем, в продуктовом магазине.
- Последовательность обработки элементов *очереди* хорошо отражают аббревиатуры **LILO** (**L** *ast* **I** n **L** *ast* **O** ut - "последним вошел, последним вышел") и **FIFO** (**F** irst **I** n **F** irst **O** ut - "первым вошел, первым вышел").
- Реализовать *очередь* также можно при помощи массива, хотя здесь уже не удастся полностью избежать перемещения его компонент. Пусть k -я компонента массива хранит начало *очереди*, а (k+s)-я - ее конец. Тогда можно приписать новый элемент *очереди* в (k+s+1)-ю компоненту массива, а при удалении элемента из начала *очереди* ее голова сдвинется в (k+1)-ю компоненту. В процессе работы может оказаться, что вся *очередь* "сдвинулась" к концу массива, и ее снова нужно вернуть к началу. В этом случае и потребуется s перемещений компонент массива (s - это текущая длина *очереди*). Однако наиболее эффективной снова будет реализация при помощи односвязного линейного списка

Операции

Для *очереди* должны быть определены следующие операции:

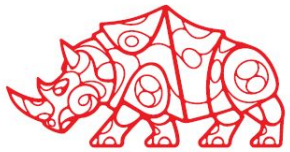
`empty(<нач_очереди>):boolean`	- проверка *очереди* на пустоту;
`add(<кон_очереди>,<нов_эл-т>):<кон_очереди>`	- добавление элемента в конец *очереди* ;
`take_beg(<нач_очереди>):<тип_эл-тов_очереди>`	- считывание значения первого элемента;
`take_end(<кон_очереди>):<тип_эл-тов_очереди>`	- считывание значения последнего элемента;
`del(<нач_очереди>):<нач_очереди>`	- удаление элемента из начала *очереди*.

Дек

Дональд Кнут[1] ввел понятие усложненной *очереди*, которая называется ***дек*** (*deque* – **D** ouble- **E** nded **QUE** ue – двухконцевая *очередь*). В каждый момент времени у *дека* доступны как первый, так и последний элемент, причем добавлять и удалять элементы можно и в начале, и в конце *дека*. Таким образом, *дек* – это симметричная двусторонняя *очередь*.

Реализация *дека* при помощи массива ничем не отличается от реализации обычной *очереди*, а вот в терминах списков *дек* удобнее представлять *двусвязным* (разнонаправленным) линейным списком (см. лекцию 10).

Набор операций для *дека* аналогичен набору операций для *очереди*, с той лишь разницей, что добавление и удаление элементов можно производить и в конце, и в начале структуры.



Домашние задание (Задание на самоподготовку)

1 выполнить задания несделанные во время практики