

## 20. Модули (Unit)

### 20.1 Что такое модуль?

Программный модуль (unit) представляет собой определенным образом оформленный набор описаний (констант, типов данных, переменных) а также процедур и функций, которые могут использоваться другими программами и модулями. Модуль в отличие от программы не может быть запущен на выполнение самостоятельно, он может только участвовать в построении программ и других модулей. Модуль предназначен не для непосредственного выполнения (как программа), а для того, чтобы экспортировать свои ресурсы для использования другими модулями и программами.

Модуль в TURBO PASCAL представляет собой отдельно хранимую (в виде исходника как файл с расширением \*.pas) и/или независимую откомпилированную программную единицу (с расширением \*.tpp или \*.tpp).

Все программные элементы модуля можно разбить на две части:

- программные элементы, предназначенные для использования другими программами или модулями, такие элементы называют видимыми вне модуля;
- программные элементы, необходимые только для работы самого модуля, их называют невидимыми или скрытыми.

В соответствии с этим модуль, кроме заголовка, содержит две основные части, называемые интерфейсом и реализацией.

1

### 20.2 Зачем нужны модули и какие есть средства, аналогичные (в какой-то мере) модулям

1) Borland Pascal обеспечивает доступ к большому числу встроенных констант, типов данных, переменных, процедур и функций. Их количество велико, однако, в каждой программе они редко используются все сразу. Поэтому они (встроенные константы, типы данных, переменные, процедуры и функции) разделены на связанные группы, называемые модулями и имеется возможность использовать только те модули, которые реально необходимы в программе. 2) Используя собственные модули, можно разбивать программу на отдельные части и строить программы практически любого размера. 3) При разработке программы большим количеством людей разделение программы на модули позволяет добиться того, чтобы решения, принимаемые одним программистом (например, выбор имен переменных), не влияли на работу других программистов.

Строить большие программы по модульному принципу можно и без использования модулей. Для этого есть, по крайней мере, следующие возможности:

- разбить исходный текст программы (на Паскале) на части и хранить каждую часть в отдельном файле; к основной части можно программе подключать с помощью директивы

{\$I имя файла}

Естественно, что компилироваться таким образом собранные программы будут значительно медленнее, чем неразделенные; кроме того, возможен конфликт имен, используемых в разных частях программы

можно указывать путь

При этом для гибкости сборки можно использовать условную компиляцию:

```
{ $define A}
.....
{ $ifdef A}
// { $include имя_1}
{ $else}
{ $include имя_2}
{ $endif}
```

– **использовать так называемые оверлеи (overlay)**; при этом программа состоит из резидентной части (постоянно находящейся в памяти) и набора транзитных (замещающих друг друга в памяти) частей; все части программы при этом должны быть специальным образом оформлены и наименьшей оверлейной единицей является модуль; для организации таких программ имеется специальный стандартный модуль OVERLAY, который в данном случае д.б. обязательно подключен к программе; естественно, что выполняться оверлейная программа будет значительно медленнее, чем неоверлейная, за счет времени подкачки в память транзитных частей;

– **использовать динамически связываемые библиотеки - DLL** (Dynamically Linked Libraries); такая возможность есть только при работе в защищенном режиме DOS или под MS Windows; структура DLL похожа на структуры программы, но вместо слова program используется слово library; при компиляции такой структурной единицы получается файл с расширением .dll; особенностью использования DLL является то, что:

- 1) код из DLL существует отдельно от программы (соответственно файл программы не увеличивается);
- 2) при т.н. статической компоновке код из DLL прикомпоновывается статически к программе (к файлу \*.exe) и вся эта сборка загружается в оперативную память, при этом все необходимые программе функции из DLL уже на месте в памяти;
- 3) при динамической компоновке DLL связываются с программой (прилинковываются) только динамически (с помощью системного вызова LoadLibrary()) в оперативной памяти; адрес нужной функции в библиотеке определяется с помощью системного вызова GetProcAddress() (LoadLibrary() и GetProcAddress() - в библиотеке kernel32.dll)
- 4) в отличие от модулей DLL могут экспортировать лишь процедуры и функции (модули могут экспортировать данные, код, типы).

## 20.3 Структура модуля (секции)

Структура модуля аналогична структуре программы, однако есть несколько существенных различий.

В общем случае можно выделить три части, из которых состоит модуль:

- 1) описания, предназначенные для использования в других модулях и программах (видимые в других модулях);
- 2) описания, которые должны быть видимыми только в данном модуле;
- 3) действия, которые надо выполнить для инициализации переменных, используемых в модуле, и данные, которые надо инициализировать (в самом начале работы программы).

Поэтому модуль обычно имеет следующую структуру (3 части, 3 секции):

модуль должен быть помещён в файл, имя которого совпадает с именем модуля (с расширением \*.pas)

unit <имя модуля/идентификатор>; {заголовок модуля (обязателен)}

interface { Секция интерфейса (**обязательна**)}

uses <список импортируемых (подключаемых) модулей>;

const .....	{ Описание	}
type .....	{ видимых/глобальных	}
var .....	{ элементов данного модуля	}
procedure .....	{	}
function .....	{	}

только заголовки

implementation { Секция реализации (**обязательна**)}

uses <список импортируемых (подключаемых) модулей>

label ...	; { Описание	}
const ..	; { внутренних (невидимых для других модулей)	}
type .	; { элементов модуля, а так же описание (реализация)	}
var ....	; { всех тех процедур и функций, для которых	}
procedure .....	{ в интерфейсной части <u>были приведены только заголовки</u>	}
function .....	{	}

полные описания

Begin {Секция инициализации модуля (**не обязательна**)}

{ действия по инициализации элементов модуля}

end.

Если отсутствует секция инициализации, то должно быть пропущено слово begin и действия по инициализации

В частном случае модуль может быть пустым (пустой модуль):

```
unit <имя модуля>; {заголовок модуля}

    interface

    implementation

end.
```

### Интерфейсная секция

Все, что здесь перечисляется (константы, типы, переменные, заголовки процедур и функций), будет видимо из программы и модулей, использующих данный модуль. Другими словами, здесь описывается интерфейс (стыковочные узлы) данного модуля с другими модулями и программами.

NB: В этой секции обычно приводятся только заголовки подпрограмм. Их полное описание обычно помещают в секцию реализации.

### Секция реализации

Описанные в интерфейсной секции обычные процедуры и функции, должны описываться полностью (заголовок + описания + вложенные процедуры и функции + тело) в секции реализации. Заголовок procedure/function должен быть или идентичным тому, который указан в секции интерфейса, или иметь краткую форму (без списка формальных параметров и круглых скобок).

Только здесь описываются (и заголовок и тело) те процедуры и функции, которые надо скрыть от использования сторонними модулями, при этом заголовки этих процедур и функций не задаются в секции интерфейса. Обычно так поступают с теми процедурами и функциями, которые предназначены для работы самого модуля.

Подпрограммы, локальные для секции реализации (то есть не описанные в секции интерфейса), должны иметь полный (несокращенный) заголовок.

Таким образом, кроме процедур и функций (заголовки которых приведены в интерфейсной секции) в секции реализации также описываются константы, переменные, процедуры и функции, являющиеся глобальными/видимыми по отношению к подпрограммам секции реализации, но являющиеся одновременно локальными (доступными только в данном модуле), то есть недоступными (невидимыми) для основной программы и других модулей.

### Секция инициализации (в Delphi есть еще секция finalization)

Обычно вся секция реализации модуля заключена между зарезервированными словами implementation и end. Однако, если перед end поместить зарезервированное слово begin, а между ними - операторы, то получившийся составной оператор, очень похожий на основное тело программы, становится секцией инициализации модуля (initialization).

Секция инициализации представляет собой место, где инициализируются структуры данных (переменные), которые использует модуль или которые он делает доступными программе, использующей данный модуль. Можно, например, использовать эту секцию для открытия файлов, которые программа будет использовать позднее, или для инициализации указателей.

**Замечание.** При выполнении программы, использующей некоторый модуль, секция инициализации этого модуля вызывается *перед* запуском основного тела программы. Если программа использует более одного модуля, то секции инициализации всех модулей вызываются (в порядке, обратном тому, в котором модули указаны в операторе uses в программе) перед тем, как выполнить основное тело программы.

Подключение других модулей к данному (модулю)

Программный модуль может использовать другие модули, для этого они определяются в операторе uses. Оператор в описании модуля uses (если он имеет место) может содержаться в двух местах (в вызывающем модуле).

**Во-первых**, он может следовать сразу после ключевого слова *interface*. В этом случае любые константы и типы данных, описанные в интерфейсной секции этих (подключаемых) модулей, могут использоваться в любом месте (в любой из трёх секций) данного (подключающего) модуля и, следовательно, во всей основной программе так же.

**Во-вторых**, он может следовать немедленно за ключевым словом *implementation*. В этом случае предполагается, что все описания из интерфейсных частей этих импортируемых модулей не используются в интерфейсной части данного модуля и, следовательно, не могут быть использованы в основной программе, а могут использоваться только в секции реализации данного модуля. Причем об использовании импортируемых таким образом модулей не будет знать ни один модуль кроме данного (импортирующего).

20.4 Ссылки на описания модуля. Области видимости.

Как только модуль включается в программу (через uses), все константы, типы данных, переменные, процедуры и функции, описанные в секции интерфейса этого модуля, становятся доступными для этой программы.

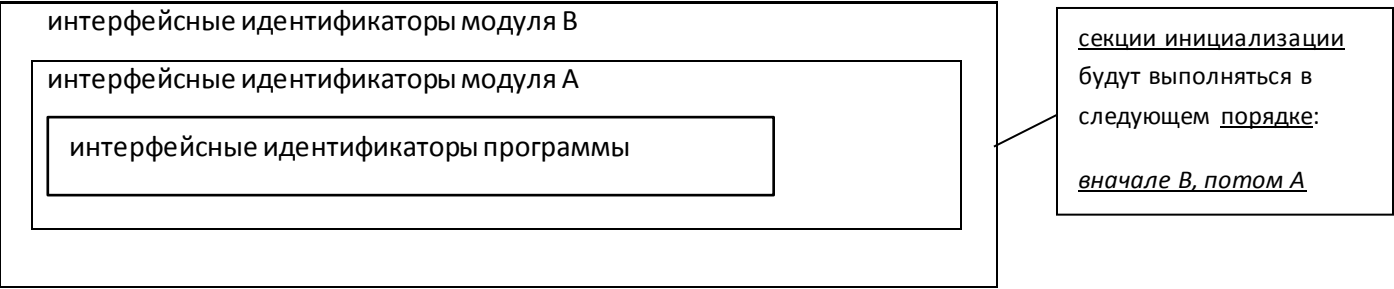
В том случае, если имена переменных и функций в интерфейсной части модуля и в программе, использующей этот модуль, совпадают, то действует следующее правила видимости имен:

- интерфейсные идентификаторы модулей, указанных в списке USES других модулей (т.е. используемых в других модулях), образуют самый внешний (глобальный) блок программы (то, что описано в нем, видно всем);
- интерфейсные элементы модулей, использующих с помощью USES другие модули, образуют блок, вложенный в первый блок, и т.д.
- самый внутренний блок – программа, использующая модули (то, что описано в ней, не видно за ее пределами, в том числе не видно в подключенных к программе модулях).

Например:

Program P;  Uses A;  ...  end.	Unit A;  Interface  Uses B;  ...  end.	Unit B;  Interface  ...  end.
--	--	---

то вложенность блоков будет иметь вид:



Пример показывает:

- из программы видно то, что и в А, и в В;
- из А видно то, что в В, но не видно (нельзя обратиться) то, что в программе;
- из В не видно того, что в А и не видно того, что в программе.

Таким образом, при совпадении имени (идентификатора) в программе и модулях А и В, обращение в программе будет происходить к переменной, описанной в программе (действует старое правило видимости переменных в Паскале – локальная переменная перекрывает видимость глобальной).

Для обращения в теле основной программы к переменной или функции, описанной в модуле, необходимо применить составное имя, состоящее из имени модуля и имени переменной, разделенных точкой.

Например, пусть имеется модуль, в котором описана переменная К:

unit A;	unit B;
interface	interface
uses B;	var
var	K: String;
K: Integer;	.....
implementation	end.
.....	
end.	

В модуле А локальная (для него) переменная К будет перекрывать область видимости глобальной переменной К (из модуля В).

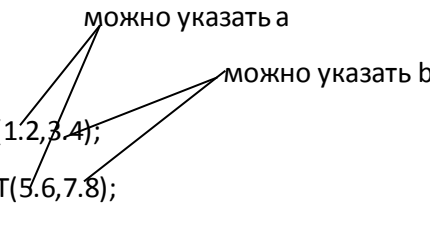
Пусть программа Р, использующая этот модуль, также содержит переменную К. Для того, чтобы в программе Р иметь доступ к переменной К из модуля А, необходимо задать составное имя вида <имя модуля>.К:

```

Program P;
uses A;
var
  K: Char;
begin
  k := 'a'; { обращение из программы к переменной К, описанной в программе }
  A.k := 25; { обращение из программы к переменной К, описанной в модуле А }
  B.k := '25'; { обращение из программы к переменной К, описанной в модуле В }
end.
```

Следует отметить, что имя модуля (в составном имени?) может предшествовать любому идентификатору: константе, типу данных, переменной или подпрограмме.

## 20.5 Пример создания модуля

<pre>Unit MyUnit;  Interface  Function f_ADD(x,y:real):real;  Function f_MULT(x,y:real):real;  Implementation  Function f_ADD; //сокращенный заголовок  Begin      F_ADD:=x+y;  End;  Function f_MULT; //сокращенный заголовок  Begin      F_MULT:=x*y;  End;  End.</pre>	<pre>Program p_unit;  Uses MyUnit;  Var      a,b:real;  begin      a:=f_ADD(1.2,3.4);     b:=f_MULT(5.6,7.8);      ...      Writeln('a=',a:8:2,'b=',b:8:2);  end.</pre> 
---	--

7 этот модуль надо записать в файл MyUnit.pas

**Вопрос:** что изменится (видимость чего и как изменится), если использовать функции (в модуле) без параметров, а переменные *x* и *y* объявить глобальными а) в программе или б) в модуле

## 20.6 Использование модулей. Режимы Compile, Build и Make при компиляции модулей

При использовании модулей предполагается, что модули, которые используются (через Uses) в программе, должны быть уже оттранслированы и храниться, как машинный код, а не как исходный код на Паскале.

При компиляции программы компилятор обычно не перекомпилирует модуль (пока он не изменяется), поэтому использование модулей в программе ускоряет процесс ее построения.

Только тогда, когда в интерфейсную часть модуля вносятся изменения, другие модули, использующие этот модуль, должны быть заново скомпилированы. При использовании команд Make или Build компилятор делает это автоматически. Однако если изменения коснулись только секции реализации или секции инициализации, то другие модули, в которых используется этот модуль, перекомпилировать не нужно.

Borland Pascal предоставляет пользователю ряд стандартных модулей, таких как System, Crt, DOS, Graph, WinCrt и др.

Модуль System всегда подключается автоматически. В этом модуле сосредоточены все стандартные функции и для поддержки таких средств, как файловый ввод-вывод, обработка строк, операции с плавающей запятой, динамическое распределение памяти и других этот модуль реализует весь нижний уровень.

Чтобы найти файл, содержащий уже скомпилированный модуль, компилятор усекает указанное в операторе `uses` имя модуля до первых восьми символов и добавляет расширение файла. Если целевой платформой является DOS, расширением будет `*.TPU`. Если целевая платформа - Windows, то расширением файла будет `*.TPW`. Если целевой платформой является защищенный режим DOS, то расширением файла будет `*.TPP`. Хотя имена файлов усекаются, в операторе `uses` должен указываться полный идентификатор модуля. Например, если в программе используется предложение:

**Uses unit\_A;**

то компилятор будет при перетрансляции программы пытаться найти на диске файл с именем `unit_a.tpu`. Если есть необходимость хранить модуль в файле с другим именем, например, `unit_B`, то следует использовать директиву компилятора `$U` для переопределения имени TPU-файла. Эта директива задает "настоящее" имя TPU-файла и должна размещаться непосредственно перед именем подключаемого модуля в предложении `USES`. Так в данном случае:

**uses {\$U unit\_B} unit\_A;**

компилятор будет для подключения модуля `unit_A` пытаться найти файл:

**unit\_B.tpu.**

в нее включены некоторые стандартные модули

Если для трансляции программы с модулями задана опция компилятора **Compile**, то предполагается, что все используемые модули уже откомпилированы. Компилятор выполняет поиск **поиск файла с модулем** (откомпилированного модуля) по следующей схеме (3 шага):

- 1) просматривает библиотеку исполняющей системы (обычно `turbo.tpl`, `tpw.tpl` или `tppl`);
- 2) если модуль не найден в библиотеке исполняющей системы, то поиск продолжается в текущем каталоге;
- 3) если модуль не найден в текущем каталоге, то поиск продолжается в каталогах, заданных с помощью установок в среде `Options | Directories | Unit Directories`

Если ничего не найдем - то ошибка компиляции.

Если для трансляции программы с модулями заданы опции компилятора **Make** или **Build**, то предполагается, что и программа и какие-то (не все) модули были изменены. В этом случае описанные выше 3 шага поиска проводятся с целью найти не файл с расширением TPU, а в поисках исходного текста измененных модулей, которые д.б. перекомпилированы перед (при) трансляцией самой программы:

- если выбран режим **Make** (условная компиляция), то будет проверяться (как - см. 3 шага поиска выше по тексту) наличие всех необходимых `tpu`-файлов и если какой-то файл не найден или соответствующий исходник был изменен (после последней компиляции), то **исходные** файлы только этих модулей (и тех модулей, которые от них зависят) будут отыскиваться и перекомпилироваться;

- если выбран **Build** (безусловная компиляция), то будет выполнена принудительная перекомпиляция всего, что входит в проект (программа + текст модулей)).



## 20.7 Косвенные и перекрестные ссылки на модули

Вначале рассмотрим случай косвенного использования модуля. Например, пусть имеются два модуля:

<pre>unit A;   interface   ..... end.</pre>	<pre>unit B;   interface     uses A;   ..... end.</pre>
---	---

Здесь, как ясно видно, модуль В использует модуль А, и если в вызывающей программе будет использовано предложение uses вида:

**USES B; = Uses B,A;**

то все равно при этом неявно(косвенно) будет использован и модуль А, и он, естественно, будет тоже прикомпонован к вызывающей программе.

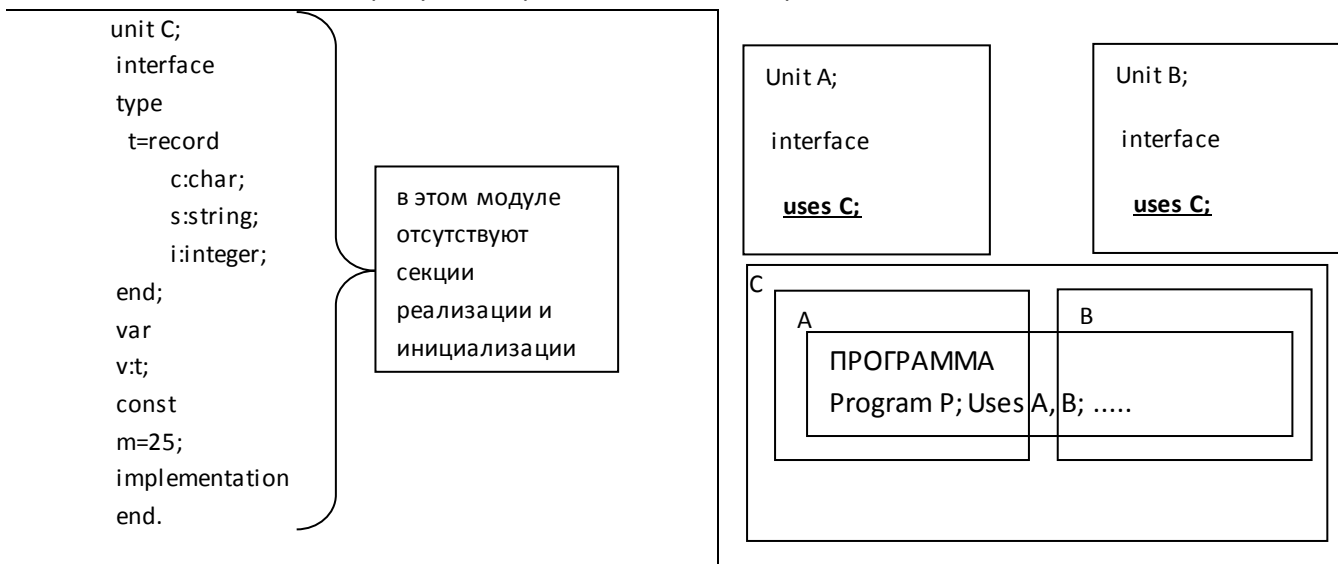
Для случая перекрестных ссылок вначале рассмотрим следующий пример. Пусть имеются два следующих модуля:

<pre>unit A;   interface     uses B;   ..... end.</pre>	<pre>unit B;   interface     uses A;   ..... end.</pre>
---	---

Здесь модули А и В с помощью косвенной рекурсии обращаются сами к себе. Такие отношения между модулями возникают тогда, когда **данные (или действия) из одного модуля нужны в другом, а в другом нужны данные или действия из первого**. По правилам Паскаля такие перекрестные ссылки на модули недопустимы, т.е. недопустимо прямое или косвенное обращение модуля к самому себе через интерфейсную секцию.

В этом случае можно завести третий модуль и поместить в него все те типы, переменные, константы, процедуры и функции из первых двух модулей, которые ссылаются друг на друга. После этого надо удалить эти общие ресурсы из первых двух модулей и подключить заведенный третий модуль к первым двум.

Таким приемом часто пользуются, когда нужно какие-то типы, переменные и константы сделать общими для программы и/или разных модулей. Проблема здесь в том, что само понятие "модуль" было задумано для того, чтобы сделать модули независимыми от глобальных переменных в основной программе. Выход в данном случае состоит в том, чтобы создать модуль из одних только этих глобальных описаний и подключать его везде, где требуется обращение к общим переменным, типам и константам:



20.8Пример модуля (стек)

Рассмотрим пример построения модуля, объединяющего в себе средства работы со структурой данных "стек". В общем случае стек (магазин) работает по принципу "первым пришел - последним ушел" (LIFO), причем доступ к элементам стека возможен только через одно место - через т.н. голову стека, которая изменяет свое положение после каждой записи/чтения в/из стек/стека. В зависимости от реализации стек м.б. бесконечным (при реализации в виде динамической связанной структуры, где память выделяется и освобождается динамически - при выполнении программы/модуля) и конечным (при реализации в виде статической структуры - массива, для которой память выделяется только один раз - при компиляции программы).

Чаще всего стек используется в 2 случаях:

1) При интерпретации выражений, записанных в виде так называемой обратной польской записи, при выполнении которой выражение просматривается слева направо и до определенного момента (пока не встретится знак операции) операнды заталкиваются в стек, а после поступления знака операции операнды извлекаются и вместо них заталкивается в стек результат операции и т.д.

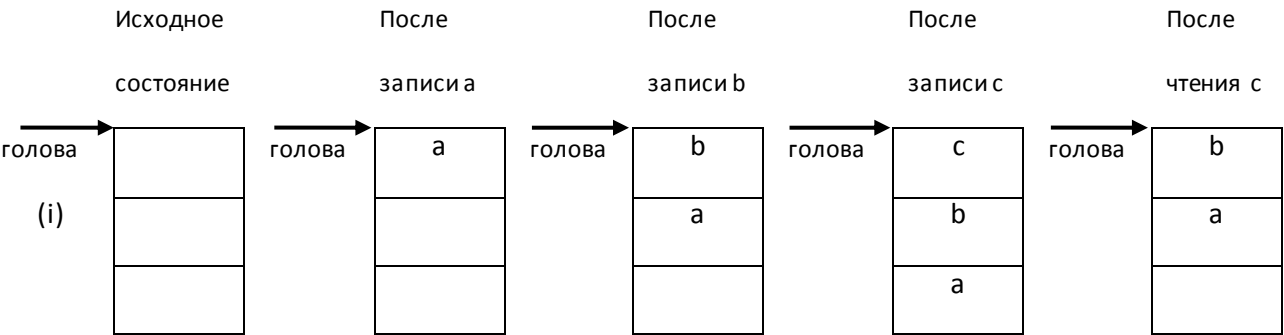
(a + b) \* c -----> ab+c\* (обратная польская запись)  
инфиксная            постфиксная  
форма записи        форма записи

2) При работе в таких ситуациях, где надо одновременно держать в памяти несколько поколений какого-то объекта, при этом доступен будет лишь самый «молодой» потомок. Пример – при каждом рекурсивном вызове копии локальных данных вызываемой подпрограммы сохраняются в стеке.

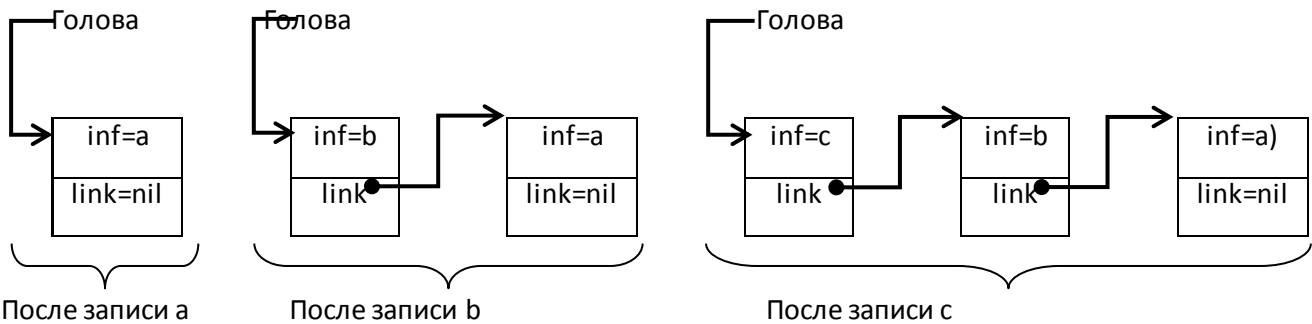
Основными операциями над стеком являются: а) включение элементов в стек (push); б) исключение элементов из стека (pop); в) чтение информации из головы стека без извлечения (peek). Все эти операции выполняются через вершину стека, которая адресуется через специальный указатель (голова стека).

Пример: пусть в стек надо записать последовательность символов a,b,c,.....

Обобщенный вид стека при выполнении действий:



Реализация стека в виде динамически связанной структуры:

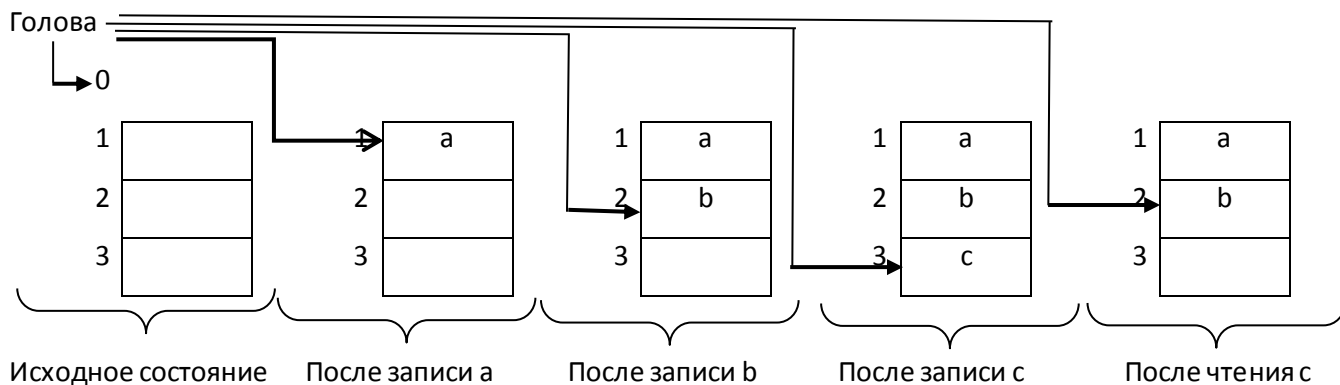


Реализация в виде массива из 3-х элементов.

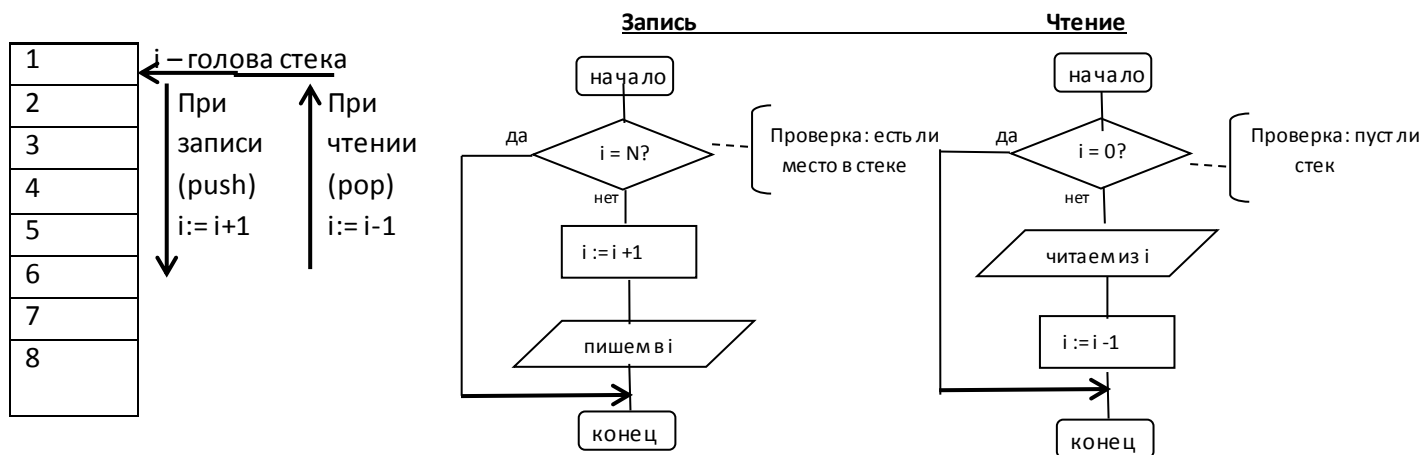
При этом выполняются следующие правила:

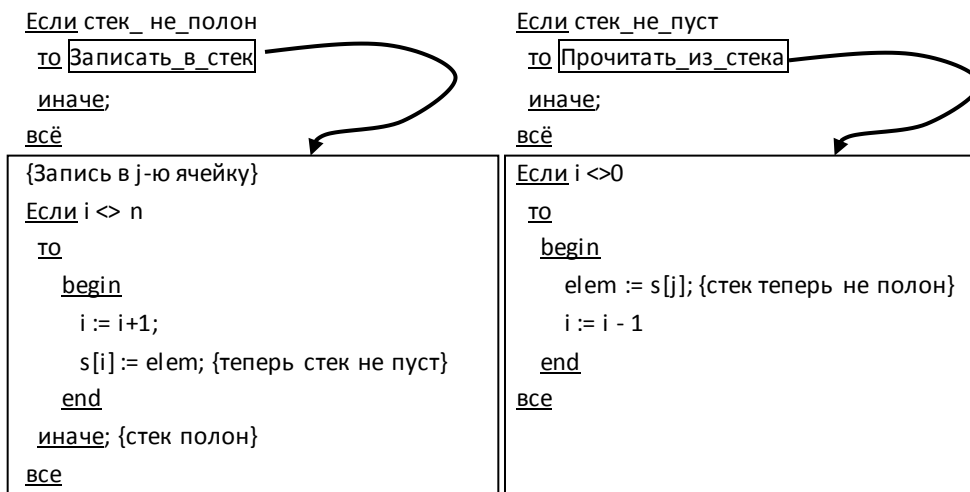
- при включении нового элемента в стек сначала указатель (голова) (у нас для нее используется переменная  $i$ ) увеличивается (начальное значение = 0) (увеличивается только если  $i < N$ ) на 1 (изменяется в сторону больших адресов - стек, как говорят, растет вниз). Затем по новому значению указателя стека в голову стека (в ячейку, адресуемую  $i$ ) записывается информация о новом элементе;
- при выполнении исключения (чтения) вначале из ячейки, адресуемой текущим значением указателя стека (у нас для нее используется переменная  $i$ ) читается (извлекается) (только если  $i > 0$ ) информация о доступном (исключаемом) элементе. Затем значение указателя уменьшается на 1 (если  $i \geq 1$ ) (т.е. изменяется в сторону нижней границы);
- стек считается пустым (при условии, что он растет вниз), если указатель стека равен 0, т.е. смещен вниз на 1 относительно нижней границы индекса носителя стека (массива);
- стек считается полным (при условии, что он растет вниз), если указатель стека равен  $N$ , т.е. равен верхней границе индекса носителя стека (массива).

В нашем случае указатель стека (вершина  $i$  стека) будет отмечать (адресовать) последний занятый элемент стека (откуда читать). Писать в стек при этом надо будет в ячейку с номером  $i+1$ .



Мы будем в нашем примере реализовывать стек с помощью массива (массив, как известный Паскалю тип данных будет использован в качестве носителя для значений неизвестного Паскалю типа данных). При этом способ реализации скроем в секции реализации, так что в программе, использующей наш модуль, ничего не изменится при изменении реализации стека.





Вопрос: Что будет, если описание переменной elem поместить:

- 1) В секцию реализации
- 2) В основную программу

Unit stack;

Interface

Var elem:byte; {то, что считывается и записывается в стек}

Procedure push(elem:byte);  
Procedure pop(var elem:byte);

Implementation

```

Const
  N=10; {размер стека}
Var
  i: byte; {голова стека}
  s: array[1..N] of byte; {носитель стека}
Procedure push; {помещение эл-та в вершину стека.}
begin
  if (i<>N)    {проверка: полон или не стек?}
  then
  begin
    i := i+1;
    s[i] := elem;
  end
  else Writeln('Стек полон');
end;
Procedure pop(var elem : byte); {Чтение эл-та из стека}
begin
  if (i<>0)    {проверка: пуст или нет стек?}
  then
  begin
    elem := s[i];
    i:= i - 1;
  end
  else Writeln('Стек пуст');
end;
  
```

begin { Инициализация }

i := 0; {Начальное значение головы стека-массива}

end.

т.к. характеристики носителя и действия с ним скрыты в секции реализации, то за пределами модуля не будет видно, как мы реализовали стек (как массив

Заголовок приведен без списка параметров, потому что полный заголовок приведён в интерфейсной части

Приведем программу, которая использует этот модуль. Она добавляет в стек три числа, потом одно удаляет и выводит на экран:

```
Program P;  
  uses  
    stack;  
begin  
  push(1);  
  push(2);  
  push(3);  
  pop(elem);  
  writeln('Считанный элемент = ', elem);  
end.
```