

Увод

Настоящото ръководство е предназначено за студентите от МГУ "св. И. Рилски", изучаващи езика за програмиране Pascal. То е съобразено с приетите програми за това обучение. Поради това в ръководството не са включени, някои средства на езика като: типове изброим, множество и указатели; модулно и обектно-ориентирано програмиране, които не са включени в програмата за обучение на студентите поради малкия брой часове. За любознателните студенти, желаещи по-подробно запознаване с езика Pascal препоръчвам книгите

- Програмиране на Pascal с 99 примерни програми. - автори Радмила Стоянова и Гочо Гочев;
- Програмиране на Pascal - автор Магдалина Тодорова, София 1993.

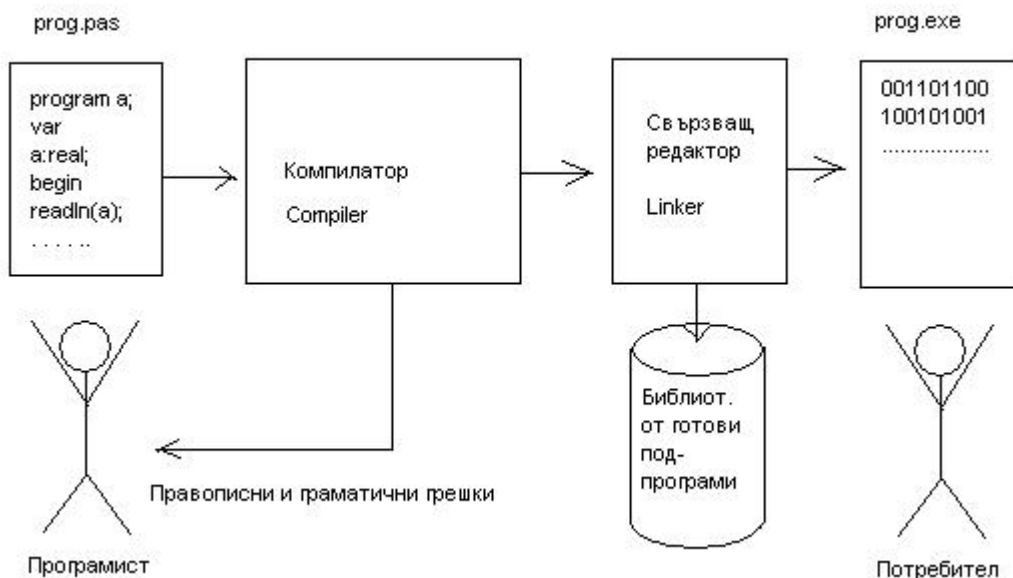
В ръководството освен описанието на езика са включени и решени примери, като в решението са включени не само програмите, но и анализ на данните и описание на алгоритмите, защото целта е не само да се научи езика Pascal, но и да се помогне на студентите да се научат да мислят алгоритмично. Тези примери са предназначени за компилиране и изпълнение в среда на операционната система MS Windows, поради което е прибавена декларацията

```
uses WinCrt;
```

след заглавието на програмите. Така примерите ще могат да се компилират с Borland Pascal v.7.0 за Windows. Ако се създава конзолно приложение (например за операционна система DOS), тази декларация трябва да се премахне.

1. Среда за програмиране Borland Pascal v.7.0. Въвеждане и редактиране на програма. Компилиране. Използване на Help.

Една програма написана на езика Pascal не може веднага да се изпълни от компютъра. Както знаете процесорът на компютъра може да изпълнява само програми на неговия машинен език - така наречените изпълними програми. Това обикновено са файловете с разширение .EXE и .COM. Така че програмата написана на езика Pascal трябва да се преведе на машинен език. Преводът на програмата се извършва от друга програма наречена транслятор.



След като програмистът напише програмата на езика Pascal, той я подава на транслятора, който по време на превода извършва проверка на правописа и граматиката на програмата. Ако намери грешки, издава съобщения за грешка и посочва, къде е грешката. Програмистът отстранява грешката и отново подава програмата към транслятора. Това продължава, докато програмистът отстрани всички правописни и граматически грешки от програмата и едва тогава тя се транслира на машинен език.

Транслаторите биват два вида: компилатори и интерпретатори. Компилаторите превеждат цялата програма на машинен език и за да се ползва по-нататък тази програма не са необходими нито компилатора, нито изходната програма на езика Pascal, нито програмиста. Интерпретаторът превежда програмата команда по команда, като след като се преведе една команда до няколко машинни команди, те веднага се изпълняват. При тях всъщност няма цяла изпълнима програма. Езикът Pascal е реализиран чрез компилатори.

Транслаторите са едни сложни програми и те се създават от специализирани софтуерни фирми. Компилатори за Pascal за компютри PC се създават главно от фирмата Borland. Тази фирма предлага не само компилатор, а цяла програмна система наречена среда за програмиране. Тази система непрекъснато се модифицира и така се получават отделните нейни версии - Turbo Pascal, Borland Pascal, Delphi.

Средата за програмиране Borland Pascal v.7 включва следните основни средства:

- компилатор - compiler;
- свързващ редактор - linker;
- езиково-ориентиран текстови редактор - editor;
- библиотеки от готови подпрограми - libraries;
- анализатор на програмата - debugger;
- помощна система - help.

Компиляторът всъщност превежда програмата до един междинен език наречен обектен код - файл с разширение .OBJ. Ако той срещне правописна или граматична грешка, спира компилирането, издава съобщение за грешка и посочва реда, в който е грешката.

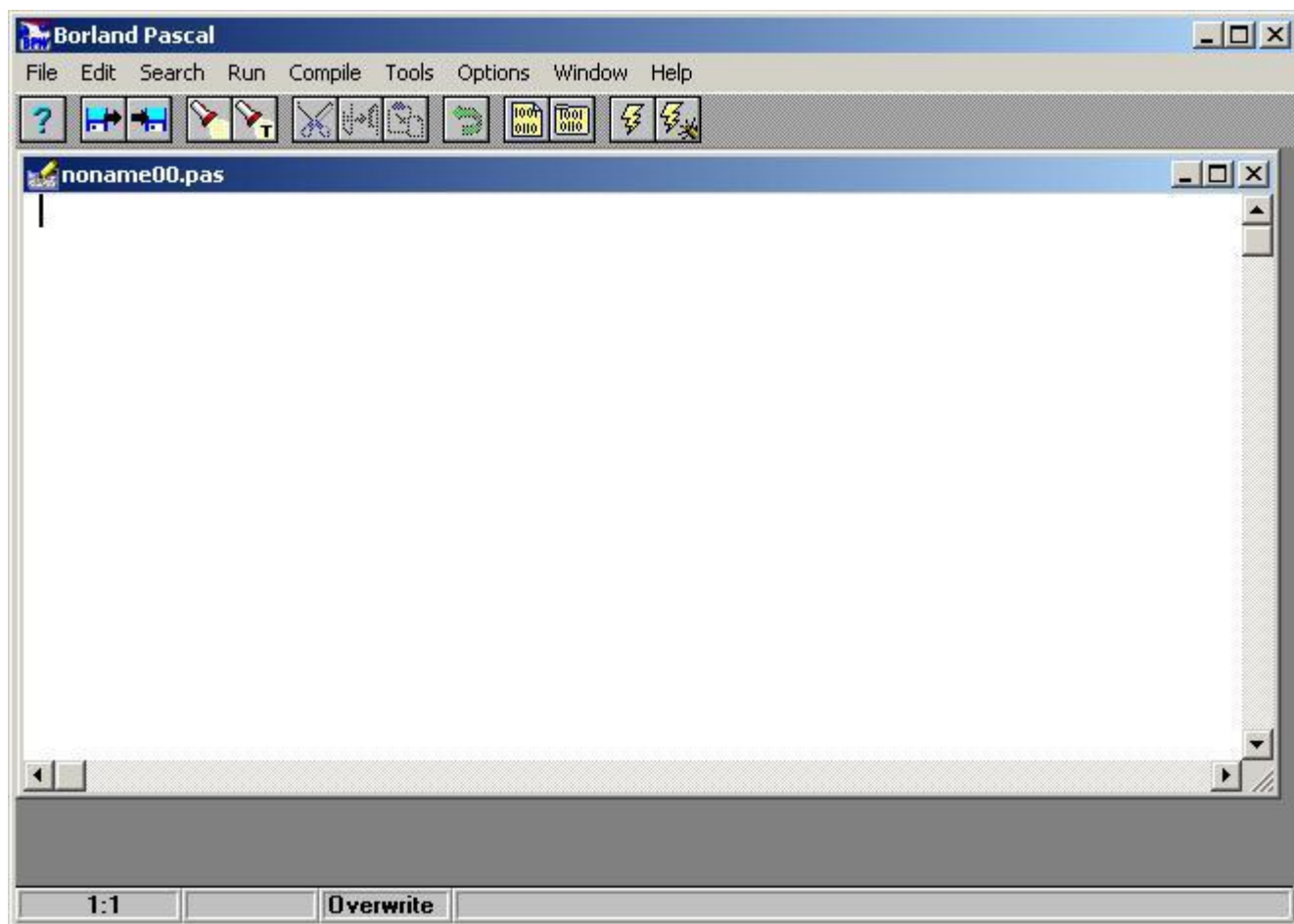
Свързващият редактор превежда програмата от обектен код до изпълнима програма (обикновено с разширение .EXE), като я свързва с библиотеките от тези готови подпрограми, които се ползват в програмата. Освен това той свързва в едно цяло програма състояща се от няколко файла. По време на свързване също могат да се открият грешки. Например, ако сте сбъркали името на някоя библиотечна подпрограма. В такъв случай свързващият редактор издава съобщение за грешка при свързване и прекратява генерирането на изпълнимата програма.

Текстовият редактор служи програмиста да въведе текста на програмата, да го съхрани във файл и да го отредактира. Този редактор е езиково-ориентиран и форматира отделните думи в програмата според тяхното място в граматиката на Pascal с различен цвят, удебеляване, наклоненост. Така, ако програмистът е достатъчно наблюдателен, може да открие правописни грешки в програмата си преди още да я компилира. Съхранените на диска файлове с програми на езика Pascal имат разширение .PAS.

Библиотеките от готови подпрограми съдържат подпрограми, които най-често се използват, например, подпрограми за вход/изход, тригонометрични функции, подпрограми за изчертаване на графични примитиви - линия, дъга, правоъгълник и т. н.

Анализаторът на програмата служи да подпомогне програмиста да открие логическите грешки в програмата. Help системата съдържа информация за средата за програмиране, за езика Pascal и за библиотеките от готови подпрограми.

Обикновено средата за програмиране се намира в папка \BP на някой от твърдите дискове (обикновено C:). Тя има две модификации: текстова - за стартиране в команден режим на MS Windows и прозоречна за стартиране в нормалния графичен режим на MS Windows. Стартирането на текстовата модификация става чрез BP.EXE от подпапка \BIN, а стартирането на прозоречната модификация става чрез BPW.EXE от същата подпапка. Удобно е да си направите шорткьти върху десктопа на тези два файла. Ето как изглежда графичната среда:



Когато се стартира BPW, се отваря прозорец, който има: меню, лента с бутони, един или няколко подчинени прозорци и статусна лента. Текстът на програмата се въвежда в подчинен прозорец, който преди съхраняване във файл има служебно име например noname00.pas. След съхраняване той получава името на файла.

Някои управляващи клавиши на текстовия редактор:

- Backspace - изтрива символа пред текстовия курсор;
- Delete - изтрива символа след текстовия курсор;
- Home - текстовият курсор скача в началото на реда;
- End - текстовия курсор скача в края на реда;
- Page Up - превърта програмата един прозорец нагоре;
- Page Down - превърта програмата един прозорец надолу;
- Insert - превключва редактора от режим на вмъкване (Insert в статусния ред) в режим на заместване (Overwrite в статусния ред) и обратно. В режим на вмъкване, когато въвеждате текст, текста надясно от курсора се изтласква надясно и с Enter се отваря нов празен ред. В режим на заместване новият текст замества стария текст;
- Ctrl-Home - курсорът скача в началото на програмата;
- Ctrl-End - курсорът скача в края на програмата;
- Ctrl-Y - изтрива реда от програмата, в който се намира курсора.

Команди за редактиране на текста на програмата. Текстовият редактор ви дава възможност да копирате, преместите или изтриете част от текста на програмата. За тази цел вие първо трябва да маркирате този текст. Маркирането става или с влачене на

мишката през текста, или чрез Shift и стрелките. Маркираният текст се оцветява в черно.

Копиране на текст става по следния начин:

- маркирате текста, който ще копирате;
- изберете команда Copy от менюто Edit или натискате отляво надясно седми бутон от лентата с бутони.
- позиционирате курсора там, където искате да бъде копието;
- изберете команда Paste от менюто Edit или натискате отляво надясно осми бутон от лентата с бутони..

Преместването на текст става по следния начин:

- маркирате текста, който ще преместите;
- изберете команда Cut от менюто Edit или натискате отляво надясно шести бутон от лентата с бутони.
- позиционирате курсора там, където искате да бъде преместеното;
- изберете команда Paste от менюто Edit или натискате отляво надясно осми бутон от лентата с бутони..

Изтриването на текст става по следния начин:

- маркирате текста за изтриване;
- натискате клавиша Delete..

Някои команди на менюто.

Команди на опция File:

New - отваря нов подчинен прозорец за въвеждане на текст на програма в друг файл.

Open - зарежда файл с програма на Pascal от диск в нов подчинен прозорец. Първо се показва диалогова кутия, в която избирате устройство, папка и файл за зареждане.

Save - съхранява програмата от текстовия редактор във файл на диск. При първото съхраняване на нова програма от вас се изисква да въведете име на файла. Името трябва да е на латиница до осем букви дълго без интервали в него. Няма нужда да пишете разширение, .PAS се добавя автоматично. При повторно съхраняване програмата се съхранява със старото си име на файл.

Save as - съхранява програмата под ново име. Трябва да въведете това ново име.

Save all - съхранява програмите от всички подчинени прозорци във файлове на диск.

Print - отпечатва текста на програма в активния подчинен прозорец.

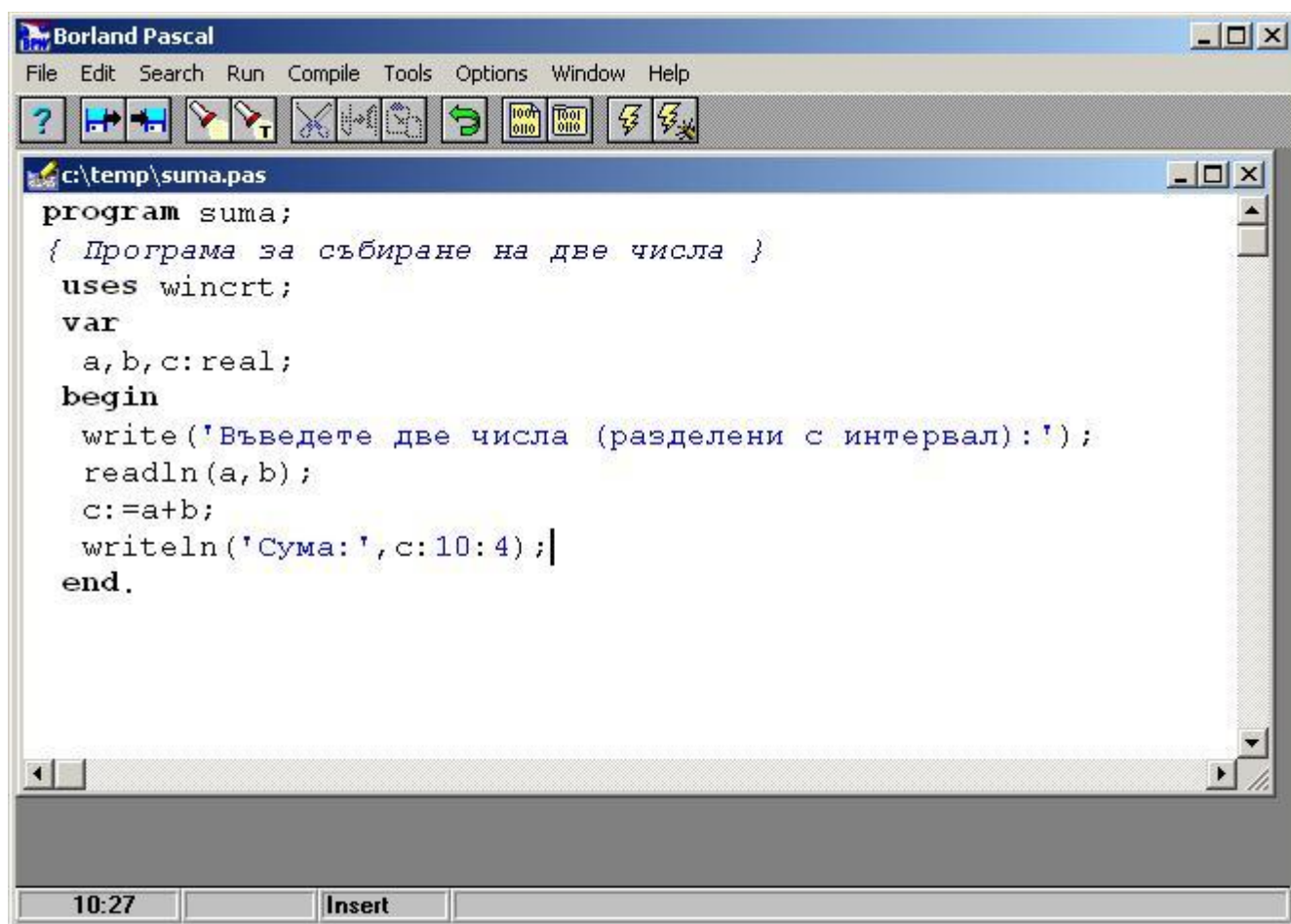
Printer setup - задавате на кой принтер да се отпечатва.

Exit (Alt-F4) - излизате окончателно от Borland Pascal. Ако не сте съхранили последните промени в програмата, ще бъдете попитате дали да съхраните.

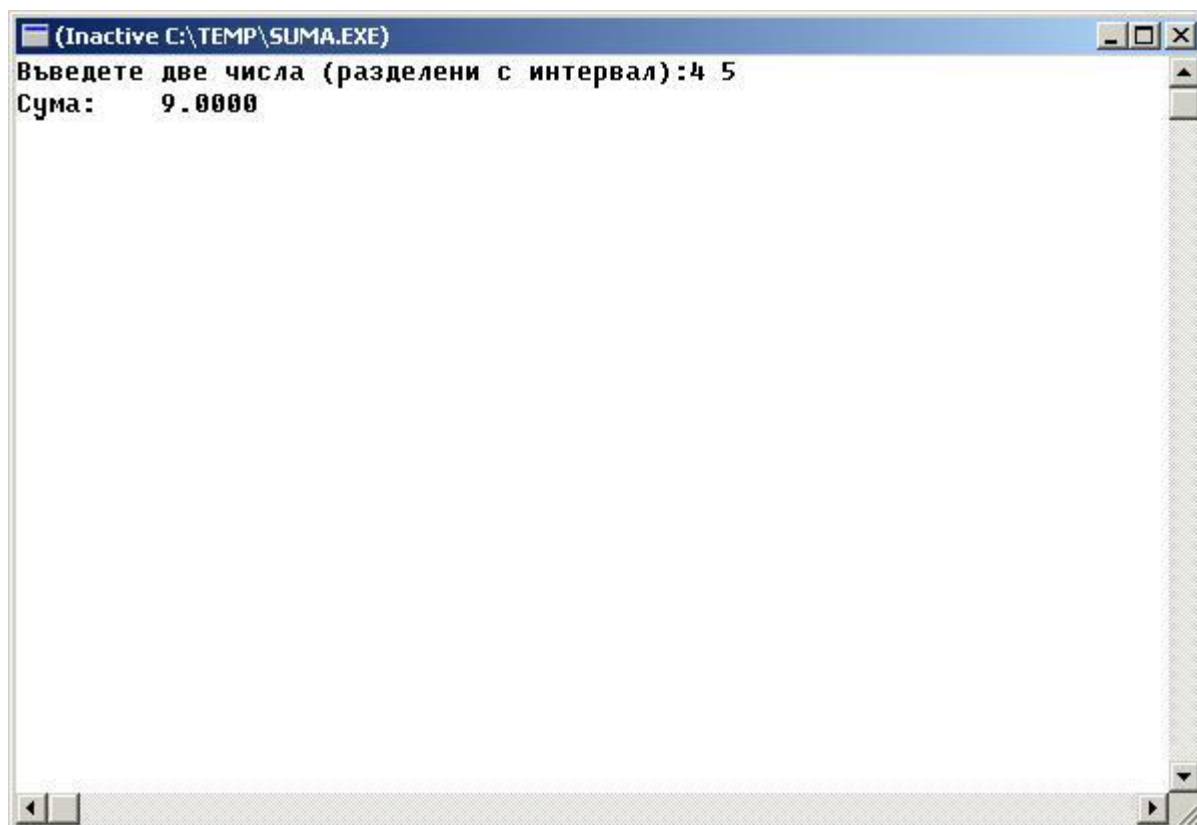
Команди на опция Run:

Run (Ctrl-F9) - стартиране на заредената в Borland Pascal програма. Ако тя не е била компилирана или е била променена, първо се компилира и след това изпълнява.

Изпълнението на програмата става в отделен прозорец. Тук е показана готова за компилиране и изпълнение програма за сума на две числа.



А тук е показан прозореца, в който се изпълнява тази компилирана и свързана програма.



Parameters - задават се начални параметри на програмата, с който тя би трябвало да се стартира. Такива параметри се задават само, ако в програмата има код за тяхната обработка.

Команди на опция Compile:

Compile (Alt-F9) - превежда програмата от активния подчинен прозорец само до обектен файл. Ако при компилация се открият правописни и граматични грешки в статусния ред се дава съобщение, а реда, в който е грешката, се маркира. Ако няма грешки се показва диалогова кутия, в която пише: Successfully compiled.

Make (F9) и Build - изграждат цялата програма, т.е. извършват компилиране и свързване. Ако няма грешки също се показва диалогова кутия, в която пише: Successfully compiled.

Target - задава се дали да се изгради конзолно или Windows изпълнима програма.

Primary file - задава се, кой е главния файл при програма състояща се от много файлове.

Команди на опция Tools.

Turbo Debugger - стартира динамичния анализатора на програмата. имате възможност да изпълнявате програмата ред по ред, да наблюдавате как се променят стойности на променливи, да задавате контролни точки за спиране на изпълнение на програмата и др.

Чрез опцията Windows се подреждат и активират отделните подчинени прозорци заредени в Borland Pascal.

Работа с помощната система Help. Тя работи в отделен прозорец. Той е организиран като хипертекст, т. е. парчета текст, в които има връзки към други парчета текст. Основната информация се извиква с команда Contents от менюто Help. Ако маркирате дума от вашата програма и натиснете Ctrl-F1 се показва помощна информация за тази дума, ако има такава.

Лента с бутони.

Бутон 1 - Help.
Бутон 2 - Open.
Бутон 3 - Save.
Бутон 4 - Find text - търсене на текст в активния прозорец.
Бутон 5 - Find next text - търсене на следващ текст в активния прозорец.
Бутон 6 - Cut.
Бутон 7 - Copy.
Бутон 8 - Paste.
Бутон 9 - UnDo - връщане назад с едно редактиращо действие.
Бутон 10 - Compile.
Бутон 11 - Make.
Бутон 12 - Run
Бутон 13 - turbo Debugger.

За да се илюстрира работата в Borland Pascal въведете следната програма:

```
program hello;  
{ Моята първа програма }  
uses WinCrt;  
begin  
    write('Здравей свят !');  
end.
```

Тази програма извежда съобщението "Здравей свят !". **program** hello; е заглавие на програмата. { **Моята първа програма** } е коментар, който не се компилира и служи само за подпомагане разбирането на програмата от програмиста. **uses** WinCrt; служи да извика библиотеката WinCrt за работа в среда на MS Windows. Ако създавате конзолна изпълнима програма, този ред от програмата не се пише. **begin** и **end** заграждат тялото на програмата, което в случая се състои от една команда. Програмата винаги завършва с точка.

След като въведете програмата натиснете Save за да я съхраните и след това натиснете Ctrl-F9 за да я транслирате и изпълните. Ако програмата има грешки, в статусния ред ви се показва съобщение за вида на грешката, а редът в който е грешката се маркира. След като отстраните грешката натискате Save за да съхраните коригираната програма отново и след това Ctrl-F9 за да транслирате и изпълните програмата. Този процес продължава до отстраняване на всички грешки.

Увод

2. Език за програмиране Pascal. Азбука на езика, думи в езика, структура на програма на Pascal. Основни вградени числови данни в Pascal. Деклариране на константи и променливи. Коментарии в програмата. Празен оператор. Оператор за присвояване. Стандартни входни и изходни процедури в Pascal.

10. Файлове в Pascal. Файлове от определен тип. Достъп до данните във файла. Процедури и функции в Borland Pascal за работа с файлове от определен тип.

- [Файлове](#)
- [Файлове от определен тип](#)
- [Грешки при работа с файлове](#)

Когато една програма работи с големи обеми от данни, възниква необходимостта те да се съхранят на някакво запамятаващо устройство - твърд диск, дискета, лента, за да може тези данни да се ползват при следващо пускане на програмата или да се ползват от някаква друга програма.

Дефиниция 10.1

Съхранените на някакво запамятаващо устройство данни се наричат *файл*.

Според вида на данните и начина на тяхната организация има различни видове файлове. В това ръководство се разглежда, как в Borland Pascal се работи с текстови файлове и файлове с определен тип данни в тях. Според начина на достъп до отделните данни във файловете те биват: с последователен и с пряк достъп. Начинът на достъп зависи и от вида на запамятаващото устройство. Например, лентовите устройства допускат само последователен достъп. Работата с файлове зависи и от операционната система. Различните операционни системи имат различни правила за задаване на запамятаващи устройства, директории и имена на файлове. Например в MS DOS името на файла е една дума до 8 латински букви и цифри дълга, а в MS Windows 95/ 98/ NT/ Me/ 2000/ XP името на файла може да е от няколко думи с обща дължина до 256 символи. В това ръководство ще се предържаме към правилата на MS DOS.

Файл от определен тип.

Дефиниция 10.2

Файл от определен тип е крайна наредена съвкупност от еднотипни данни на външен носител.

Отделните елементи могат да бъдат от всякакъв тип с изключение от тип файл. Например елементите на един такъв файл могат да са от тип **integer**, **real**, **boolean**, **char**, низ от символи, масив, запис.

За да се работи в една програма с файл трябва да се декларира файлова променлива. Деклариране на променлива за файл от определен тип става по следния начин:

име: **file of** тип;

където име е името на файловата променлива, а тип е типа на елементите на файла.

Пример 10.1:

```
type
  masiv = array[1 .. 100] of real;
  student = record
    name: string[50];
    fn: longint;
    estim: real
  end;
var
  f: file of integer;      { файл от цели числа }
  names: file of string[50]; { файл от низове от по 50 символа }
  masf: file of masiv;     { файл от масиви от по 100 реални числа }
  studf: file of student;  { файл от записи }
```

За да се работи с един файл трябва първо да се осъществи връзка между името на файла на външния носител и файловата променлива. Това става чрез библиотечна процедура

procedure assign(f: **file of** тип; s: **string**);

където f е файловата променлива, а низът s задава името на файла на външния носител. Ако е зададено само името на файла, то се счита, че файлът е в същата директория, в която е изпълнимата програма (с разширение .exe), която се получава след компилиране на програмата на Pascal. В низа s може да се зададе име на устройство и път до директорията, в която се намира файла. При това се ползват правилата на съответната операционна система за задаване на устройство и път до директория. След като се направи връзката между файловата променлива и името, файлът трябва да се отвори. Файловете от определен тип се отварят по два различни начина: когато се създава нов файл и когато се чете и пише във вече съществуващ файл. Отварянето на нов файл става чрез библиотечната процедура

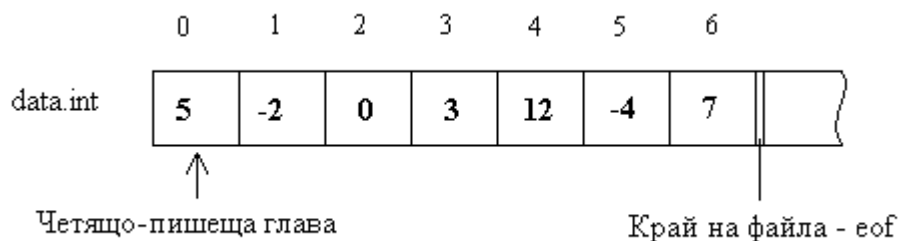
procedure rewrite(f: **file of** тип);

където f е файловата променлива. Тази процедура създава нов, празен файл с име и място указани преди това в процедурата assign. Ако вече има файл с това име, той се унищожава и се създава новия файл. В новия файл може само да се пише. Отварянето на вече съществуващ файл става чрез библиотечната процедура

procedure reset(f: **file of** тип);

където *f* е файловата променлива. Тази процедура отваря файл с име и място указани преди това в процедурата *assign*. В така отворения файл може и да се чете, и да се пише.

Във файлове от определен тип може да се четат и пишат само данни от типа на файла. Те са подредени една след друга и данната, която се чете или се пише, се сочи от четящо-пишеща глава (фиг.10.1). При отварянето на файл с *reset* тази глава сочи първия елемент на файла, който е с номер 0. При всяко четене и писане във файла главата се премества толкова елемента надясно, колкото данни са прочетени или записани.



Фиг.10.1:Файл *data.int* от 7 цели числа.

Писането във файла става с библиотечната процедура

```
write(файлова_променлива, израз1, израз2, ..., изразN)
```

където стойностите на израз1, израз2, ..., изразN са от типа на елементите на файла. След записването на стойността на съответния израз във файла четящо-пишещата глава се премества един елемент надясно, т.е. след изпълнението на процедурата *write* главата се е преместила N елемента надясно. Писането може да продължи, докато има място на външния носител.

Четенето от файл става с библиотечната процедура

```
read(файлова_променлива, променлива1, променлива2, ...,променливаN)
```

където променлива1, променлива2, ..., променливаN са от типа на елементите на файла. При прочитане на елемент от файла, сочен от четящо-пишещата глава той се зарежда в съответната променлива и главата се премества към следващия елемент надясно. Така след завършване на процедурата *read* главата се преместила N елемента надясно. Четенето ще продължи, докато се стигне до края на файла.

Когато завърши работата с файла, той трябва да се затвори. Затварянето на файл става с библиотечната процедура

```
procedure close(f: file of тип);
```

където *f* е файловата променлива.

Пример 10.2 (Виж пример 10.1):

```

var
  i, k, n: integer;
  s: string;
begin
  { Прочитане и показване на екрана числата от файл data.int. Виж фиг.10.1. }
  assign(f, 'data.int'); { data.int се намира в директорията на програмата }
  reset(f); { Отваряне на съществуващ файл }
  for i := 1 to 7 do begin
    read(f, k); { Четене на число от файла }
    writeln(k)
  end;
  close(f); { Затваряне на файла }
  { Записване във файл N имена на студенти }
  repeat
    write('Брой имена:');
    readln(n)
  until n > 0;
  assign(names, 'c:\student\students.str'); { Файлът се казва students.str и ще се създаде
на устройство c: в директория student }
  rewrite(names); { Отваряне на нов файл }
  for i := 1 to n do begin
    write('Студент ', i, ':');
    readln(s);
    write(names, s) { Записване на низ във файл }
  end;
  close(names)
end.

```

Когато се чете от един файл, при достигане края на файла трябва да се спре четенето. Проверката за край на файла става чрез библиотечната функция

function eof(**f: file of** тип):**boolean**;

където **f** е файловата променлива. Тази функция връща **true**, когато четящо-пишещата глава е достигнала маркера за край на файла (виж фиг.10.1) и **false**, когато главата сочи елемент от файла. Обикновено тази функция се поставя в условието на оператора за цикъл **while**.

Пример 10.3:

```

var
  a: array[1 .. 100] of real;
  f: file of real;
  i: integer;
begin
  assign(f, 'data.rel');
  reset(f);
  i := 0;
  while not eof(f) do begin
    i := i + 1;

```

```
        read(f, a[i])
    end;
    close(f);
end.
```

В този пример се четат реални числа от файл и се поставят в масив. Цикълът **while** ще се върти, докато главата не е стигнала края на файла.

Отваряйки файла с `reset` можем да четем и да пишем в него. И при четене, и при писане във файла четящо-пишещата глава се премества надясно. Ако трябва да се позиционира главата върху определен елемент от файла за да се прочете или да се запише нова стойност в него се използва библиотечната процедура

```
procedure seek(f: file of тип; k: integer);
```

където `f` е файловата променлива, а `k` е номера на елемент от файла, върху който се позиционира главата. Номерата на елементите във файла се номерират от 0 на нататък, т.е. първият елемент е с номер 0 (виж фиг.10.1). Точно чрез `seek` се осъществява пряк достъп до файл.

Чрез библиотечната функция

```
function filesize(f: file of тип): integer;
```

се проверява броя на елементите във файл. За файла от фиг.10.1 `filesize` ще даде 7.

Библиотечната функция

```
function filepos(f: file of тип): integer;
```

определя номера на елемента сочен от четящо-пишещата глава.

Забележка. Библиотечните подпрограми `seek`, `filesize` и `filepos` могат да се ползват само, ако файлът е отворен с `reset`.

Задача 10.1: Да се състави програма, която

- записва във файл `N` реални числа,
- променя или добавя елемент към съществуващ файл.

Анализ: Ясно е, че програмата се състои от две части. В първата част се създава нов файл и се записват в него `N` реални числа, а във втора част се отваря съществуващ файл и по указан номер на елемент и нова стойност на този елемент се позиционира главата и се записва новата стойност.

Програма

```
program rwfiles;
uses WinCrt;
var
```

```

f: file of real;
i, n: integer;
a: real;
name: string[30];
ch: char;
begin
  repeat
    writeln('1. Създаване на файл с реални числа');
    writeln('2. Промяна или добавяне на елемент към файл');
    write('Изберете:');
    readln(n)
  until (n > 0) and (n < 3);
  if n = 1
    then begin                                { I част }
      repeat
        write('Име на файл:');
        readln(name)
      until length(name) > 0;
      repeat
        write('Брой числа:');
        readln(n)
      until n > 0;
      assign(f, name);
      rewrite(f);                             { Отваряне на нов файл }
      writeln('Въведете');
      for i :=1 to n do begin
        write(i, '-то число:');
        readln(a);
        write(f, a)                          { Записване поредното число във файла }
      end;
      close(f)                               { Затваряне на файла }
    end
  else begin                                { II част }
    repeat
      write('Име на файл:');
      readln(name)
    until length(name) > 0;
    assign(f, n);
    reset(f);                                { Отваряне на съществуващ файл }
    repeat
      while not eof(f) do begin { Извеждане на елементите на файла на екрана }
        read(f, a);
        write(a:8:2)
      end;
      writeln;
    repeat
      write('Номер на елемент от файла:');
      readln(n)
    until (n >= 0) and (n <= filepos(f)); { Ако n = filepos(f)}, се добавя ново число
    към края на файла }
  end

```

```

        seek(f, n);    { Позициониране на главата върху избрания елемент }
        write('Нова стойност:');
        readln(a);
        write(f, a);  { Записване на новото число във файла }
        seek(f, 0);   { Позициониране на главата върху първия елемент }
        writeln('Още промени(Y/N):');
        readln(ch)
    until (ch = 'N') or (ch = 'n');
    close(f)           { Затваряне на файла }
end
end.

```

Грешки при работа с файлове. При работа с файлове са възможни най-различни грешки. Например:

- при създаване на нов файл дискът или директорията, в която той се създава, е забранена за писане;
- при отваряне на вече съществуващ файл няма файл с указаното име;
- при запис във файл дискът се препълва;
- при четене и писане във файл има повредена част от диска;
- изваждане на дискета по време на работа с неин файл;
- четене след края на файла и др.

В общия случай всяка от тези грешки ще прекъсне програмата и ще се появи съобщение за грешка по време на изпълнение (Runtime - error), което е твърде неприятно, защото може да доведе до загуба на информация или работно време.

Това може да се избегне чрез инструкцията (директивата) `{I-}` към компилатора. Чрез тази инструкция се казва на компилатора така да превежда вашата програма, че от мястото на инструкцията нататък в програмата, ако се срещне входно-изходна грешка програмата да не се прекъсва. Това означава, че програмата ще продължи да работи дори при възникване на входно-изходна грешка. Тя обаче ще работи грешно, ако тази грешка не се обработи програмно. Дали е възникнала входно-изходна грешка се установява чрез библиотечната функция

function ioresult: integer;

Тази функция връща 0, ако предходната входно-изходна операция е протекла успешно. В противен случай тя връща някакво число, съответстващо на възникналата входно-изходна грешка. Значението на тези числа е описано в документацията на средата за програмиране, която се получава при нейното закупуване.

Чрез обратната инструкция `{I+}` програмата се връща в режим на прекъсване при възникване на входно-изходна грешка.

Задача 10.2: Да се състави информационна система за служещи във фирма. За всеки служещ да има следната информация: име, длъжност, пол, дата на постъпване, трудов стаж, заплата. Архивът на системата да е записан във файл. Системата да извършва следните дейности:

- да прави пълна справка за всички служещи;
- да добавя нов служащ;
- да извежда имената на служещи от дадена специалност;
- да променя заплатите на всички служещи с определен процент.

Програма

```

program employee;
uses WinCrt;
type
    sluzesht = record
        ime: string[50];
        dlaznost: string[20];
        data: string[10];
        staz: integer;
        zaplata: real
    end;
var
    f: file of sluzesht;
    s: sluzesht;
    k: integer;
    proc: real;
    find: boolean;
    dlaznost: string[20];
begin
    assign(f, 'employee.dat');
    repeat
        writeln('1. Пълна справка за всички служещи ');
        writeln('2. Добавяне нов служащ ');
        writeln('3. Имената на служещи от дадена специалност');
        writeln('4. Промения заплатите на всички служещи с определен процент ');
        writeln('0. Изход');
        write('Изберете:');
        readln(k);
        case k of
            0: ;                { Изход }
            1: begin           { Пълен списък }
                {$I-} reset(f); {$I+}    { Отваряне на съществуващ файл }
                if ioresult <> 0        { Грешка при отварянето }
                then writeln('Няма служители !')
                else begin           { Файлът се е отворил }
                    find := false;
                    while not eof(f) do begin
                        read(f, s);
                        writeln('Име: ', s.ime);
                        writeln('Длъжност: ', s.dlaznost);
                        writeln('Дата на постъпване: ', s.data);
                        writeln('Стаж: ', s.staz);
                        writeln('Заплата: ', s.zaplata:8:2, ' лв. ');
                        writeln;

```

```

        find := true
    end;
    if not find
        then writeln('Няма служители !');
        close(f)
    end
end;
2: begin      { Добавяне към файла на нов служител }
    {$I-} reset(f); {$I+}      { Отваряне на съществуващ файл }
    if ioresult <> 0            { Грешка при отварянето }
        then rewrite(f)        { Създаване на нов файл }
        else seek(f, filesize(f); { Позициониране главата в края на файла }
    write('Име:'); readln(s.ime); { Запълване полетата на записа s }
    write('Длъжност:'); readln(s.dlaznost);
    write('Дата на постъпване:'); readln(s.data);
    write('Стаж:'); readln(s.staz);
    write('Заплата:'); readln(s.zaplata);
    write(f, s);                { Записване на записа s във файла }
    close(f)
end;
3: begin      { Служители с определена длъжност }
    {$I-} reset(f); {$I+}      { Отваряне на съществуващ файл }
    if ioresult <> 0            { Грешка при отварянето }
        then writeln('Няма служители !')
        else begin            { Файлът се е отворил }
            find := false;
            write('Въведете длъжност:');
            readln(dlaznost);
            while not eof(f) do begin
                read(f, s);
                if dlaznost = s.dlaznost then begin
                    writeln('Име: ', s.ime);
                    writeln('Дата на постъпване: ', s.data);
                    writeln('Стаж: ', s.staz);
                    writeln('Заплата: ', s.zaplata:8:2, 'лв. ');
                    writeln;
                    find := true
                end
            end;
            if not find
                then writeln('Няма служители с тази длъжност !');
                close(f)
            end
        end;
end;
4: begin      { Промяна на заплатата }
    {$I-} reset(f); {$I+}      { Отваряне на съществуващ файл }
    if ioresult <> 0            { Грешка при отварянето }
        then writeln('Няма служители !')
        else begin            { Файлът се е отворил }
            find := false;

```

```

write('Въведете процент на промяна на заплатата:');
readln(proc);
while not eof(f) do begin
    read(f, s);           { Прочитане на запис }
    s.zaplata := s.zaplata * (1 + proc / 100);
    seek(f, filepos(f) - 1); { Връщане главата 1 позиция назад }
    write(f, s);           { Записване на коригирания запис }
    find := true
end;
if not find
then writeln('Няма служители !');
close(f)
end
end;
else writeln('Грешен избор!')
end
until k = 0;
end.

```

Съдържание

9. Записи. Деклариране и използване на записи. Запис - поле на друг запис. Вариантни записи. Оператор with.

11. Текстови файлове. Процедури и функции в Borland Pascal за работа с текстови файлове. Изтриване и преименуване на файлове от програма на Borland Pascal.

11. Текстови файлове. Процедури и функции в Borland Pascal за работа с текстови файлове. Изтриване и преименуване на файлове от програма на Borland Pascal.

- [Текстови файлове](#)
- [Изтриване и преименуване на файлове от програма на Turbo Pascal](#)

Дефиниция 11.1

Текстов файл, наречен още *ASCII файл*, се състои само от символи от азбуката на компютъра, разпределени по редове. В тях няма други управляващи символи освен символите за нов ред, начало на ред и табулация (виж [точка 8](#)).

Такива файлове са много удобни за съхраняване на големи обеми от данни, защото могат да се четат и да се редактират от всякакви текстови редактори, като Edit, Pe2 за

MS DOS, Notepad, WordPad, Write, Word за Windows и др. Много програмни системи генерират или четат данни от текстови файлове (напр. MS Excel, Statgraph, Statistica, Mathematica и др.). Следователно данни, въведени и редактирани от удобни за работа текстови редактори или генерирани от различни програмни системи, ако са съхранени в текстови файлове, могат да се прочетат и обработят от програма на езика Pascal и обратно, изходни данни на програма на Pascal, ако са във текстов файл, могат да се прочетат, редактират, форматират и включат в някакви по-големи документи от повечето текстообработващи системи или могат да се заредят и допълнително обработят от най-различни програмни системи.

Друго предимство на текстовите файлове е, че те могат да съдържат разнотипна информация в текстов вид.

За текстови файлове в Pascal има специален тип **text**. За да се работи с такъв файл трябва да се дефинира файлова променлива от тип **text**

име: **text**;

Подобно на файловете с еднотипни елементи от предната глава за текстовия файл първо трябва да се осъществи връзка между името на файла и файловата променлива и след това той трябва да се отвори. Накрая, след като завърши работата с файла, той трябва да се затвори. Връзката между името на файла и файловата променлива става с библиотечната процедура

procedure assign(f: text; s:string);

където *f* е файловата променлива, а в низа *s* е името на файла върху външния носител (виж [точка 10](#)).

Отварянето на файла става по три различни начина според това, какво ще се прави с него.

Създаване на нов файл става с библиотечната процедура

procedure rewrite(f: text);

където *f* е файловата променлива. Създава се нов празен файл с указаното в *assign* име на външния носител. Ако вече съществува файл с това име, той се унищожава. В така отворения файл може само да се пише.

Отваряне на съществуващ файл само за четене става с библиотечната процедура

procedure reset(f: text);

където *f* е файловата променлива. Файлът с указаното в *assign* име се отваря, четящата глава се позиционира върху първия символ от файла. От така отворения файл може само да се чете.

Отваряне на съществуващ файл за добавяне на текст към края на файла става с библиотечната процедура

procedure append(f: text);

където f е файловата променлива. Файлът с указаното в assign име се отваря, четящата глава се позиционира след последния символ. Към края на така отворения файл може само да се записва допълнителен текст. От тук става ясно, че при текстови файлове нямаме пряк достъп и че не може да се промени съдържанието от вътрешността на вече съществуващ текстов файл, а най-много само да се добави текст към неговия край.

Пример 11.1:

```
var
  f1, f2, f3: text;
begin
  assign(f1, 'c:\newfile.txt');
  rewrite(f1);    { Създава нов текстов файл в главната директория на диск c: }
  assign(f2, 'readfile.txt');
  reset(f2);      { Отваря файл от директорията на програмата само за четене }
  assign(f3, 'a:appfile.txt');
  append(f3);     { Отваря файл от дискета за добавяне на данни към края му }
  .....
end.
```

Писането в текстов файл става по начин аналогичен с този при писане върху екрана (виж [точка 2](#)) с библиотечните процедури

```
write(f, израз1, израз2, ..., изразN)
writeln(f, израз1, израз2, ..., изразN)
```

където f е файловата променлива, а стойностите на израз1, израз2, ..., изразN се записват във файла като текст. Разликата между двете процедури е, че след записване на стойността на изразN при write пишещата глава остава на същия ред и чака следващо записване във файла, а при writeln тя създава нов ред и чака да пише на него. Може да се напише само

```
writeln(f)
```

тогава само се създава нов ред във файла. Към изразите, чието стойности се записват във файла може да има форматиращи команди, указващи в колко позиции ще се запише стойността. Ако израза е от тип **integer**, то форматираща команда има вида

```
израз:брой_позиции
```

Ако числото не се побира в брой_позиции, те се допълват, ако ли пък е с по-малко разряди от брой_позиции, то се допълва с интервали отляво.

Ако израза е от тип **real**, то форматираща команда има вида

```
израз:брой_позиции:точност
```

където точност задава брой числа след десетичната точка и точност < брой_позиции - 2..

Ако израза е от тип **string** или **char** то форматираща команда има вида

израз:брой_позиции

Ако низът е по-дълъг от брой_позиции, той се отрязва, ако ли пък е по-къс, той се допълва с интервали от бяло. брой_позиции и точност са изрази със стойности цели положителни числа.

Ако няма форматиращи команди или интервали между записваните стойности, те ще се залепят една за друга.

Четене от текстов файл става чрез библиотечните процедури

```
read(f, променлива1, променлива2, ..., променливаN)
readln(f, променлива1, променлива2, ..., променливаN)
```

където f е фазовата променлива, а в променлива1, променлива2, ..., променливаN ще се заредят стойностите прочетени от файла. Четенето започва от символа сочен от четящата глава. Отделните стойности в текста трябва да са разделени една от друга с интервал, табулация или нов ред. Стойностите, ако е възможно, се преобразуват от текстов вид към типа на съответните променливи, в които те се зареждат. Ако има несъответствие между типа на данната в текста, от който се чете и типа променливата, в която тази данна трябва да се зареди, възниква входно-изходна грешка. Т.е главата трябва да сочи правилно изписано цяло число за да може то да се прочете и зареди в променлива от тип **integer** или **real**. Разликата между read и readln е, че след завършване четенето от файла с read четящата глава остава на същия ред, а след завършване на readln главата преминава на нов ред. Възможно е да се зададе само

```
readln(f)
```

и тогава главата ще премине на нов ред в файла.

Възниква въпроса, ако се чете информация от един ред от текста с read, как да се разбере, че главата е стигнала края на реда за да се премине на нов ред с readln. За тази цел трябва да се ползва библиотечната функция

```
function eoln(f: text): boolean;
```

Тази функция дава **true**, ако четящата глава е достигнала края на ред и **false**, ако сочи символ вътре в реда.

Проверката, дали четящата глава е стигнала края на файла, става с библиотечната функция

```
function eof(f: text): boolean;
```

Тази функция дава **true**, ако четящата глава е достигнала края на файла и **false**, ако сочи символ вътре във файла. .

Накрая след завързване работата с даден файл той се затваря. Това става с библиотечната процедура

```
procedure close(f: text);
```

При работа с текстови файлове могат да възникнат редица грешки по време на изпълнение на програмата (run time error). Обикновено това ще предизвика прекъсване на програмата. Как да предпазим програмата от такова прекъсване бе показано в [точка 10](#) - чрез инструкция към компилатора {SI-}, не позволяваща прекъсване на програмата при входно-изходна грешка и чрез функцията ioread, проверяваща за входно изходни грешки.

Задача 11.1: Във текстов файл има записани няколко реда и колони реални числа, например

3 5.4 -7 0

-2.1 4 0.5 7

1 -0.3 3 4.3

Да се състави програма, която да прочете тези числа в двумерен масив и да намери средните аритметични от колоните в масива. Резултатите да се добавят към края на файла.

Анализ: Първо, ще отворим файла за четене, ще прочетем числата в масива, като през това време ще преброим редовете и колоните. След това в друг едномерен масив ще намерим средните аритметични. Накрая ще отворим файла за добавяне и ще добавим тези средно аритметични.

Програма

```
program textfile;
uses WinCrt;
const max = 20;
var
  f: text;
  a: array[1 .. max, 1 .. max] of real;
  b: array[1 .. max] of real;
  n, m, i, j: integer;
  name: string[50];
begin
  write('Въведете името на файла:');
  readln(name);
  assign(f, name);    { Връзка между файловата променлива и името на файла }
  reset(f);           { Отваряне на файла за четене }
  n := 0;
```

```

while not eof(f) do begin      { Цикъл до достигане края на файла }
    m := 0;
    n := n + 1;
    if n > max then exit;      { Прекъсване на цикъла }
    while not eoln(f) do begin { Цикъл до достигане края на ред }
        m := m + 1;
        if m > max then exit;
        read(f, a[n, m])      { Четене на поредно число от файла }
    end;
    readln(f)                  { Предвижване четящата глава на нов ред }
end;
close(f);                     { Затваряне на файла }
if (n > max) or (m > max) or (n = 0) or (m = 0)
    then writeln('Твърде много редове или стълбове или грешна структура на файла
!')
    else begin
        for i := 1 to n do begin
            b[i] := 0;
            for j := 1 to m do
                b[i] := b[i] + a[i, j];
            b[i] := b[i] / n
        end;
        append(f);             { Отваряне на файла за добавяне към края му }
        writeln(f);            { Нов ред във файла }
        for i := 1 to m do
            write(f, a[i, j]:8:2); { Записване на число във файла }
        close(f);              { Затваряне на файла }
    end
end.

```

Понякога възниква необходимостта от програма да се преименува или изтрие файл. За тази цел в Borland Pascal има две процедури. Процедурата за преименуване е

```
procedure rename(файлова_променлива; s: string);
```

където низът s задава новото име на файла. Старото име трябва преди това да е свързано с файловата променлива чрез assign.

Процедурата за изтриване на файл е

```
procedure erase(файлова_променлива);
```

Файлът, чието име преди това е било свързано чрез assign към файловата променлива, се изтрива.

Задача 11.2: Да се състави програма, която от файл с реални числа премахва зададено число.

Анализ: Ще копираме във временен файл всички числа от файла, различни от зададеното число. След това ще изтрием файла и временният файл ще преименуваме с името на изтрения файл.

Програма

```
program delfromfile;
uses WinCrt;
var
    f, f1: file of real;
    a, b: real;
    find: boolean;
    name: string[50];
begin
    write('Въведете името на файла:');
    readln(name);
    write('Задайте число за премахване от файла:');
    readln(b);
    assign(f, name);           { Връзка между файловата променлива и името на файла }
    reset(f);                  { Отваряне на файла за четене }
    assign(f1, 'temp.dat');
    rewrite(f1);               { Отваряне на нов временен файл }
    find := false;
    while not eof(f) do begin
        read(f, a);            { Четене на число от файла }
        if a = b
            then find := true
            else write(f2, a)  { Записване на различното от b число във временния файл }
    end;
    close(f);
    close(f1);
    if find
        then begin            { Във файла са открити числа равни на b }
            erase(f);          { Изтриваме файла }
            rename(f1, name)   { Преименуване на временния файл с името на изтрения }
        end
        else begin
            writeln('Няма такова число във файла !');
            erase(f2)           { Изтриваме временния файл }
        end
    end.
```

Съдържание

10. Файлове в Pascal. Файлове от определен тип. Достъп до данните във файла. Процедури и функции в Borland Pascal за работа с файлове от определен тип.

2.Език за програмиране Pascal. Азбука на езика, думи в езика, структура на програма на Pascal. Основни вградени числови данни в Pascal. Деклариране на константи и променливи. Коментарии в програмата. Празен оператор. Оператор за присвояване. Стандартни входни и изходни процедури в Pascal.

- [Азбука на езика, думи в езика, структура на програма на Pascal](#)
- [Деклариране на константи и променливи](#)
- [Коментарии в програмата](#)
- [Празен оператор](#)
- [Оператор за присвояване](#)
- [Стандартни входни и изходни процедури в Pascal](#)

Както всеки език и Pascal има азбука и препинателни знаци, от тях се изграждат думи и от думите според граматика - по сложни фрази (в случая програми). Азбуката е следната:

- латински букви: a, b, ... , z, _ - в Pascal не се прави разлика между големи и малки букви;
- цифри: 0, 1, ... , 9;
- знаци за операции: + - * / > >= < <= <> =
- специални знаци: := -> ^
- разделители и скоби: () [] {} ; , . ' "

Вижда се, че някои от буквите в Pascal се пишат с две букви на компютъра (например: >= <= := или->). Такива букви не бива да се разделят една от друга с интервал, нов ред или табулация

От азбуката на езика се изграждат думи. В Pascal има три вида думи:

- ключови (резервирани) думи. Тези думи са двадесетина, трябва точно да се изписват и да се ползват само там, където налага граматиката на езика. Пълен списък от ключовите думи на езика има в Help на Borland Pascal. Такива думи са: **program, uses, label, const, type, var, integer, real, procedure, function, begin, end, if, for, do, goto** и др. В Borland Pascal те се показват обикновено като тъмно черни.
- идентификатори. Те служат да именуват различни неща в програмата, като например променливи, подпрограми, типове, записи и др. Те се измислят от програмиста. Правилото за създаване на идентификатор е следното: идентификаторът трябва да е различен от ключовите думи, трябва да започва с латинска буква и трябва да се състои само от латински букви и цифри. Например a, ab, a12, first_name. В една и съща програмна част не бива да има дублиране на идентификатори (не забравяйте, че не се прави разлика между малки и големи букви). Borland Pascal те се показват обикновено в нормален черен цвят.
- константи-литерали. Това са константите в програмата, които присъстват със своята стойност. Те в Borland Pascal обикновено се оцветяват в синьо. Например: 5, -12.5, 'Здравей свят !' и т. н. Те биват:
- числови

цели: 5, -24, +15

реални: 5.5, -4.98, +3.45, . 45, 0.123e-5

- символни - включват всичките 256 символа на компютъра: 'a', 'A', 'Ж', '+'.
• низ от символи - поредица от всичките 256 символа на компютъра заградени с апострофи: 'Въведете число:', 'Eggo'
- логически: **true** - истина, **false** - лъжа.

Символите и низовете от символи се заграждат със знака апостроф '. Внимавайте, кой точно апостроф се ползва, защото на клавиатурата има два подобни знака. Ползва се този при кавичките.

Отделните думи се разделят помежду си със знаците разделители, знаци за операции, специални знаци, интервали, табулация или нов ред. Няма значение как разпределяте програмата по редове.

Основната структура на една програма е следната:

заглавие
декларации
тяло .

Програмата винаги завършва с точка.

Заглавието има следния вид:

program име;

Името задава име на цялата програма и е идентификатор.

Декларациите декларират използваните в програмата библиотеки, етикети, константи, типове, променливи и подпрограми. Основен принцип в Pascal е, че всичко се декларира.

Тялото на програмата съдържа съвкупността от нейните команди и има следния вид:

begin
 команда 1;
 команда 2;
 .
 .
 команда N
end

Знакът ";" служи да раздели две поредни команди една от друга и не е към командите. Затова след последната команда може да няма ";", защото **end** не е команда.

Командите в езика Pascal са или оператори или обръщения към подпрограми процедури.

Освен константи-литерали (т. е. такива, чието стойност е някъде вътре в тялото на програмата), програмистът може да декларира константи в началото на програмата като зададе име и стойност на константата. Всички такива константи в програмата се обявяват в декларативната част след ключовата дума **const**. Декларацията на константа има следния вид:

```
име = стойност;
```

Името е идентификатор, а стойността е константа-литерал или израз от преди това декларирани константи и константи-литерали. Пример:

```
const  
count = 5;  
lihva = count * 0.05;  
message = 'Грешен брой!';
```

Променливите в програмата са клетки от оперативната памет, в които можете да съхранявате и променяте данни по време на изпълнение на програмата. Променливата има име (идентификатор) и тип (ключова дума или идентификатор). Типът служи да укаже какви данни могат да се присвояват на променливата. От него се разбира дължината в байтове на променливата и нейната дефиниционна област (т. е. диапазона от стойности, които тя може да приема). Всички променливи в програмата се обявяват в декларативната част след декларацията на константите, ако има такава. Декларацията на променливите започва с ключовата дума **var**. Променлива се декларира по следния начин:

```
име: тип;
```

Ако имате няколко променливи от един и същи тип можете за икономия на писане да ги наредите в списък:

```
име1, име2, ..., имеN: тип;
```

Например:

```
var  
a: integer;  
p: real;  
k, n, m: integer;  
x1, x3: real;  
ch: char;
```

Типът на променлива може да е дефиниран от програмиста или да е един от вградените в езика типове.

Основните вградени в Pascal типове са: **integer**, **real**, **char**, **boolean**, **string**, **text**. Освен тях специално в Borland Pascal има още няколко. В Help на Borland Pascal има подробна информация за вградените типове, дължината на полето от паметта, където те се помещават и дефиниционните им области. Нека за сега разгледаме най-използваните скаларни типове.

integer - целочислен тип данни. Реализира се в 2 байта и променливите от този тип могат да съдържат цели числа приблизително от -32000 до 32000.

real - реални 7-8 цифрени числа. Реализира се в 6 байта.

char - единични символи. Реализира се в 1 байт. Променливите от тип **char** могат да получават като стойност всеки един от 256-те символи на компютъра.

boolean - логически тип. Променливите от този тип могат да получават стойности **true** (истина) или **false** (лъжа).

Вътре на произволни места в текста на програмата може да се вложи свободен текст наречен коментари, които да дават пояснение за програмиста, който чете програмата. Коментарът се прескача от транслятора и не влиза в кода на изпълнимата програма. Коментарът се загражда с {}. Например:

{ това е коментар }

Както бе казано по-горе, тялото на програмата се състои от поредица от команди, които са или обръщения към процедури, или оператори. Най-простия оператор е празния оператор. Той нищо не върши и се маркира само с ";". Понякога се използва да маркира не готови още части от програмата, празни цикли, празен избор в оператор за избор.

Операторът за присвояване служи да зададе стойност на променлива. Той има следния вид:

променлива := израз

Действието му е следното: изчислява се израза и получената стойност се присвоява на променливата. Трябва при това да се внимава стойността на израза да е от същия тип като този на променливата и тази стойност да е в дефиниционната област на променливата, защото иначе ще настъпи препълване, което ще доведе до грешни резултати при изпълнение на програмата. Какво представляват изразите се описва в следващата точка.

Чрез готови библиотечни подпрограми се осъществява входа и изхода от и към програма на Pascal.

Процедура `readln` служи за въвеждане на данни от клавиатурата. В този случай тя има вида:

`readln(променлива1, променлива2, ..., променливаN)`

Нейното действие е следното: програмата спира изпълнението си и чака да се въведат от клавиатурата N-те стойности разделени с интервал. Натиска се клавиша Enter и стойностите се присвояват на съответните променливи и програмата продължава да се изпълнява нататък. Необходимо е тези стойности да са съобразени с типа на променливите, иначе процедурата дава грешка и програмата се прекъсва. Ако стойностите са извън дефиниционната област на съответните променливи, възниква препълване и програмата не работи вярно. Пример:

```
a: integer; x: real;  
readln(a, x);
```

Вход: 3 5.34

Резултат: a=3, x=5.34

Процедурата readln може да се ползва и без параметри. Тогава тя само спира изпълнението на програмата и се чака да се натисне клавиша Enter за да се продължи.

В Pascal съществува и процедура read за вход от клавиатурата, но в Borland Pascal тя е реализирана твърде странно и не се препоръчва да се използва.

Чрез процедурите write и writeln си осъществява извеждане на данни към екрана. В този случай те имат вида:

```
write(израз1, израз2, ..., изразN)
```

```
writeln(израз1, израз2, ..., изразN)
```

Тези процедури извеждат стойностите на изразите една след друга на екрана. Разликата между двете е, че при write курсора остава на същия ред на екрана след последната изведена стойност, а при writeln след извеждане на стойностите курсорът преминава в началото на нов ред. writeln може да се използва и без параметри. Тогава тя просто премества курсора в началото на нов ред. И при двете процедури програмистът трябва да се погрижи да отдели стойностите на изразите една от друга, иначе те ще се слепят. Това може да се постигне или като между тях се изведе по един символ интервал ' ', или ако към изразите се добави форматираща информация. Освен това без форматираща информация реалните стойности се извеждат в трудния за четене научен формат (например 1.5 ще се изведе като 0.150000000E01). Форматиращата информация има вида

```
израз:n
```

```
реален_израз:n:m
```

където n задава броя на позициите, в които ще се изведе стойността на израза, m задава броя на цифрите след десетичната точка (точността на числото). Ако стойността е по-дълга от n, тя се извежда цялата, ако е по-къса тя се допълва до n с интервали отляво.

Задача 2.1: Да се състави програма, която събира две числа.

Анализ

Входни данни: a и b - реални

Изходни данни: c - реална

Знание: c=a+b

Алгоритъм

Начало

1. Въведи a, b

```
2.c <- a + b
3.Изведи c
Край
```

Програма

```
program suma;      { заглавие на програмата }
uses WinCrt;
var                { декларации на променливи }
  a, b, c: integer;
begin              { тяло на програмата }
  readln(a, b);    { команда1 - въвеждане от клавиатурата }
  c := a + b;      { команда2 - оператор за присвояване }
  writeln(c);      { команда3 - извеждане на екрана }
end.
```

След като изпълните тази програма се вижда, че така написана тази програма е неудобна за ползване. Затова след **begin** добавете `write('Въведете две събираеми:');` и вместо `writeln(c);` напишете `writeln('Сума:',s:10:4);`

Задача 2.2: Да се състави програма, която по зададен радиус изчислява дължината и лицето на кръга.

Анализ

Входни данни: r - реални, константа $\pi=3.1416$

Изходни данни: l и s - реални

Знание: $l=2 \pi r$, $s=\pi r^2$

Алгоритъм

```
Начало
1.Въведи r
2.l <- 2  $\pi$  r
3.s <-  $\pi r^2$ 
4. Изведи l
5.Изведи s
Край
```

Програма

```
program circle;
const
  p = 3.1416;
var
  r, l, s : real;
begin
  write('Въведете радиус на кръга:');
  readln(r);
  l:=2 * p * r;
  s:=p * r * r;
```

```
writeln('Дължина:', l:5:2);  
writeln('Лице:', s:5:2);  
end.
```

Съдържание

1. Среда за програмиране Borland Pascal v.7.0. Въвеждане и редактиране на програма. Компилиране. Използване на Help.

3. Изрази: аритметични и логически изрази, видове операции и приоритет за тяхното изпълняване.

3. Изрази: аритметични и логически изрази, видове операции и приоритет за тяхното изпълняване.

Изразите биват:

- аритметични (цели или реални) - стойността им е число;
- логически - стойността им е **true** или **false**;
- низ от символи;
- други.

В тази точка се разглеждат само първите два вида.

Изразът може да бъде:

- константа;
- променлива;
- константи, променливи и обръщения към подпрограми-функции свързани със знаци за операции и скоби ().

Операциите имат приоритет и равно приоритетните операции се изпълняват отляво надясно. Приоритетът се нарушава чрез скобите. Използват се само кръгли скоби () независимо от тяхното влагане. Според броя на операндите си операциите биват:

- унарни - с един операнд: $-a$, **not** b ;
- бинарни - с два операнда: $a + b$, $b < a$.

Аритметични изрази. В целочислените изрази участват само целочислени константи, променливи, функции и операции. В реалните изрази могат да участват както цели, така и реални константи, променливи и функции, но само реални операции. Аритметичните операции са (отгоре надолу според приоритета):

- (отрицателна стойност) - унарна: $-a$, $-(a + b)$
- + (положителна стойност) - унарна: $+a$, $+(a - b)$
- * (умножение), / (реално деление), **div** (целочислено деление), **mod** (модул)
- + (събиране), - (изваждане).

Целочислената деление е деление без остатък

$$1 \text{ div } 2 = 0, 7 \text{ div } 3 = 2.$$

Модулът е остатъка от целочисленото деление

$$6 \text{ mod } 3 = 0, 7 \text{ mod } 3 = 1, 8 \text{ mod } 3 = 2.$$

Освен тези операции в Pascal съществуват и побитови операции, които не се разглеждат в това ръководство.

Логически изрази. В тези изрази участват операции за сравнение и логически операции.

Операциите за сравнение са 6: $>$, $>=$, $<$, $<=$, $=$, $<>$ или $><$. Те са бинарни, като от двете страни на операцията стоят еднотипни изрази. Резултатът от тях е **true** или **false** според това, дали сравнението се удовлетворява или не. Те са равноприоритетни и са с по-малък приоритет от аритметичните операции. Примери:

$$a < 0, k > n \text{ div } 4, s / n >= 3.5 + x, ch = 'ж', name <> 'Иван Петров'.$$

Към тези операции може да се добави и операцията **in**, която проверява, дали един елемент принадлежи към множество.

Логическите операции са 4 и са следните:

- not** - логическо не (отрицание), унарна;
- and** - логическо и (конюнкция), бинарна;
- or** - логическо или (дизюнкция), бинарна;
- xor** - логическо или, или (изключващо или, сума по модул 2), бинарна.

Те действат върху логически изрази и резултатът от тях е **true** или **false**. **not** е с по-висок приоритет от останалите три, които са равно приоритетни. В стандартният Pascal логическите операции са с по-нисък приоритет от операциите за сравнение, обаче в Borland Pascal те са с по-висок приоритет и поради това сравненията трябва да се заграждат в скоби. Например:

$$\text{not } (a < 0), (a = 0) \text{ and } (b = 0).$$

Нека a, b: **boolean**; Действието на логическите операции у следното:

a	not a
---	-------

true	false
false	true

b	true	false
a	a and b	
true	true	false
false	false	false

b	true	false
a	a or b	
true	true	true
false	true	false

b	true	false
a	a xor b	
true	false	true
false	true	false

Някои свойства на логическите операции:

a and b = b and a, a or b = b or a, a xor b = b xor a

a or (b and c) = (a or b) and (a or c), a and (b or c) = (a and b) or (a and c)

not not a = a

true or a = true

false and a = false

Закони на Деморган:

not (a and b) = not a or not b,

not (a or b) = not a and not b

Задача 3.1: Дадена е точка P с координати (x_p, y_p) и права L с уравнение $Ax + By + C = 0$. Да се състави програма, която изчислява разстоянието от точката до правата.

Анализ

Входни данни: x_p, y_p, A, B, C - реални.

Изходни данни: s - реална.

Знание:

$$s = \frac{Ax_p + By_p + C}{\sqrt{A^2 + B^2}}$$

Алгоритъм

Начало

1. Въведи x_p, y_p

2. Въведи A, B, C

3.

$$s \leftarrow - \frac{Ax_p + By_p + C}{\sqrt{A^2 + B^2}}$$

4. Изведи s

Край.

Програма

```
program distance;
uses WinCrt;
var
  a, b, c, xp, yp, s: real;
begin
  write('Въведете координати на точката:');
  readln(xp, yp);
  write('Въведете коефициенти на правата:');
  readln(a, b, c);
  s := (a * xp + b * yp + c) / sqrt(a * a + b * b);
  writeln('Разстояние:', s:6:2);
end.
```

Упътване: sqrt(x) е библиотечна функция в Pascal за квадратен корен.

Задача 3.2. Даден е интервал от две числа (a,b) и трето число x. Да се състави програма, която проверява дали числото е в интервала.

Анализ

Входни данни: a, b, x - реални.

Исходни данни: yes - логическа.

Знание Числото е в интервала, ако $(a < x) \text{ and } (x < b) = \text{true}$.

Алгоритъм

Начало

1. Въведи a, b
2. Въведи x
3. yes $\leftarrow (a < x) \text{ and } (x < b)$
4. Изведи yes

Край.

Програма

```
program interval;  
uses WinCrt;  
var  
    a, b, x: real;  
    yes: boolean;  
begin  
    write('Въведете интервал:');  
    readln(a, b);  
    write('Въведете число:');  
    yes := (a < x) and (x < b);  
    writeln('Числото е в интервала ?', yes);  
end.
```

Забележка. Тази програма не проверява дали $a < b$.

Задача 3.3. Дадена е окръжност с координати на центъра (x_c, y_c) и дължина на радиуса r , дадена е точка с координати (x, y) . Да се състави програма, която проверява дали точката е вътре в окръжността.

Анализ

Входни данни: x_c, y_c, r, x, y - реални.

Исходни данни: yes - логическа

Знание: Точката е в окръжността ако $(x_c - x)^2 + (y_c - y)^2 < r^2 = \text{true}$.

Алгоритъм

Начало

1. Въведи x_c, y_c
2. Въведи r
3. Въведи x, y
4. yes $\leftarrow (x_c - x)^2 + (y_c - y)^2 < r^2$
5. Изведи yes

Край.

Програма

```

program p_in_circle;
uses WinCrt;
var
    xc, yc, r, x, c: real;
    yes: boolean;
begin
    write('Въведете координати на центъра:');
    readln(xc, yc);
    write('Въведете дължина на радиуса:');
    readln(r);
    write('Въведете координати на точка:');
    readln(x, y);
    yes :=  $\text{sqr}(xc - x) + \text{sqr}(yc - y) < r * r$ ;
    writeln('Точката е вътре в окръжността?', yes);
end.

```

Упътване: $\text{sqr}(x)$ е библиотечна функция за x^2 .

Съдържание

2. Език за програмиране Pascal. Азбука на езика, думи в езика, структура на програма на Pascal. Основни вградени числови данни в Pascal. Деклариране на константи и променливи. Коментarii в програмата. Празен оператор. Оператор за присвояване. Стандартни входни и изходни процедури в Pascal.

4. Оператори за логическо управление на програмата. Логически оператор. Съставен оператор (оператор-блок). Оператор за безусловен преход. Оператор за избор.

4. Оператори за логическо управление на програмата. Логически оператор. Съставен оператор (оператор-блок). Оператор за безусловен преход. Оператор за избор.

- Логически оператор
- Съставен оператор (оператор-блок)
- Оператор за безусловен преход
- Оператор за избор

Алгоритмите, чието програми писахме до сега, се наричат линейни. Те се състоят от наредени една след друга команди, които се изпълняват едни след други без никакви варианти от началото до края на алгоритъма. Чрез тези алгоритми могат да се решат обаче много тесен кръг прости задачи. Езикът Pascal притежава оператори, чрез които

може да се наруши последователността на изпълняване на командите. Това се извършва според логиката на алгоритъма.

Логически оператор. Чрез този оператор програмата може по някакво логическо условие да се разклони в един или в два допълнителни клона. Той съществува в два варианта: непълен и пълен. Непълният оператор има следната граматика:

```
if логически_израз
  then команда
```

Действието в този случай на оператора е следното: първо се изчислява логическия израз и ако той е истина (**true**), се изпълнява командата след ключовата дума **then** и се продължава нататък, ако изразът е лъжа (**false**), командата след **then** се прескача и се продължава нататък. Пример:

```
if a < 0
  then a := -a
```

В резултат на този оператор, ако *a* е отрицателно число, то *a* приема стойността си с обратен знак, ако *a* е неотрицателно число, нищо не се прави - т. е. *a* получава абсолютната си стойност.

Пълният логически оператор има следната граматика:

```
if логически_израз
  then команда1
  else команда2
```

Тук той има следното действие: първо се изчислява логическия израз и ако той е истина (**true**), се изпълнява команда1 след ключовата дума **then** и се продължава нататък, като се прескача команда2, ако изразът е лъжа (**false**), команда1 след **then** се прескача, а се изпълнява команда2 след ключовата дума **else** и се продължава нататък. Пример:

```
if a < b
  then c := a
  else c := b
```

В резултат на този оператор, ако *a* е по-малко от *b*, тогава на *c* се присвоява *a*, иначе на *c* се присвоява *b* - т. е. на *c* се присвоява минималното от *a* и *b*.

Забележете, че след команда1 не се поставя ";". Напомняме, че ";" служи да раздели две поредни команди, а не е към командите. Ако поставите ";" след команда1, компилатора ще ви издаде съобщение за грешка "Error in statment" - "Грешка в оператора".

Съставен оператор. Често в програмата се налага според някакво логическо условие да се изпълни не една, а няколко команди. В оператора **if** обаче след **then** и след **else** може да има само по една команда. В този случай се използва съставният оператор, наречен още оператор - блок. Този оператор има следната граматика:

begin

команда 1;
команда 2;
.
.
команда N

end

Той служи да групира няколко команди в една и може да бъде поставен на всяко едно място в програмата, където е допустима една отделна команда. Всъщност тялото на цялата програма е един съставен оператор. Всяка една команда вътре в съставния оператор може да е също съставен оператор или до съдържа в себе си съставни оператори. Така се получават много вложени един в друг съставни оператори, което позволява да се пишат произволно сложни програми с Pascal. Затова Pascal се определя като език с блокова структура. Пример:

```
if a <= 0
  then begin
    write('Въведете ново число:');
    readln(a);
    b := b + a
  end
else b := b + a;
```

В този случай след **then** имаме съставен оператор групиращ в един оператор 3 команди. Забележете, че след **end** не се пише ";", защото следва **else** на оператора **if**.

Задача 4.1: Дадено е линейното уравнение

$$a x + b = 0$$

Да се състави програма, която да решава това уравнение.

Анализ

Входни данни: a и b - реални.

Изход: Ако a и b са 0: Всяко число;
 Ако a = 0, a b ≠ 0: Няма решение;
 Иначе: x = -b / a

Алгоритъм

Начало

1. Въведи a, b
2. Ако a = 0 и b = 0
 Изведи Всяко число
 Край
3. Ако a = 0 и b ≠ 0
 Изведи Няма решение
 Край

4. Ако $a \neq 0$
 $x \leftarrow -b / a$
 Изведи x
Край

Програма

```
program lneq;  
uses WinCrt;  
var  
  a, b, x: real;  
begin  
  write('Въведете двата коефициента:');  
  readln(a, b);  
  if (a = 0) and (b = 0)  
  then writeln('Корен: Всяко число')  
  else if a = 0  
  then writeln('Корен: Няма')  
  else begin  
    x := -b / a;  
    writeln('Корен:', x:8:4)  
  end  
end.
```

Упътване: Във втория **if** не се проверява $b \neq 0$, защото ако $a = 0$, b трябва да е различно от 0, иначе щеше условието на първия **if** да е **true**. След **else** на втория **if** трябва да се изпълнят две команди и затова се ползва съставен оператор.

Задача 4.2: Дадени са три отсечки. Да се състави програма, намираща лицето на триъгълника образуван от тях, ако те образуват триъгълник.

Анализ

Входни данни: $a > 0$, $b > 0$, $c > 0$ - реални
Изход: Грешка: Отрицателна отсечка
 Невъзможен триъгълник
 s - лицето по формулата на Херон

Алгоритъм

Начало

1. Въведи a , b , c
2. Ако $a \leq 0$ или $b \leq 0$ или $c \leq 0$
 Изведи Грешка: Отрицателна отсечка
 Край
3. Ако $a + b \leq c$ или $a + c \leq b$ или $b + c \leq a$
 Изведи Невъзможен триъгълник
 Край

4.
$$p \leftarrow -\frac{a + b + c}{2}$$

$$5. s < -\sqrt{p(p-a)(p-b)(p-c)}$$

6. Изведи s

Край

Програма

```

program lice;
uses WinCrt;
var
    a, b, c, p, s: real;
begin
    write('Въведете дължините на три отсечки:');
    readln(a, b, c);
    if (a <= 0) or (b <= 0) or (c <= 0)
    then writeln('Грешка: Отрицателна отсечка')
    else if (a + b <= c) or (a + c <= b) or (b + c <= a)
    then writeln('Невъзможен триъгълник')
    else begin
        p := (a + b + c) / 2;
        s := sqrt(p * (p - a) * (p - b) * (p - c));
        writeln('Лице на триъгълника:', s:10:4)
    end
end.

```

Оператор за безусловен преход. Този оператор служи да наруши последователното изпълнение на командите в програмата. Обикновено командите в програмата се изпълняват една след друга. Ако по някаква причина искаме да прескочим няколко команди или да се върнем назад с няколко команди, тогава може да се използва операторът за безусловен преход. Той е нехарактерен за стила на езика Pascal и е остатък от по-старите езици за програмиране. Доказано е, че всяка програма на Pascal може да се напише и без този оператор, като се използват други средства. Граматиката на оператора за безусловен преход е следната:

goto етикет

където етикетът е цяло положително число. То се поставя пред командата от програмата, към която искаме да се извърши прехода:

етикет: команда

Всички етикети в програмата трябва да са различни и да са описани в декларативната и част. Описанието е следното:

label етикет1, етикет2, ... , етикетN;

Декларирането на етикетите става преди декларирането на константите и променливите (**const** и **var**) и след декларацията за ползвани библиотеки от готови подпрограми (**uses**). Пример:

```

program ...;
  uses .....
  label 1, 2; { Деклариране на етикетите }
  const .....
  var ....
  begin
1:   readln(a);
      if a > 0 then goto 2; { Преход напред }
      ....
      goto 1; { Преход назад - неявен цикъл }
      ...
2:   writeln(a);
      ....
  end.

```

В горната непълна програма два пъти се използва безусловен преход: напред - **goto** 2 и назад - **goto** 1. В първият случай се прескачат няколко команди, а във втория се връщаме назад няколко команди и те ще се изпълнят отново. Когато в един алгоритъм няколко команди се изпълняват многократно, се казва, че те образуват цикъл. Циклите образувани с оператора **goto** се наричат неявни цикли. В Pascal има специални оператори за цикъл.

Операторът **goto** е опасен за ползване, защото нарушава естествената логика на програмата. Чрез него тя се накъсва и става трудна за асимилиране. Това често води до грешки. Затова използването на **goto** би трябвало да се избягва. Съществуват обаче случаи, където опита да се избегне **goto** също усложнява и забавя програмата.

Задача 4.3: Да се състави програма, която събира N реални числа.

Анализ

Входни данни: $N > 0$ цяло

$a_1, a_2, \dots, a_N$ - реални.

Изход: $s = a_1 + a_2 + \dots + a_N$

Алгоритъм

Начало

1. Въведи N

2. Ако $N \leq 0$

Изведи Грешен брой

Край

3. $s \leftarrow 0$

4. $i \leftarrow 1$

5. Въведи a_i

6. $s \leftarrow s + a_i$

7. $i \leftarrow i + 1$

8. Ако $i \leq N$ Отиди към 5

9. Изведи s

Край

Програма

```
program suma1;
uses WinCrt;
label 1, 2;
var
  a, s: real;
  i, n: integer;
begin
  write('Въведете броя на числата');
  readln(n);
  if n <= 0
  then begin
    writeln('Грешен брой!');
    goto 2
  end;
  s := 0;
  i := 1;
1: write('Въведете ', i, '-то число:');
  readln(a);
  s := s + a;
  i := i + 1;
  if i <= n then goto 1;
  writeln('Сума:', s:10:4);
2: end.
```

Коментар. В тази програма има два безусловни прехода. Първият (**goto 2**) е към края на програмата и вторият (**goto 1**) е няколко команди назад към началото. В този случай имаме неявен цикъл изхода, от който е с оператора **if**.

Оператор за избор. С помощта на логическия оператор **if** и съставен оператор можем според някакво условие да разклоним алгоритъма на един или два клона. Често обаче се налага разклоняването на много клонове според някакви изброими ситуации (например избор от меню). Вместо да се правят каскади от вложени логически оператори, се използва оператор за избор. Той има следната граматика:

```
case изброим_израз of
  константа1: команда1;
  константа2: команда2;
  .
  .
  .
  константаN: командаN;
  else командаN+1
end
```

else командаN+1 е незадължително.

Дефиниция 4.1

Изброим израз е този, чието стойност може да се преброи.

Такива са целите числа и символите. Реалните числа и низовете от символи не могат да се преброят. Действието на оператора **case** е следното: Първо се изчислява изброимия израз и неговата стойност се търси измежду изброените константи. Ако има константа съвпадаща със стойността на израза, то се изпълнява командата срещу тази константа. Ако стойността не съвпада с нито една от изброените константи, се изпълнява командата след **else**, ако има **else**, иначе, ако няма, нищо не се прави. Всяка една от командите вътре в оператора **case** може да е съставен оператор. Така се получават много разклонения в алгоритъма.

Пример 4.1:

```
case (k + 4) mod 6 of  
  0: c := a + b;  
  1, 3, 5: c := a - b;  
  2: begin  
      write('Въведете число:');  
      readln(b);  
      c := a + b  
  end;  
  else writeln('Грешка')  
end;
```

От примера се вижда, че ако за няколко константи трябва да се изпълни една и съща команда, те се изброяват в списък.

Задача 4.4: Да се състави програма, която намира лицата на фигурите: окръжност по даден радиус, квадрат по дадена страна, правоъгълник по дадени две страни и триъгълник по дадени страна и височина към нея.

Анализ

Входни данни: k = 1, 2, 3 или 4 - вид на фигурата, и съответните параметри на фигура, изброени в условието. Константа $\pi = 3.1416$.

Изход: s - лице на съответната фигура.

Алгоритъм

Начало

1. Въведи k
2. Ако k = 1
 Въведи r
 $s \leftarrow \pi r^2$
3. Ако k = 2
 Въведи a
 $s \leftarrow a^2$
4. Ако k = 3
 Въведи a, b
 $s \leftarrow a b$
5. Ако k = 4
 Въведи a, h

```

    s <- a h / 2
6. Иначе
    Изведи Грешен номер на фигура
    Отиди към 1
7. Изведи s
Край

```

Програма

```

program figurs;
uses WinCrt;
label 1;
var
    a, b, h, r, s: real;
    k: integer;
begin
1:  writeln('Вид фигура');
    writeln('1. Кръг');
    writeln('2. Квадрат');
    writeln('3. Правоъгълник');
    writeln('4. Триъгълник');
    write('Изберете:');
    readln(k);
    case k of
        1: begin
            write('Въведете радиус:');
            readln(r);
            s := Pi * r * r;
        end;
        2: begin
            writeln('Въведете страна:');
            readln(a);
            s := a * a;
        end;
        3: begin
            writeln('Въведете две страни:');
            readln(a, b);
            s := a * b;
        end;
        4: begin
            writeln('Въведете страна и височина към нея:');
            readln(a, h);
            s := a * h / 2;
        end;
    else begin
        writeln('Грешен номер на фигура !');
        goto 1
    end
end;

```

```
writeln('Лице:', s:10:2);  
end.
```

Забележка: π е библиотечна константа за числото π . В програмата не се проверява дали дължините на отсечките са положителни числа.

Съдържание

3. Изрази: аритметични и логически изрази, видове операции и приоритет за тяхното изпълняване.

5. Оператори за цикъл. Оператори за цикъл с брояч, с предусловие и с постусловие.

5. Оператори за цикъл. Оператори за цикъл с брояч, с предусловие и с постусловие.

Дефиниция 5.1

Цикъл в един алгоритъм или програма наричаме група от една или няколко команди, които се изпълняват многократно. Циклите, организирани чрез специалните за целта оператори в Pascal се наричат *явни*, а тези, организирани чрез други средства на езика се наричат *неявни*.

Един начин за организиране на неявен цикъл бе показан в [предходната точка](#) чрез преход **goto** назад в програмата. Друг начин е чрез рекурсия, която ще бъде описана в [седма точка](#).

В Pascal има четири оператора за явен цикъл. Те са:

- [два оператора за цикли с брояч](#),
- [оператор за цикъл с предусловие и](#)
- [оператор за цикъл с постусловие](#).

Цикли с брояч. Това са цикли, при които предварително знаем, колко пъти ще се завъртим. При тях имаме целочислена променлива-брояч, която брой завъртанията (итерациите) на цикъла. Единият оператор е за цикъл с нарастване на брояча. Той има следния формат:

```
for променлива_брояч := израз1 to израз2 do  
    команда
```

където израз1 задава началната стойност, която приема брояча преди започването на цикъла, израз2 задава крайната стойност на брояча, при която цикълът продължава да се върти, а команда е тялото на цикъла, което се изпълнява многократно. Ако в цикъл трябва да се завъртят не една команда, а няколко, тогава се използва съставен оператор. При всяко завъртане на цикъла броячът нараства с едно. Цикълът завършва, когато стойността на брояча надхвърли стойността на израз2. Ако

израз1 > израз2,

тогава тялото на цикъла въобще не се изпълнява, а се преминава към следващата команда след цикъла, иначе броят на завъртанията е

израз2 - израз1 + 1.

Вътре в тялото на цикъла броячът не може да се променя.

Пример 5.1:

```
i : integer;  
for i := 1 to 10 do  
  writeln( i );
```

Този фрагмент от програма ще изведе числата от 1 до 10.

Другият оператор е за цикъл с намаляване на брояча. Той има следния формат:

```
for променлива_брояч := израз1 downto израз2 do  
  команда
```

Неговото действие е сходно с това на предходния оператор с тази разлика, че при всяко завъртане на цикъла броячът намалява с едно. Затова при него за да се завърти цикъла би трябвало

израз1 > израз2,

иначе тялото на цикъла се прескача, а броя на завъртанията е

израз1 - израз2 + 1.

Пример 5.2:

```
i : integer;  
for i := 10 downto 1 do  
  writeln( i );
```

Този фрагмент от програма ще изведе числата от 1 до 10 в обратен ред.

Задача 5.1: Да се състави програма, която намира средното аритметично на N реални числа.

Анализ:

Входни данни: $N > 0$ цяло

$a_1, a_2, \dots, a_N$ - реални.

$$s = \frac{a_1 + a_2 + \dots + a_N}{N}$$

Изход:

Алгоритъм

Начало

1. Въведи N
2. Ако $N \leq 0$
Изведи Грешен брой
Отиди към 1
3. $s \leftarrow 0$
4. За $i \leftarrow 1$ до N повтаряй
Въведи a_i
 $s \leftarrow s + a_i$
5. $s \leftarrow s / N$
6. Изведи s

Край

Програма

```
program average;
uses WinCrt;
label 1;
var
  a, s: real;
  i, n: integer;
begin
1:  write('Въведете броя на числата');
   readln(n);
   if n <= 0
   then begin
        writeln('Грешен брой!');
        goto 1 { неявен цикъл }
     end;
   s := 0;
   for i := 1 to n do
   begin
        write('Въведете ', i, '-то число:');
        readln(a);
        s := s + a;
     end;
   s := s / n;
   writeln('Средно аритметично:', s:10:4);
end.
```

Цикъл с предусловие. Това е цикъл, който се изпълнява, ако е налице някакво условие (логически израз е истина). Първо се проверява условието и ако то е изпълнено, тогава се изпълнява тялото на цикъла, състоящо се от една или няколко команди. После пак се проверява условието и т.н. Цикълът завършва, когато условието повече не е изпълнено (логическият израз стане лъжа). Ако първоначално условието не е изпълнено, тялото се прескача и без то да се изпълни нито веднъж се преминава нататък в програмата. Формата на оператора за цикъл с предусловие е следния:

```
while логически_израз do  
    команда
```

където логически_израз е условието за изпълнение на цикъла, а команда е неговото тяло, което се завърта многократно. Ако в цикъл трябва да се завърти не една, а няколко команди, тогава се използва съставен оператор.

Ако при изпълнение на цикъла логическият израз остава постоянно истина (**true**), то цикълът няма да завърши и ще се получи безкраен цикъл. Т.е. тялото на цикъла трябва да въздейства на логическия израз, така че той в един момент да се промени от **true** на **false** и цикълът да завърши.

Пример 5.3:

```
а: real;  
а := 0;  
while а <= 10 do  
    а := а + 2;
```

Преди цикъла а е 0. При всяко завъртане на цикъла а се увеличава с 2. В края на цикъла а ще стане 12.

Задача 5.2. Да се състави програма, намираща най-големия делител на дадено цяло положително число.

Анализ

Входни данни: $N > 0$ - цяло.

Изходни данни: К - цяло.

Знание: Най-големият делител е първото число К като се започне от $N \text{ div } 2$ до 1, за което $N \bmod K = 0$.

Алгоритъм

Начало

1. Въведи N
2. Ако $N \leq 0$ тогава
 Изведи "Числото не е положително"
 Отиди към 1
3. Ако $N = 1$ тогава
 $K \leftarrow 1$
 иначе
 $K \leftarrow N \text{ div } 2$

4. Докато ($K > 1$) и ($N \bmod K \neq 0$) повтаряй
 $K \leftarrow K - 1$
5. Изведи K
Край

Програма

```
program max_dividor;  
  uses WinCrt;  
  label 1;  
  var  
    k, n: integer;  
  begin  
1:  write('Въведете цяло положително число:');  
    readln(n);  
    if n <= 0 then  
      begin  
        writeln('Числото не е положително !');  
        goto 1  
      end;  
    if n = 1  
      then k := 1  
      else k := n div 2;  
    while (k > 1) and (n mod k <> 0) do  
      k := k - 1;  
    writeln('Най-голям делител на ', n, ' е числото ', k);  
  end.
```

Цикъл с постусловие. При този цикъл условието за края му (логически израз) е след неговото тяло (повтарящите се команди). Цикълът действа по следния начин: първо се изпълнява неговото тяло, проверява се условието за край и ако то не е изпълнено (логическият израз е **false**), тялото отново се изпълнява и т.н. Цикълът завършва, когато се изпълни условието за край (логическият израз е **true**). Формата на оператора за цикъл с постусловие е следния

```
repeat  
  команда1;  
  команда2;  
  .....  
  командаN  
until логически_израз
```

където командите от 1 до N образуват тялото на цикъла. И тук, ако логическият израз остане винаги **false**, цикълът е безкраен. Основната разлика между циклите с предусловие и постусловие е, че при първия тялото може въобще да не се изпълни (ако условието първоначално е лъжа), докато при втория то се изпълнява поне веднъж.

Пример 5.4

```

n : integer;
repeat
  write('Въведете цяло положително число:');
  readln(n)
until n > 0;

```

Този фрагмент от програма осигурява, че потребителят ще въведе положително число.

Задача 5.3. Да се състави програма за изчисляване на кубичен корен от x , като се

$$\beta = \frac{2\alpha + \frac{x}{\alpha^2}}{3}$$

използва, че ако α е някакво приближение до търсения корен, то β е по-добро приближение.

Анализ

Входни данни: x - реално

Изходни данни: β – реално.

Знание: Ще се започне от $\alpha = 1$ и по горната формула ще се получи β . После $\alpha = \beta$ и отново се прилага формулата, като се получава ново още по-точно β и т.н., докато разликата между две последователни приближения стане по-малка от някаква зададена точност - много малко число.

Алгоритъм

□□□ $\varepsilon = 0.0001$

Начало

1. Въведи x

2. $\beta \leftarrow 1$

3. Повтаряй

$\alpha \leftarrow \beta$

$$\beta \leftarrow \frac{2\alpha + \frac{x}{\alpha^2}}{3}$$

докато $|\beta - \alpha| < \varepsilon$

4. Изведи β

Край

Програма

```

program cubrt;
uses WinCrt;
const t = 0.0001;
var
  x, a, b: real;
begin
  write('x = ');
  readln(x);
  b := 1;

```

```

repeat
    a := b;
    b := (2 * a + x / (a * a)) / 3
until abs(b - a) < t;
writeln('Cubrt(', x:8:3, ') =', b:10:4);
end.

```

Забележка. abs е библиотечна функция за абсолютна стойност.

Съдържание

4. Оператори за логическо управление на програмата. Логически оператор. Съставен оператор (оператор-блок). Оператор за безусловен преход. Оператор за избор.

6. Деклариране на нови типове в Pascal. Тип диапазон. Масиви: едномерни, двумерни и тримерни масиви. Индексиране на масив. Обхождане на масив.

6. Деклариране на нови типове в Pascal. Тип диапазон. Масиви: едномерни, двумерни и тримерни масиви. Индексиране на масив. Обхождане на масив.

- Деклариране на нови типове в Pascal.
- Тип диапазон.
- Масиви:
- едномерни.
- двумерни и тримерни масиви.

Освен вградените в езика типове като **integer**, **real**, **char** и др. програмистът може да дефинира нови типове. Дефинирането на нови типове става в декларативната част на програмата преди декларирането на променливи и след декларирането на константи. То започва с резервираната дума **type** и следват дефинициите на новите типове. Дефиницията на нов тип има вида

нов_тип = дефиниция;

Новият тип може да се ползва за дефиниране на променливи от този тип.

Пример 6.1:

```

type
    broj_studenti = integer;

```

```

zaveril = boolean;
ocenka = real;
var
n: broj_studenti;
sem, god: zaveril;
uspeh: ocenka;

```

В този пример имаме всъщност предефиниране на съществуващи типове. Това понякога се прави с цел по-добра четаемост на програмата. По-нататък ще бъдат показани по-сложни дефиниции на нови типове.

Тип диапазон. Този тип се дефинира въз основата на изброим тип данни. Изброими данни са тези, които за всяка тяхна стойност може да се посочи, коя е следващата или предходната стойност и между две поредни стойности няма трета между тях. Такива са вградените типове **integer**, **boolean** и **char**. Например: при **integer** след 5 следва 6 и между тях няма друго цяло число. **real** не е изброим тип. Ако си спомняте в оператор **case** се ползва израз от изброим тип. Тип диапазон се дефинира по следния начин

начална_стойност .. крайна_стойност

където начална_стойност е константа задаваща началото на диапазона, а крайна_стойност е константа задаваща неговия край. Трябва да е изпълнено, че

начална_стойност $\leq$ крайна_стойност.

Между начална_стойност и крайна_стойност се поставят точно две последователни точки "..". Те са една буква от езика Pascal и не могат да се разделят една от друга с интервал, нов ред и т.н..

Пример 6.2:

```

type
mesec = 1 .. 12;
sedmica = 1 .. 7;
var
m: mesec;
day: sedmica;
glavna_bukva: 'A' .. 'Я';

```

За променливите от тип диапазон не е необходимо в програмата да се организира проверка за коректност на диапазона. Компиляторът автоматично генерира такъв код навсякъде в крайната програма, където на променлива от такъв тип се присвоява стойност. Недостатъкът в случая е, че ако променлива от тип диапазон излезе извън него, програмата веднага се прекратява с неразбираемо съобщение за грешка.

Масиви. Данните, с които работи една програма, са единични (*скаларни*) данни и структури от данни. Единични данни са число, буква или логическа стойност. Скаларни типове са вградените типове **integer**, **real**, **char** и **boolean**, а също и тип диапазон.

Дефиниция 6.1

Структури от данни са организирани и подредени по някакви правила съвкупности от данни. Те биват статични и динамични. *Статични* са такива структури, чието организация и брой елементи се задава при тяхната дефиниция и не може повече да се променя.

Езикът Pascal поддържа следните вградени структури:

- масив,
- низ от символи,
- запис и
- файл.

Дефиниция 6.2 Масив е крайна, наредена съвкупност от еднотипни данни.

Еднотипните елементи на масива могат да бъдат, както скаларни данни, така и структури от данни с изключение на файлове. Те могат да са, както от вградените в езика типове, така и от типове дефинирани от програмиста. Според наредбата на елементите си масивите биват едномерни, двумерни, тримерни и т. н. Масивът е статична структура от данни. Броят на елементи му се задава при неговата декларация и трябва да бъде константен. Могат да се дефинират типове-масиви и направо променливи-масиви.

Едномерни масиви. Променлива едномерен масив се дефинира по следния начин

име: **array** [целочислен_диапазон] **of** тип;

където целочислен_диапазон определя, от къде до къде се изменят номерата (индексите) на елементите на масива, а тип определя типа на тези елементи. Границите на диапазона трябва да са целочислени константи и от тях и от типа компилатора определя, колко памет да отпусне за масива. Едномерният масив отговаря на математическото понятие вектор.

Пример 6.3:

```
const
    max = 100;
type
    diapazon = 1 .. max;
    masiv = array [ diapazon ] of integer;
var
    a: masiv;
    b, c: array [-3 .. 4] of real;
```

Масивът a е от 100 цели числа, а масивите b и c са от по 8 реални числа.

Вътре в тялото на програмата се работи с елементите на масива по същия начин както с променливи от съответния тип. Обръщението към отделните елементи става по следния начин

име[индекс]

където име е името на променливата масив, а индекс е целочислен израз със стойност в диапазона на номерацията на неговите елементи. Програмистът внимателно трябва да следи индекса да не излиза извън рамките на този диапазон, защото в едни случаи това води до прекъсване на програмата, а в други - до погрешна работа на програмата.

Пример 6.4 (виж Пример 6.3):

```

var
  i, j: integer;
begin
  .....
  readln(a[2 * i + j - 1]);  { стойността на 2*i + j - 1 трябва да от 1 до 100 }
  b[-2] := -5.3;
  c[5] := b[-2] / 4;        { Грешка ! Индекс 5 е извън диапазона на c. }
  for i := 1 to max do
    writeln(a[i]);
  .....
end.

```

Една от най-често използваните дейности с масиви е обхождане на масив. Обхождане на една структура е достигане с някаква цел до всеки неин елемент. Обхождането на едномерен масив става чрез цикъл - обикновено **for**. В края на последния пример се обхожда масива а, с цел извеждане на екрана всички негови елементи.

Задача 6.1. Да се изчисли следната формула $\sqrt{\frac{1}{N} \sum_{i=1}^N (a_i + s)^2}$,

където $s = \frac{a_1 + a_2 + \dots + a_N}{N}$.

Анализ

Входни данни: $N > 0$ цяло,

$a_1, a_2, \dots, a_N$ - реални.

Исходни данни $G = \sqrt{\frac{1}{N} \sum_{i=1}^N (a_i + s)^2}$

Алгоритъм

Начало

1. Повтаряй ;контролира се въведеното $N > 0$
Въведи N
докато $N > 0$
2. $s \leftarrow 0$
3. За $i \leftarrow 1$ до N повтаряй ;въвеждат се елементите на a и се сумират в s
Въведи a_i
 $s \leftarrow s + a_i$
4. $s \leftarrow s / N$
5. $G \leftarrow 0$
6. За $i \leftarrow 1$ до N повтаряй ;намира се сумата вътре във формулата

$$G \leftarrow G + (a_i + s)^2$$

$$7. \quad G \leftarrow \sqrt{\frac{G}{N}}$$

8. Изведи G

Край

Програма

```

program deviation;
uses WinCrt;
const max = 100;
var
  a: array [1 .. max] of real;
  s, g: real;
  i, n: integer;
begin
  repeat { контролера се n да е в границите на масива }
    write( ' N = ' );
    readln( n );
    until (n > 0) and (n <= max);
  s := 0;
  for i := 1 to n do begin
    write( 'Въведете ', i, '-то число:');
    readln( a[i] );
    s := s + 1;
  end;
  s := s / n;
  g := 0;
  for i := 1 to n do
    g := g + sqr(a[i] + s);
  g := sqrt(g / n);
  writeln('G = ', g:10:4);
end.

```

Забележка. **sqr** е библиотечна функция за повдигане на квадрат.

Двумерен масив. В този масив елементите са подредени по редове и стълбове. Той отговаря на математическото понятие матрица или таблица. Променлива двумерен масив се декларира по следния начин

име: **array** [целочислен_диапазон1, целочислен_диапазон2] **of** тип;

където целочислен_диапазон1 задава диапазона на номерация на редовете на масива, а целочислен_диапазон2 - на стълбовете.

Пример 6.5:

```

const
  max = 100;

```

```

type
    diapazon = 1 .. max;
    masiv2d = array [diapazon, diapazon] of integer;
var
    m: masiv2d;
    x, y: array [-3 .. 4, -1 .. 9] of real;

```

Масивът m е от 10 000 цели числа, а масивите x и y са от по 88 реални числа.

Обръщението към елемент на двумерен масив става чрез два индекса:

```
име[ индекс1, индекс2]
```

където индекс1 и индекс2 са целочислени изрази със стойности съответно в целочислен_диапазон1 и целочислен_диапазон2. И в този случай програмистът трябва да следи тези стойности да не напускат съответните диапазони.

Пример 6.6 (виж Пример 6.5):

```

var
    i, j: integer;
begin
    .....
    m[25, 34] := 143;
    i := 10;
    j := 15;
    x[i - 7, j div 5 - 1] := 3.43;
    for i := -3 to 4 do
        for j := -1 to 9 do
            readln( y[i, j]);
    .....
end.

```

Обхождането на двумерен масив става чрез двоен цикъл т. е. цикъл вътре в тялото на друг цикъл. В последния пример имаме обхождане на масива y с цел въвеждане на стойности на елементите му от клавиатурата. Обхождането на двумерни масиви става по два начина: по редове и по стълбове. В горния пример имаме обхождане по редове, където във външния цикъл се изменя първия индекс (за редовете), а във вътрешния цикъл се изменя втория индекс (за стълбовете). Т. е. за i-то завъртане на външния цикъл имаме i-ти ред и с вътрешният цикъл преминаваме през елементите на този ред. При обхождане по стълбове във външния цикъл се изменя втория индекс, а във вътрешния - първия.

Задача 6.2. Дадена е таблица от N реда и M стълба с реални числа. Да се състави програма, която изчислява сумите от стълбовете на таблицата, входните данни и резултатите да се покажат в табличен вид.

Анализ

Входни данни: $N > 0$, $M > 0$ - цели,

$a_{1,1}, a_{1,2}, \dots, a_{1,M}$

$a_{2,1}, a_{2,2}, \dots, a_{2,M}$

.....

$a_{N,1}, a_{N,2}, \dots, a_{N,M}$ - реални

Изходни данни

$a_{1,1}, a_{1,2}, \dots, a_{1,M}$

$a_{2,1}, a_{2,2}, \dots, a_{2,M}$

.....

$a_{N,1}, a_{N,2}, \dots, a_{N,M}$

 $b_1, b_2, \dots, b_M$

Знание:
$$b_j = \sum_{i=1}^M a_{i,j}$$

Алгоритъм

Начало

1. Повтаряй

Въведи N, M

докато $N > 0$ и $M > 0$

2. За $i \leftarrow 1$ до N повтаряй ;Двоен цикъл - обхождане на a по редове

За $j \leftarrow 1$ до M повтаряй

Въведи $a_{i,j}$

3. За $j \leftarrow 1$ до N повтаряй ;Двоен цикъл - обхождане на a по стълбове

$b_j \leftarrow 0$

;Сумиране на стълбовете в масива b

За $i \leftarrow 1$ до M повтаряй

$b_j \leftarrow b_j + a_{i,j}$

4. За $i \leftarrow 1$ до N повтаряй ;Двоен цикъл - обхождане на a по редове

За $j \leftarrow 1$ до M повтаряй

Изведи $a_{i,j}$

Изведи нов ред

5. За $j \leftarrow 1$ до M повтаряй

Изведи b_j

Край

Програма

program tablesum;

uses WinCrt;

const max = 100;

var

a : **array** [1 .. max, 1 .. max] **of** real;

b : **array** [1 .. max] **of** real;

i, j, n, m : **integer**;

begin

repeat { контролера се n и m да са в границите на масива }

write(' N = ');

```

readln( n );
write( ' M = ' );
readln( m );
until ( n > 0) and ( n <= max) and ( m > 0) and ( m <= max);
for i := 1 to n do
  for j:= 1 to m do begin
    write( 'a[', i, ',', j, ']=');
    readln( a[i, j] )
  end;
for j := 1 to m do begin
  b[i] := 0;
  for i := 1 to n do
    b[i] := b[i] + a[i, j]
end;
for i := 1 to n do begin
  for j:= 1 to m do
    write( a[i, j]: 8: 2);
  writeln
end;
for j:= 1 to 8 * m do
  write( '-' );
writeln;
for j:= 1 to m do
  write(b[j]: 8: 2);
end.

```

Тримерни масиви. Те се декларират и ползват подобно на двумерните. При декларирането си имат три диапазона за изменение на индексите си. При ползване към техните елементи се обръщаме с три индекса. Тримерните масиви се обхождат с троен цикъл.

Съдържание

5. Оператори за цикъл. Оператори за цикъл с брояч, с предусловие и с постусловие.

7. Подпрограми. Видове подпрограми в Pascal. Параметри на подпрограмите. Подпрограма процедура. Подпрограма функция. Локални и глобални величини в програмите. Предварително деклариране на подпрограма. Ползване на готови библиотеки от подпрограми. Рекурсия.

7. Подпрограми. Видове подпрограми в Pascal. Параметри на подпрограмите. Подпрограма процедура. Подпрограма функция.

Локални и глобални величини в програмите. Предварително деклариране на подпрограма. Ползване на готови библиотеки от подпрограми. Рекурсия.

- [Подпрограми.](#)
- [Подпрограма процедура.](#)
- [Подпрограма функция.](#)
- [Локални и глобални величини в програмите.](#)
- [Предварително деклариране на подпрограма.](#)
- [Ползване на готови библиотеки от подпрограми.](#)
- [Рекурсия.](#)

Дефиниция 7.1

Подпрограма е част от програмата, в която се отделя някакво логически обособено действие. Подпрограмите се описват само веднъж и към тях се обръщат от много места в главната програма или в подпрограмите.

Възможно е една подпрограма да вика себе си, което се нарича рекурсия. В Pascal една подпрограма може да има собствени подпрограми, т. е. получава се йерархия от подпрограми, което съответства на инженерния подход нещо сложно да се раздели на няколко по-прости неща, а те от своя страна също да се разделят на по-прости и т. н. Могат да се посочат следните причини за създаване на подпрограми:

- дълга и сложна програма - едно изискване на структурното програмиране е, че ако програмата стане по-дълга от двайсетина команди, тя да се раздели на подпрограми. Дългата и сложна програма зависи от много параметри и трудно се тества за да се изчисти от логически грешки. Кратките подпрограми зависят от малко параметри и по-лесно се тестват;
- често срещани в програмата действия. Най-често срещаните действия са обособени в библиотеки от готови подпрограми;
- създаване на големи проекти от колектив от програмисти. Всеки програмист създава няколко подпрограми от проекта, обособени в отделен модул.

Една подпрограма обменя данни с останалата част от програмата по два начина: чрез параметри и чрез глобални по отношение на нея променливи и константи.

Когато една подпрограма се описва, параметрите и се наричат формални, защото нямат конкретна стойност, а служат само за нейното описание. Когато една подпрограма се вика, нейните параметри се наричат фактически и имат конкретни стойности. Параметрите имат тип, който задава типа на данните, обменяни между подпрограмата и останалата част на програмата. Параметрите на подпрограмите са два вида:

- параметри по стойност - служат да подадат стойност от викащата подпрограмата програмна част, те не могат да се променят вътре в подпрограмата и
- параметри по име (по адрес) - служат да подадат стойност в двете посоки както към подпрограмата, така и от подпрограмата към викащата я програмна част, те обикновено получават стойност или си променят стойността вътре в подпрограмата.

Когато подпрограмата се вика, броят на фактическите параметри е равен на броя на формалните в нейното описание. На мястото на фактическия параметър по стойност се поставя израз, чието стойност е от типа на параметъра и се подава към подпрограмата. На мястото на фактическия параметър по име се поставя променлива от тип еднакъв с типа на съответния формален параметър. Чрез тази променлива се осъществява обмяна на стойности между подпрограмата и викащата програмна част. Редът на описание на параметрите е съществен, защото в същия ред се задават фактическите параметри при викане на подпрограмата. Както вече бе казано, една подпрограма може да бъде викана много пъти, като при всяко викане фактическите параметри са различни.

Ако като параметър се иска да се използва масив или диапазон, трябва да се дефинира съответен нов тип за масива или диапазона и параметъра да е от този тип.

Нека разгледаме общата структура на една програма на Pascal за да видим, къде се описват подпрограмите.

program заглавие

uses списък от имена на библиотеки с готови подпрограми

label списък от етикети

const дефиниране на константи

type описание на нови типове

var деклариране на променливи

описание на подпрограми

begin

тяло на програмата

end.

В Pascal има два вида подпрограми: процедури и функции.

Подпрограма процедура. Този вид подпрограма отделя в себе си някакво логически обособено действие. Там където в програмата трябва да се осъществи това действие, се вика процедурата. Това обръщение към нея се реализира в езика като една отделна команда. Описанието на процедура става по следния начин:

procedure име_на_процедура (списък_от_формални_параметри);

label списък от етикети

const дефиниране на константи

type описание на нови типове

var деклариране на променливи

описание на подпрограми вътрешни за процедурата

begin

тяло на процедурата

end;

където име_на_процедура е идентификатор. Тялото на процедурата е също поредица от команди като тялото на програма. Описаните в процедурата подпрограми са вътрешни за нея и не могат да се викат извън нея. Същото се отнася и за етикетите, константите, новите типове и променливите зададени вътре в процедурата - те се ползват само вътре в нея. Ако процедурата няма параметри, то в заглавието и няма (списък_от_формални_параметри).

Отделните елементи на списъка от формални параметри се отделят един от друг чрез ; . Един формален параметър се описва по следния начин

име : тип

ако е параметър по стойност, или

var име : тип

ако е параметър по име. Типът може да е някой от вградените в езика типове като **integer**, **real**, **char** или да е някой от описаните вече в програмата нови типове. Ако имаме няколко еднотипни параметри всичките по стойност или по име, то те могат да се зададат в списък

параметър1, параметър2, ... , параметърN : тип

Пример 7.1:

type

meseci = 1 .. 12;

masiv = **array**[meseci] **of real**;

procedure example (**var** a : masiv; **var** b, c : real; d : real; n : integer);

type

sedmica = 1 .. 7; { нов тип дефиниран в процедурата }

var

i: integer;

m : sedmica;

begin

for i := 1 **to** 12 **do**

 readln(a[i]); { a се променя, защото е параметър по име }

 b := (b + c) / n; { b и c се променят защото са параметри по име }

 c := (b - c) / d;

 d := a[3] / n; { грешка - d е параметър по стойност }

 m := n **mod** 7 + 1;

 writeln(m)

end;

В тази примерна процедура параметрите a, b и c са по име и затова се променят вътре в нея. Параметрите d и n са по стойност и не би трябвало да се променят. Типът sedmica и променливите i и m са дефинирани вътре в процедурата и действат само вътре в нея.

Процедурата се вика като една отделна команда

име_на_процедура(списък_от_фактически_параметри)

като списъкът от фактически параметри трябва да отговаря на съответния списък от формални параметри: по брой, по тип и по вид (по име или по стойност). Ако процедурата няма параметри, тя се вика само чрез своето име. Списъкът от фактически параметри има следния вид

параметър1, параметър2, .. , параметърM

като отделните параметри са изрази, ако са по стойност или променливи, ако са по име.

Пример 7.2 (виж пример 7.1)

```
var
  a, b : masiv;
  x , y :real;
  k : integer;
begin
  .....
  x := 1;   y := 2;   k := 0;
  example(b, x, y, x + 2 * y, k + 6);
  { След процедурата x = 0.5, y = -0.2 }
  .....
  example(a, y, x, y - x / k, 20);
  .....
end.
```

В примера има две обръщения към процедурата example. Първите три параметъра на тази процедура са по име и затова фактическите параметри са променливи, които след излизане от процедурата имат нови стойности. Последните два параметъра са по стойност и затова фактическите параметри са изрази.

Задача 7.1. Да се състави програма за умножение на две матрици, ако е възможно.

Анализ.

Входни данни: Размерности на матрица A: $n > 0$ и $m > 0$; на матрица B: $l > 0$ и $k > 0$ - цели числа. Поради това, че при умножението на двете матрици се умножават елементите на ред от първата матрица със стълб от втората, то трябва $m = l$.

Матрицата A

$a_{1,1}, a_{1,2}, \dots, a_{1,m}$

$a_{2,1}, a_{2,2}, \dots, a_{2,m}$

.....

$a_{n,1}, a_{n,2}, \dots, a_{n,m}$ - реални числа.

Матрицата B

$b_{1,1}, b_{1,2}, \dots, b_{1,k}$

$b_{2,1}, b_{2,2}, \dots, b_{2,k}$

.....

$b_{M,1}, b_{M,2}, \dots, b_{M,k}$ - реални числа.

Исходни данни: Резултантната матрица C е с n реда и k стълба и елементи

$$c_{i,j} = \sum_{p=1}^M a_{i,p} b_{p,j} \quad \text{за } i=1, \dots, n \text{ и } j=1, \dots, k$$

Програмата ще има 3 процедури: за въвеждане на двумерен масив, за извеждане в матричен вид на двумерен масив и за умножението на два масива.

Програма

```
program matrixmulty;
uses WinCrt;
const max = 50;
type
  matrix = array[1 .. max, 1 .. max] of real;
var
  a, b, c: matrix;
  k, l, m, n: integer;

procedure vvhod(var a: matrix; n, m: integer);
{ Процедура за въвеждане на двумерен масив
  a - масива, който се въвежда - параметър по име,
  n - брой редове, m - брой стълбове - параметри по стойност }
var
  i, j: integer;
begin
  for i:= 1 to n do
    for j:= 1 to m do begin
      writeln('Елемент от ред ', i, ' и стълб ', j, '=');
      readln(a[i, j])
    end
  end;
end;

procedure izhod(a: matrix; n, m: integer);
{ Процедура за извеждане на двумерен масив в матрична форма
  a - масива, който се извежда - параметър по стойност,
  n - брой редове, m - брой стълбове - параметри по стойност }
var
  i, j: integer;
begin
  for i:= 1 to n do begin
    for j:= 1 to m do
      write(a[i, j]):8:2; { Извежда се ред от масива }
    writeln;              { Нов ред }
  end;
end;
```

```

procedure multiply(a, b: matrix; var c: matrix; n, m, k: integer);
{ Процедура за умножение на матриците a и b - параметри по стойност,
  c - резултантна матрица - параметър по име,
  n и m - размерност на матрицата a,
  m и k - размерност на матрицата b,
  n и k - размерност на матрицата c }
var
  i, j, p: integer;
begin
  for i:= 1 to n do
    for j:= 1 to k do begin
      c[i, j]:= 0;           { Сума на m събираеми }
      for p:= 1 to m do
        c[i, j] := c[i, j]+a[i, p] * b[p, j]
      end
    end;
end;

begin                                { Главна програма }
repeat
  writeln('Въведете брой редове и стълбове за матрица A:');
  readln(n, m)
  until (n > 0) and (n <= max) and (m > 0) and (m <= max);
  writeln('Въведете елементите матрица A:');
  viod(a, n, m);                     { I обръщение към viod }
  writeln('Матрица B трябва да има ', m, ' реда. ');
  repeat
    writeln('Въведете брой стълбове за матрица B:');
    readln(k)
    until (k > 0) and (k <= max);
  writeln('Въведете елементите матрица B:');
  viod(b, m, k);                     { II обръщение към viod }
  multiply(a, b, c, n, m, k);
  { Обръщение към процедура multiply. B с се получава резултата
    от умножението на a и b. }
  writeln('Матрица A');
  izhod(a, n, m);                    { I обръщение към izhod }
  writeln('Матрица B');
  izhod(b, m, k);                    { II обръщение към izhod }
  writeln('Матрица C = A x B');
  izhod(c, n, k);                    { III обръщение към izhod }
end.

```

Подпрограма функция. Функцията за разлика от процедурата освен, че извършва някакво логически обособено действие, изчислява някаква стойност. Затова функцията се вика вътре в изразите, където е необходима тази стойност. Функцията има тип - типа на изчислената от нея стойност. Описанието на функция става по следния начин

```

function име_на_функция (списък_от_формални_параметри): тип;
  label списък от етикети
  const дефиниране на константи

```

type описание на нови типове
var деклариране на променливи

описание на подпрограми вътрешни за функцията

begin

тяло на функцията

.....

име_на_функция := израз

.....

end;

където име_на_функция е идентификатор, тип е типа на стойността изчислявана от функцията. Тази стойност трябва да е скаларна или от тип низ от символи и не може да е от тип масив, множество или запис. Типът може да е някой от вградените в езика скаларни типове като **integer**, **real**, **char**, тип **string** или да е някой от описаните вече в програмата нови типове. Описанието на формалните параметри е същото като при процедури. Ако функцията няма параметри, то в заглавието и няма (списък_от_формални_параметри).

Вътре в тялото на функцията на едно или няколко места трябва да има оператор за присвояване, където на нейното име се присвоява стойността на израз от тип еднакъв с този на типа на функцията. Това е стойността, която тя изчислява и връща при нейно извикване. Затова логиката на функцията трябва да е такава, че задължително да се премине през такъв оператор и тя да получи стойност. Ако функцията завърши без на нейното име да се присвои стойност, тя връща някаква случайна стойност - "боклук".

Пример 7.3:

```
const max = 100;
type
  masiv = array[1 .. max] of real;

function fsin( x: real; a: masiv; n: integer; var err: boolean): real;
  const pi = 3.1416;
  var
    s: real;
    i: integer;
  begin
    err := false;
    if x < 0
    then begin
      err:= true;
      fsin := 0
    end
```

```

else if (0 >= x) and (x <= pi) then
  begin
    s:= 0;
    for i := 1 to n do
      s:= s + sin(a[i] * x);
    fsin := s
  end
else fsin := -sin(x)
end;

```

Функцията fsin изчислява следната математическа функция

$$f(x) = \begin{cases} \sum_{i=1}^n \sin(a_i x) & \text{за } 0 \leq x \leq \pi \\ -\sin(x) & \text{за } x \geq \pi \\ \text{не е дефинирана} & \text{за } x < 0 \end{cases}$$

Понеже $f(x)$ не е дефинирана за $x < 0$, в този случай fsin връща 0, защото все пак тя трябва да върне някаква стойност и параметърът по име ег е **true**, а за всички останали стойности за x ег е **false**. Така чрез параметъра ег викащата fsin програмна част може да разбере, дали се изчислява коректна стойност.

Функцията се вика вътре в изразите, там където е необходима стойността, която тя изчислява. Това става по следния начин:

```
име_на_функция(списък_от_фактически_параметри)
```

като списъкът от фактически параметри трябва да отговаря на съответния списък от формални параметри: по брой, по тип и по вид (по име или по стойност). Ако функцията няма параметри, тя се вика само чрез своето име. Задаването на фактическите параметри става по същия начин както при процедурите.

Пример 7.4 (виж пример 7.3):

```

var
  a, b: masiv;
  x, y, z: real;
  undef: boolean;
begin
  .....
  readln(x);
  y:= 5.3 + fsin(x, a, 10, undef) / 2;
  if undef then
    begin
      writeln('Функцията не дефинирана за x < 0');
      exit
    end;
  z := y + fsin(y, b, 15, undef) * x;

```

```
writeln(fsin(z, a, 10, undef):10:4);
.....
end.
```

В този пример функцията fsin се вика три пъти. Чрез променливата undef се проверява дали тя работи коректно. Библиотечната процедура exit прекъсва програмата.

Задача 7.2: Дадени са n реални числа. Да се изчисли формулата $\sum_{i=1}^n a_i^m$, за m - цяло число.

Анализ

Входни данни: $n > 0$ цяло,
 $a_1, a_2, \dots, a_n$ - реални,
 m - цяло число.

Изходни данни $f =$

Знание: За да се реши задачата е необходимо да имаме функция за степенуване. Тази функция има следния алгоритъм

Параметри a - реално, m - цяло.

Начало

```
1. Ако  $a = 0$ 
    резултат 0
иначе Ако  $m = 0$ 
    резултат 1
иначе Ако  $m < 0$ 
     $b \leftarrow 1 / a$ 
     $n \leftarrow -m$ 
иначе
     $b \leftarrow a$ 
     $n \leftarrow m$ 
     $p \leftarrow 1$ 
    За  $i \leftarrow 1$  до  $n$  повтаряй
         $p \leftarrow p * b$ 
    резултат  $p$ 
```

Край

Програма

```
program formula;
uses WinCrt;
const max = 100;
type
    masiv = array[1 .. max] of real;
var
    a: masiv;
    n, m, i: integer;
```

```

function power(a: real; m: integer): real; { Функция за  $a^m$  }
    var
        i,n: integer;          { n от функцията няма нищо общо с n от главната
програма }
        p, b: real;
    begin
        if a = 0
        then power := 0
        else if m = 0
        then power := 1
        else begin
            if m < 0
            then begin          {  $a^m = (1/a)^{-m}$  }
                b := 1 / a;
                n := -m
            end
            else begin
                b := a;
                n := m
            end;
            p := 1;
            for i := 1 to n do
                p := p * b
            power := p
        end
    end;

```

```

function f(a: masiv; n, m: integer): real;
{ Функция изчисляваща формулата }
    var
        i: integer;
        s: real;
    begin
        s := 0;
        for i:= 1 to n do
            s := s + power(a[i], m);
        f := s
    end;

```

```

begin          { Главна програма }
    repeat
        write('n =');
        readln(n)
    until (n > 0) and (n <= max);
    writeln('Въведете масива');
    for i := 1 to n do begin
        write('a', i, '=');
        readln(a[i])
    end;
    write('m =');

```

```

readln(m);
writeln('Резултат:', f(a, n, m):10:4);
end.

```

Задача 7.3: Да се изчисли определен интеграл от непрекъсната функция $f(x)$ за интервал $[a, b]$.

Анализ

Входни данни: a и b - реални и $a < b$;

$f(x)$ - някаква непрекъсната функция на x

$$s = \int_a^b f(x) dx$$

Изходни данни:

Знание: За да се реши тази задача е необходимо да се избере някой от съществуващите числени методи. Избираме метода на правоъгълниците. При нея интервалът $[a, b]$ се разделя на N части. За всяка част се изчислява лицето на правоъгълник с основа дължината на частта $(b - a) / N$ и височина стойността на $f(x)$ в средата на частта. Интегралът е сумата от тези

правоъгълници $s = \frac{b-a}{N} \sum_{i=1}^N f\left(\frac{x_{i-1} + x_i}{2}\right)$, където $x_i = a + i \frac{b-a}{N}$. Точността зависи от N .

Алгоритъм: Да се напише програма за коректно въвеждане на математическа функция е твърде сложно и излиза извън рамките на това ръководство. Ще използваме, че компилатора може да провери за коректност един аритметичен израз и ще въвеждаме функцията $f(x)$ направо в кода на програмата.

Програма

```

program integral;
uses WinCrt;
var a, b, s: real;

function f(x: real): real; { Под интегралната функция f(x) }
begin
    f:= ..... { На това място въведете функцията f(x) }
end;

function intergate(a, b: real): real;
const N = 10000;
var
    s, x1, x2: real;
    i: integer;
begin
    x1 := a;
    s := 0;
    for i := 1 to N do begin
        x2 := x1 + (b - a) / N;
        s := s + f((x1 + x2) / 2);
    end;
end;

```

```

        x1 := x2
    end;
    integrate := s
end;

begin
    repeat
        write('a = ');
        readln(a);
        write('b = ');
        readln(b);
    until b > a;
    writeln('Integral = ', integrate(a,b):10:4);
end.

```

Упътване: Пробвайте програмата със следните примерни функции като внимавате интервала: [a, b] да е в дефиниционната област на съответната функция:

```

f := sqrt(x) / sqrt(x - 1)
f := 1 / (x * sqrt(x) * (x + 1))
f := cos(x) / (1 - sin(x))
f := ln(x) / (x * x)

```

Локални и глобални величини. Всички величини, дефинирани вътре в една подпрограма: етикети, константи, нови типове, променливи, подпрограми и нейните формални параметри са локални, защото областта на тяхното действие е само подпрограмата и те нямат смисъл и са непознати извън нея. От друга страна всички величини, дефинирани извън дадена подпрограма в подпрограмата вътре, в която тя е дефинирана, или в главната програма са глобални по отношение на тази подпрограма. Величините дефинирани в главната програма са глобални по отношение на всички подпрограми в тази програма. Глобалните величини могат да действат в подпрограмата, спрямо която те са глобални.

Пример 7.5:

```

program p75;
    var
        a: real;           { a - глобална за процедурата pr }

    procedure pr;
        var
            x, y: real;     { x, y - локални за процедурата pr }
        begin
            .....
            x:= 5; y:= 10;
            a:= x + y;       { a променя стойността си вътре в процедурата pr }
            .....
        end;

```

```

begin
.....
a:= 0;
pr;           { След изпълнение на процедурата pr, a = 15 }
x := a;       { Грешка "Непознат идентификатор", x е локална в pr и
               непозната в главната програма }
.....
end.

```

Когато една глобална променлива си сменя стойността в някаква подпрограма, се проявява така наречения *страничен ефект* при изпълнение на тази подпрограма, променливата не явно си изменя стойността при викане на подпрограмата (в примера променлива a). Когато подпрограмата се пише от един програмист, а се вика в програмни части писани от други програмисти, това може да доведе до трудно откриваемите грешки. Използването на глобални променливи в подпрограмите е оправдано в някои случаи поради големия брой параметри, което води до забавяне на програмата.

Възможно е да има съвпадение на имената на глобални и локални величини. В този случай действа правилото, че локалните имена отменят глобалните, т. е. глобална величина, чието име съвпада с име на локална величина, не действа в подпрограмата с тази локална величина.

Пример 7.6:

```

program pr76;
type
  mas: array[1 .. 5] of real;   { Глобален тип }
var
  a: mas;
  b, x: real;

function fn(mas: real; var b: boolean): real;
var
  c: mas;                       { Грешка, mas е параметър, а не тип }
  x: char;
begin
.....
  x := 'Ж';                     { Локалната променлива x отменя глобалната x }
  b := true;                    { Параметърът b отменя глобалната променлива b }
  a[1] := mas + 2;              { Глобалният масив a се променя във функцията }
.....
end;

```

Глобалните променливи b и x и типа mas не действат вътре във функцията fn.

Задача 7.4: Да се състави програма, която по зададена дата чрез ден, месец и година изчислява, кой ден от годината е тази дата и обратно по зададен ден от годината изчислява съответния месеца и ден от този месец.

Анализ

Входни данни: d - ден, m - месец, day - ден от годината, g - година - цели числа

Ограничения: g - всяко число,

$1 \leq m \leq 12$, $1 \leq d \leq 28, 29, 30$ или 31 според месеца и годината,

$1 \leq \text{day} \leq 365$ или 366 според годината.

Исходни данни: day - ден от годината;

d - ден, m - месец.

Алгоритъм: В двумерен масив a (2 реда и 12 стълба) ще запишем броя на дните в месеците от високосна година (I ред) и обикновена година (II ред). Определянето дали g е високосна става по следния начин:

Ако $(g \bmod 400 = 0)$ или $((g \bmod 100 \neq 0) \text{ и } (g \bmod 4 = 0))$

тогава k <- 1 ; I ред на масива

иначе k <- 2 ; II ред на масива

Определянето на day от d и m:

day <- d

За i <- 1 до m - 1 повтаряй

day <- day + a[k, i]

Определянето на d и m от day

d <- day

m <- 1

Докато d > a[k, m] повтаряй

d <- d - a[k, m]

m <- m + 1

Програма

program dates;

uses WinCrt;

var

a: **array**[1 .. 2, 1 .. 12] **of integer**;

{ a - масив, в който са записани дните от месеците.

I ред - високосна година, II ред - не високосна година }

day, d, m, g: **integer**;

{ day - ден от годината, d - ден от месеца, m - месец, g - година }

b: **boolean**;

function year_day: **integer**;

{ Функция изчисляваща ден от годината по зададени ден, месец и година. }

var

i, k, day: **integer**; { day - локална променлива отменяща съответната глобална }

begin

if $(g \bmod 400 = 0)$ **or** $((g \bmod 100 \neq 0) \text{ and } (g \bmod 4 = 0))$

then k := 1

else k := 2;

```

    day := d;
    for i := 1 to m-1 do
        day := day + a[k, i];
    year_day := day
end;

```

```

procedure day_month;
    { Процедура изчисляваща ден и месец по зададени ден от годината и година }

```

```

var

```

```

    k, i: integer;

```

```

begin

```

```

    if (g mod 400 = 0) or ((g mod 100 <> 0) and (g mod 4 = 0))

```

```

        then k := 1

```

```

        else k := 2;

```

```

    d := day;

```

```

    m := 1;

```

```

    while d > a[k, m] do begin

```

```

        day := day[k, m];

```

```

        m := m + 1

```

```

    end

```

```

end;

```

```

begin

```

```

    a[1, 1]:= 31; a[1, 2]:= 29; a[1, 3]:= 31; a[1, 4]:= 30; a[1, 5]:= 31; a[1, 6]:= 30;

```

```

    a[1, 7]:= 31; a[1, 8]:= 31; a[1, 9]:= 30; a[1, 10]:= 31; a[1, 11]:= 30; a[1, 12]:= 31;

```

```

    a[2, 1]:= 31; a[2, 2]:= 28; a[2, 3]:= 31; a[2, 4]:= 30; a[2, 5]:= 31; a[2, 6]:= 30;

```

```

    a[2, 7]:= 31; a[2, 8]:= 31; a[2, 9]:= 30; a[2, 10]:= 31; a[2, 11]:= 30; a[2, 12]:= 31;

```

```

repeat

```

```

    write('Въведете дата:');

```

```

    readln(d, m, g);

```

```

    b := true;

```

```

    if (m < 1) or (m > 12) or (d < 1) or (d > 31)

```

```

        then b := false

```

```

    else if (m in [4, 6, 9, 11]) and (d > 30)

```

```

        then b := false

```

```

    else if m = 2

```

```

        then if (g mod 400 = 0) or ((g mod 100 <> 0) and (g mod 4 = 0))

```

```

            then begin

```

```

                if d > 29

```

```

                    then b := false

```

```

            end

```

```

        else if d > 28

```

```

            then b := false

```

```

    until b;

```

```

    writeln('Тази дата отговаря на ден от годината:', year_day);

```

```

repeat

```

```

    write('Въведете ден от годината и година:');

```

```

    readln(day, g);

```

```

    b := true;

```

```

    if (day < 1) or (day > 366)

```

```

        then b := false
    else if not ((g mod 400 = 0) or ((g mod 100 <> 0) and (g mod 4 = 0))) and (day >
365)
        then b := false
    until b;
    day_month;
    writeln('Дата:', d, '-', m, '-', g);
end.

```

Забележка: Променливите a, day, d, m и g са глобални и действат в подпрограмите (day се отменя във функцията year_day от локална променлива със същото име).
Логическата операцията **in** проверява дали елемент принадлежи на множество или не.

Предварително деклариране на подпрограма. Когато се вика подпрограма, която е описана по-надолу в програмата, е необходимо тя предварително да се декларира. Иначе компилаторът, който преглежда програмата отгоре надолу, не знае името на подпрограмата, нито броя, типа и вида на нейните параметри. Най-добре първо да си опишем подпрограмата, а след това да я викаме, но това не винаги е възможно - например при [косвена рекурсия](#). Предварителното деклариране включва описание на заглавния ред следвано от думата **forward**. При процедури това изглежда така:

```
procedure име_на_процедура ( списък_от_формални_параметри ); forward;
```

При функции:

```
function име_на_функция (списък_от_формални_параметри): тип; forward;
```

При описанието на тези предварително декларирани процедури и функции в заглавния им ред не се описват формалните параметри, защото те вече са описани.

Пример 7.7:

```

program pr77;
procedure proc(var a, b: real; n: integer); forward; { Предварително деклариране }
function func(var z: real; k: integer): real; forward; { Предварително деклариране }

procedure call;
var
    x, y, z: real;
begin
    z := func(x, 10) / 5.3;      { Викане на func преди да е описана. }
    proc(y, z, 15);             { Викане на proc преди да е описана. }
end;

procedure proc; { Описание - не се описват формалните параметри }
    .....
begin
    { Описание на proc }
    .....
end;

```

```

function func; { Описание - не се описват формалните параметри и типа }
.....
begin
    { Описание на func }
    .....
end;

```

Библиотеки от готови подпрограми. Както бе казано в първа част, фирмата доставчик на средата за програмиране, а също така и специализирани фирми предлагат библиотеки от готови подпрограми, които могат да се ползват наготово в програмата. Тези библиотеки се поставят в специалната папка за библиотеки (обикновено `units`) на средата. С изключение на библиотеката `system` ако искаме да ползваме подпрограми от тези библиотеки, трябва да ги декларираме в началото на програмата след заглавния ред по следния начин:

```

uses библиотека1, библиотека2, .. , библиотекаN;

```

т. е след **uses** в списък се изброяват имената на ползваните библиотеки. Тези имена са идентификатори и се взимат от документацията на съответната библиотека. Описанието на библиотеките на фирмата разработила средата за програмиране се намира в помощната система (`help`).

Програмистите могат сами да разработват библиотеки от готови подпрограми и да ги включват в средата за програмиране и така да я обогатяват.

Ето някои библиотеки от Borland Pascal v.7.0:

- `system` - тази библиотека няма нужда да се декларира. Тя включва всички основни входно-изходни подпрограми като например `write`, `writeln`, `read`, `readln`; подпрограмите за работа с файлове, подпрограмите за работа със символна информация, основни математически функции.
- `graph` - графична библиотека, включваща подпрограми за изчертаване на основните графични примитиви, като точка, отсечка, кръг и др., работа с цветове, прозорци, шрифтове. Може да се използва само операционната система MS DOS.
- `graph3`, `turbo3` - библиотеки с подпрограми за Turbo Pascal v.3. Те са включени за съвместимост с по-стари програми.
- `crt` - включва подпрограми за работа с прозорци и цвят в текстови режим; за по-професионално използване на клавиатурата и говорителя Може да се използва само операционната система MS DOS или команден режим на MS Windows.
- `overlay` - за *овърлейни програми*. Това са много големи програми не можещи да се съберат в оперативната памет на компютъра и затова са разделени на части - овърлеи. В даден момент от изпълнението на такава програма в паметта се намира основната част и този овърлей подпрограми, от който се изпълняват.
- `printer` - включва подпрограми за управлението на принтера.
- `dos` - включва подпрограми за връзка с операционната система DOS.
- `WinCrt` - използва се при програми работещи в операционната система MS Windows. В нея се намират функции `write`, `writeln`, `read` и `readln`, които са алтернативни на тези от библиотеката `system`.
- `WinAPI` - съдържа подпрограмите на стандарта API за MS Windows v.3.1.

Някои по-често използвани математически подпрограми от библиотеката system:

function abs(x) - абсолютна стойност на x.

function arctan(x: **real**): **real** - резултатът е в радиани.

function cos(x: **real**): **real** - параметърът е в радиани.

procedure dec(var k) - k е от изброим тип и се намалява с 1.

function exp(x: **real**): **real** - e^x .

procedure exit - прекъсва блокът, в който се намира, ако блокът е тяло на цикъл, прекъсва се цикъла, ако е в тяло на подпрограма или на програмата, прекъсва се подпрограмата или програмата.

function frac(x: **real**): **real** - дава дробната част на x.

procedure halt - прекъсва програмата.

procedure inc(var k) - k е от изброим тип и се увеличава с 1.

function int(x: **real**): **real** - дава цялата част на x.

function ln(x: **real**): **real** - натурален логаритъм.

function pi: **real** - числото π .

procedure randomize - стартира генератора на случайни числа.

function random (k: **integer**) - намира случайно число. Тя може да се ползва без параметър и тогава резултатът е реално число в интервала $[0, 1]$. Ако се ползва параметър $k > 0$, резултатът е цяло число в интервала $[0, k]$.

function round(x: **real**): **integer** - закръгля x до цяло.

function sin(x: **real**): **real**; - параметърът е в радиани.

function sqr(x) - x^2 .

function sqrt(x: **real**): **real** - корен квадратен от x.

function trunc(x: **real**): **integer** - отрязва x до цяло.

Рекурсия.

Дефиниция 7.2

Случаят, когато една подпрограма вика себе си, се нарича *рекурсия*. Рекурсията бива *пряка* и *косвена*.

Рекурсията е пряка, ако в тялото на подпрограмата има обръщение към нея. Косвена е тази рекурсия, когато една подпрограма вика друга, а тя вика първата. Може при косвена рекурсия една подпрограма да вика себе си чак след цяла верига от обръщения към други подпрограми, т. е. първата да вика втората, тя да вика трета, тя пък четвърта и т. н. и накрая последната да вика първата.

Когато една рекурсивна подпрограма започне да се изпълнява, то се достига пряко или косвено до обръщението към себе си. Тогава тя отново започва да се изпълнява и пак достига да това обръщение и т. н., т. е. всъщност рекурсията е един не явен цикъл. Като при всеки цикъл трябва да съществува изход от този цикъл, иначе ще се получи зацикляне на програмата. В рекурсивната подпрограма би трябвало да има случай, когато тя не вика себе си. Когато се изпълни този случай, се казва, че се излиза от рекурсията. Броят пъти, когато една рекурсивна подпрограма вика себе си до излизането от рекурсията се нарича *дълбочина на рекурсията*. Програмистът трябва да следи тази дълбочина да не е твърде голяма, защото това води до препълване на служебната памет на програмата (програмния стек).

Всяка една програма с рекурсивна подпрограма може да се напише и без рекурсия чрез цикъл. В повечето случаи това е за предпочитане, защото цикълът се изпълнява по бързо от рекурсията и няма ограничения в дълбочината. Съществуват обаче задачи, където прилагането на рекурсия води до елегантно и лесно решение и за които намирането на решение чрез цикъл никак не е толкова лесно.

Задача 7.5: Да се състави рекурсивна функция за изчисляване на функцията на Акерман

$$Ack(m,n) = \begin{cases} n + 1, m = 0 \\ Ack(m - 1, 1), n = 0 \\ Ack(m - 1, Ack(m, n - 1)), m > 0, n > 0 \end{cases}$$

където m и n са цели неотрицателни числа.

Анализ

Параметри: m и n - цели и $m, n \geq 0$.

Стойност на функцията - цяла.

Алгоритъм: Той е рекурсивен и точно следва дефиницията на функцията на Акерман.

Функция

```
function ack(n, m:integer): integer;
begin
    if (m < 0) or (n < 0)           { Грешни входни параметри }
    then begin
        ack := 0;
        exit      { Библиотечна процедура за прекъсване на подпрограма }
    end;
    if m = 0
    then ack := n + 1               { Изход от рекурсията }
    else if n = 0
    then ack := ack(m - 1, 1)       { Рекурсия }
    else ack := ack(m - 1, ack(m, n - 1)) { Двойна рекурсия }
end;
```

Задача 7.6: Да се състави програма реализираща играта "Ханойски кули". Играта е следната: Имате три стълба А, В, С и N диска с различни диаметри с отвор в центъра си, така че да могат да се надяват на стълбовете. Дисковете са наредени на стълб А в следния ред: най-отдолу е диска с най-голям диаметър, върху него е този с по-малък диаметър и т. н. Целта на играта е дисковете да се наредят в същия ред на стълб В, като се спазват следните правила: на едно местене може да се взема само един диск от тези най-отгоре на някой от стълбовете и не може да се поставя диск с по-голям диаметър върху диск с по-малък.

Анализ

Входни данни: $N > 0$ - цяло - брой дискове.

Изходни данни: поредица от команди към играча, кой диск да премести и къде.

Алгоритъм

Ако броя дискове е 1
тогава го преместваме от стълб А на стълб В.
Иначе допускаме, че можем да решим задачата за N-1 диска.
Товага преместваме горните N-1 диска от стълб А на стълб С.
Преместваме останалия N-ти диск от А на В.
Преместваме N-1 диска от С на А.

Забележка. Именно допускането, че можем да решим задачата за N-1 диска означава, че имаме рекурсия.

Програма

```
program hanoi;
uses WinCrt;
var
  n: integer;      { Брой дискове }
  a, b, c: char;   { Имена на стълбовете }

procedure kula(a, b, c: char; n: integer);
begin
  if n = 1        { Изход от рекурсията }
  then writeln('Премести диск 1 от стълб ', a, ' на стълб ', b)
  else begin
    kula(a, c, b, n-1); { Рекурсия }
    writeln('Премести диск ', n, ' от стълб ', a, ' на стълб ', b);
    kula(c, b, a, n-1)  { Рекурсия }
  end
end;

begin
  repeat
    writeln('Въведете брой дискове:');
    readln(n)
  until n > 0;
  a := 'A'; b := 'B'; c := 'C';
  kula(a, b, c, n)
end.
```

Съдържание

6. Деклариране на нови типове в Pascal. Тип диапазон. Масиви: едномерни, двумерни и тримерни масиви. Индексиране на масив. Обхождане на масив.

8. Обработка на символна информация в Pascal. Символен тип и тип низ от символи. ASCII таблица в DOS и ANSI таблица в Windows. Операции с отделни символи и с низове от символи. Вградени подпрограми в Borland Pascal за работа със символи.

8. Обработка на символна информация в Pascal. Символен тип и тип низ от символи. ASCII таблица в DOS. Операции с отделни символи и с низове от символи. Вградени подпрограми в Borland Pascal за работа със символи.

Освен за програми обработващи числени данни езикът Pascal може да се използва и за обработка на символни данни. В Pascal символните данни са три вида: отделни символи, низ от символи и текстов файл. Тук се разглежда работа с първите два вида символни данни.

- [Отделни символи](#)
- [Символен тип](#)
- [Тип низ от символи](#)

Отделни символи в паметта на компютъра обикновено се кодират в един байт. Поради това те са 256 и на всеки символ съответна число - неговия код. В операционната система DOS тези символи са подредени в ASCII таблица. Тя включва малките и големите латински букви, цифрите, малките гръцки букви, препинателни знаци, псевдо-графични символи за изчертаване на таблици и блокове, управляващи знаци, букви с ударения използвани в разлени азбуки и др. Някои от управляващите символи са:

- нов ред (Line Feed) - код 10,
- начало на ред (Carriage Return) - код 13,
- нова страница на принтера (Form Feed) - код 12,
- табулация (Tab) - код 9,
- символ назад (Back Space) - код 8,
- звуков сигнал (Bell) - код 7.

Символите, които не са отбелязани на клавиатурата, могат да се въведат, като се държи натиснат клавиша Alt и чрез цифрите в дясната част на клавиатурата се набере трицифрения код на символа.

В Windows символите зависят от шрифта с който се работи. Шрифтовете обикновено са подредени в ANSI таблица. Напоследък се използват разширени двубайтови шрифтове и тогава броят на символите е около 65000.

Оригиналната ASCII таблица и много от шрифтовете в Windows не съдържат кирилица. В ASCII таблицата тя се добавя чрез специална кирилизираща програма и обикновено започва от код 128. Тя се състои от общо 64 букви (големи и малки). Те заместват буквите с ударения и част от псевдо-графичните символи в оригиналната таблица. Съществуват обади други наредби на кирилицата както в DOS, така и в шрифтовете за

Windows. Затова препоръчително е в програмата да се работи със самите символи, а не с кодовете на символите.

Символен тип. Това е вграден в Pascal тип **char**. На променливи от този тип в програмата може да се присвояват стойности чрез оператор за присвояване или чрез вход от клавиатурата. Символните константи, за които бе споменато във втора точка се задават чрез символа заграден с апострофи (напр. 'ж'). Символът апостроф се задава чрез четири символа апостроф по следния начин - "".

Внимание: На клавиатурата има два символа апостроф: ляво наклонен и дясно наклонен. В Pascal се използва дясно наклонения, който е на един и същи клавиш с кавичката.

Символните данни могат да участват в операциите за сравнение. При това се извършва сравняване на кодовете на символите. Символният тип е изброим тип. В Pascal съществуват следните библиотечни функции за работа със символи

function chr(kod: **integer**): **char** - преобразува число - код на символ в самия символ.
function ord(ch: **char**): **integer** - преобразува символ в съответното му число - код на символа.
function pred(ch: **char**): **char** - намира предходния символ в таблицата.
function succ(ch: **char**): **char** - намира следващия символ в таблицата.

Забележка: Всъщност последните три функции са за всякакви изброими типове.

Пример 8.1:

```
var
  ch: char;
begin
  .....
  ch := 'Ж';           { Присвояване на символ константа }
  readln(ch);          { Въвеждане на символ от клавиатурата }
  if ch < 'И'           { Операция сравняване }
  then write(chr(7));   { Звуков сигнал }
  .....
  { Преобразуване на големи букви кирилица в малки кирилица }
  if (ch >= 'А') and (ch <= 'Я')
  then ch := chr(ord('a') + ord(ch) - ord('A'));
  .....
end.
```

Задача 8.1: Да се изведат на екрана всичките 256 символи.

Анализ

Входни данни: няма.

Изходни данни: 256 символа от кодовата таблица. Част от тях не могат да се видят (напр. управляващите).

Алгоритъм

Начало
За i <- 0 до 255 повтаряй
Изведи chr(i)
Край

Програма

```
program ascii;  
uses WinCrt;  
var i: integer;  
begin  
  for i := 0 to 255 do begin  
    write('Код ', i, '->символ ', chr( i ));  
    readln  
  end  
end.
```

Тип низ от символи. Низ от символи наричаме крайна наредена съвкупност от символи. Константен низ от символи (напр. 'Решение:'), за който бе споменато във втора точка, представлява подредени един след друг символи, заградени с апострофи. Ако в низа има символа апостроф, той се пише два пъти (напр. 'Пролет"2000'). За тип низ от символи в Pascal има вграден тип **string**. Декларирането на променливи от този тип става по два начина:

```
име: string;  
име: string[максимален_брой_символи];
```

При първия начин се подразбира, че максималният брой символи на низа е 255. При втория начин на деклариране максимален_брой_символи трябва да е цяла константа от 1 до 255. Когато се декларира параметри на подпрограми от тип низ от символи се ползва само първия начин на деклариране. От такъв тип са и функциите със стойност низ от символи.

Пример 8.2:

```
var  
  s: string;           { Максимална дължина 255 символа }  
  s1: string[5];       { Максимална дължина 5 символа }  
  
function f1(a: string; var b: string): string;  
.....  
begin  
  .....  
end;
```

Данните от тип низ от символи заемат в паметта толкова байта, колкото е тяхната максимална дължина плюс един.

Променливите от тип низ от символи могат да получават стойности чрез оператор за присвояване или чрез процедура за вход от клавиатурата. Стойността, която

променливата може да приемат, е низ с дължина от 0 символа до максималната дължина на променливата. Низ с дължина 0 се нарича празен низ и се бележи с "".

Пример 8.3 (виж пример 8.2):

```
s := 'Иван';           { Дължината на s е 4 символа }
writeln('Име:', s:10); { Извеждане на низ. s се извежда в 10 позиции }
s1 := '';              { На s1 се присвоява празния низ. Дължината е 0 символа }
readln(s);             { Въвеждане на низ от клавиатурата }
s1 := 'Николай';       { Грешка! Низът 'Николай' е дълъг 7 символа. s1 се препълва. }
```

Низове могат да участват в операции за сравнение. Сравнението на два низа става по лексикографски начин т.е. първо се сравняват първите символи на двата низа, след това вторите и т.н. Всъщност се сравняват кодовете на съответните символи. Ако един низ изцяло се съдържа в началото на друг, то той е по-малък от него.

Пример 8.4:

```
'Иван' < 'Петър'      = true
'Ана' < 'Анатолий'    = true
'Ана' < 'ана'         = true { главно 'А' е преди малко 'а' }
```

В Pascal съществува специална операция наречена *конкатенация* за слепване на низове. Конкатенация на два низа се задава чрез знака "+". Тази операция не е еквивалентна на събиране защото не е асоциативна ('Иван' + 'Петров' ≠ 'Петров' + 'Иван'). Трябва да се внимава при слепване резултантния низ да не е по-дълъг от максималната дължина.

Пример 8.5:

```
s := 'Иван'+' '+'Петров'; { Резултат: 'Иван Петров' }
```

Обръщението към отделните символи в един низ става чрез индекс. Първият символ е с индекс 1, вторият - с индекс 2 и т.н.

Пример 8.6:

```
s := 'Петров';
s[4] := 'к'; { Резултат s = 'Петков' }
```

Променливите низове всъщност имат и символ с индекс 0. Кодът на този символ съответства на дължината на низа в съответната променлива. Например, ако s = 'Иван Петров', то ord(s[0]) = 11.

В Borland Pascal има няколко библиотечни подпрограми за работа с низове.

function length(s: **string**): **string** - определя дължината на низа s. Еквивалентна на ord(s[0]).

function concat(s1, s2, ..., sN: **string**): **string** - извършва операцията конкатенация
Съществува за съвместимост с програми написани на по-старите версии на Pascal,
където тази операция не се е поддържала.

function copy(s: **string**; index, broj: **integer**): **string** - взема под низ на низа s.
Резултантния подниз започва от символ с индекс index и е с дължина broj.

procedure insert(subs: **string**; var s: **string**; index: **integer**) - вмъква низа subs вътре в низа s на позиция index. В резултат на това s става по-дълъг.

procedure delete(var s: **string**; index, broj: **integer**) - изтрива от низа s broj на брой символи от символа с индекс index нататък. В резултат на това s става по-къс.

function pos(subs, s: **string**): **integer** - търси отляво на дясно дали низът subs е подниз на низа s. Ако не е, резултатът е 0, иначе функцията връща индекса на първия символ на подниза subs в низа s. Например pos('Иван', 'Иво Иванов Иванов') = 5.

procedure str(x: числов; var s: **string**) - преобразува числото x в низа s. Към числото може да се приложат форматиращите модификатори за брой позиции и точност подобно на процедурите write и writeln за изход на екрана. Например, ако str(-3.14;6:3, s), тогава s = '-3.140'.

procedure val(s: **string**; var x: числов; var code: **integer**) - преобразува низа s в числото x, ако е възможно. В s трябва да има низ изобразяващ правилно записано цяло или реално число. Допуска се и научен формат за запис на реалното число. При правилно преобразуване параметърът code = 0, иначе code дава индекса на този символ в s, който е предизвикал грешката.

Задача 8.2: Да се състави функция, която изчислява броя на думите в един низ.

Анализ

Входен параметър: s - низ от символи.

Резултат: n - цяло - броя на думите.

Алгоритъм: За да се броят думите, трябва да се определи, кои са разделителите между думите. Ще считаме че думите се отделят една от друга чрез: интервал, табулация, запетая. Лесно могат да се добавят и други разделители. Всички останали символи са букви от думите.

Функция

function brojdumi(s: **string**): **integer**;

var

i, n: **integer**;

inw: **boolean**; { Индикатор, дали сме вътре или вън от дума }

begin

inw := **false**;

n := 0;

for i := 1 **to** length(s) **do**

if (s[i] = ' ') **or** (s[i] = chr(9)) **or** (s[i] = ',') { Разделители между думи }

```

    then begin
        if inw                                { За i - 1 сме били вътре в дума и сега излизаме }
        then inw := false
    end
    else if not inw                            { Влизаме в дума }
    then begin
        n := n + 1;
        inw := true
    end;
    brojdumi := n
end;

```

Задача 7.3: Да се състави процедура за заместване на един подниз на даден низ с друг подниз.

Анализ

Входни параметри: s1 - заместващ подниз, $\text{length}(s1) > 0$;
s2 - заместваем подниз, $s2 \neq s1$;
s - низ, в който се извършва заместването.

Резултат: s.

Алгоритъм:

Повтаряй
Търси s1 в s
Ако s1 е намерен
Изтрий толкова символи в s, колкото е дължината на s1
Вмъкни s2 на мястото на s1
докато s1 се среща в s

Процедура

```

procedure replace(s1, s2: string; var s: string);
var k: integer;
begin
    if (length(s1) > 0) and (s2 <> s1)
    then repeat
        k := pos(s1, s);
        if k > 0
        then begin
            delete(s, k, length(s1));
            insert(s2, s, k)
        end
    until k > 0
end;

```

Съдържание

7. Подпрограми. Видове подпрограми в Pascal. Параметри на подпрограмите. Подпрограма процедура. Подпрограма функция. Локални и глобални величини в програмите. Предварително деклариране на подпрограма. Ползване на готови библиотеки от подпрограми. Рекурсия.

9. Записи. Деклариране и използване на записи. Запис - поле на друг запис. Вариантни записи. Оператор with.

9. Записи. Деклариране и използване на записи. Запис - поле на друг запис. Вариантни записи. Оператор with.

- [Записи](#)
- [Вложени записи](#)
- [Вариаантни записи](#)
- [Оператор with](#)

Дефиниция 9.1 *Запис* е крайна наредена структура от данни, състояща се от разнотипни елементи, наречени *полета*.

Записът служи да групира в себе си разнотипната информация, описваща някакъв сложен информационен обект. Например за студент може да имаме следните данни: име - низ от символи, факултетен номер - цяло число, среден успех - реално число и т.н.

Когато в програмата трябва да се използват записи, обикновено се декларира тип - запис и после се декларират променливи, параметри на подпрограми, файлове от такъв тип - запис. Декларирането на тип - запис става в секцията **type** на програмата по следния начин:

```
име = record  
    поле1: тип1;  
    поле2: тип2;  
    .....  
    полеN: типN  
end;
```

където име е името на новия тип, поле1, поле2, ..., полеN са имената на полетата, от които се състои записа, а тип1, тип2, ..., типN са съответно техните типове. Може направо да се декларират променливи - записи без предварително да се декларира съответен тип запис.

Пример 9.1:

```

type
  student = record
    name: string[50];      { поле 1 - име на студент }
    fn: longint;          { поле 2 - факултетен номер - дълго цяло число }
    estim: real;          { поле 3 - оценка }
  end;
var
  s: student;              { променлива запис }
  polygon: record         { променлива запис }
    strana: real;
    broj: integer
  end;
  group: array[1 .. 20] of student; { масив от записи }

procedure vghod(var s:student); { параметър запис }
  .....

```

Променливите и параметрите записи се ползват в програмата чрез обръщение към техните полета по следния начин:

име.поле

където име е името на променливата или параметъра запис, а поле е името на полето, към което се обръщаме. Обикновено се ползват отделните полета на записа. С тях се работи като с величини от съответния тип. Цели записи могат единствено да се присвояват един на друг чрез оператор за присвояване и да се четат или записват във файлове от съответния тип записи.

Пример 9.2 (виж пример 9.1):

```

begin
  write('Име на студент:');      { въвеждане на полетата на запис }
  readln(s.name);
  write('Факултетен номер:');
  readln(s.fn);
  write('Среден успех:');
  readln(s.estim);
  if s.name = 'Иван Петров Иванов'
    then group[1] := s;          { присвояване на един запис - друг запис }
  .....
  for i: = 1 to 20 do begin
    writeln('Име ', group[i].name); { обръщение към поле от запис - елемент на масив }
  }

  writeln('Факултетен номер ', group[i].fn);
  writeln('Среден успех ', group[i].estim:5:1);
end;
.....
write('Брой страни на многоъгълник:');
readln(polygon.broj);
write('Дължина на страна:');

```

```

        readln(polygon.strana);
        .....
    end.

```

Полетата на един запис могат да са не само скаларни данни, но и структури масиви, низове, записи. В следващия пример имаме вложени записи.

Пример 9.3:

```

type
    factura = record
        nomer: integer;
        cena: real;
        ime: string[60];
    end;
    detail = record
        ime: string[30];
        broj: integer;
        dostavka: factura           { поле - запис }
    end;
var
    sklad: array[1 .. 100] of detail;
begin
    for i := 1 to 100 do begin
        write('Име на детайл:');
        readln(sklad[i].ime);
        write('Брой:');
        readln(sklad[i].broj);
        write('Номер на фактура:');
        readln(sklad[i].dostavka.nomer); { обръщение към подполе на поле - запис }
        write('Цена:');
        readln(sklad[i].dostavka.cena);
        write('Име на доставчик:');
        readln(sklad[i].dostavka.ime)
    end;
    .....

```

В горния пример записът detail има поле dostavka, което е също запис, т.е. имаме вложени записи. До полетата на вложения запис достигахме като приложим два пъти знака точка. Обърнете внимание, че sklad[i].ime и sklad[i].dostavka.ime са две различни полета.

Задача 9.1: Да се състави програма, която изпълнява следните дейности:

- в масив записва следната информация за студенти: име, факултетен номер, среден успех;
- извежда на екрана информация за всички студенти в масива;
- променя успеха на студент по зададен факултетен номер.

Анализ

Входни данни: k - цяло - вид дейност на програмата:

- 1 - добавяне на студент към масива,
- 2 - информация за всички студенти,
- 3 - промяна на успеха,
- 0 - изход.

При въвеждане информация за студент:

- name - низ от символи - име на студент,
- fn - дълго цяло - факултетен номер,
- estim - реално - факултетен номер.

При промяна успеха на студент:

- fn - факултетен номер,
- estim - успех.

Изходни данни: Меню с дейностите на програмата, полетата на елементите-записи в масива, различни съобщения: за грешен избор от менюто, за липса на студенти в масива и т.н.

Алгоритъм: Програмата ще работи циклично, като ще завърши при избор за изход от менюто (k = 0). При всяко завъртане в този цикъл ще изпълнява една от своите дейности според избора от менюто.

Програма

```
program student_info;
uses WinCrt;
const max = 100;
type
  student = record
    name: string[50];
    fn: longint;
    estim: real
  end;
group = array[1 .. max] of student;
var
  k, n: integer;      { k - избор от менюто, n -брой студенти в масива }
  g: group;           { масив от студенти }

procedure dobavi(var g: group; var n: integer); { Добавяне на студент към масива }
begin
  if n = max
  then writeln('Масивът със студенти е пълен !')
  else begin
    n:= n + 1;
    write('Име на студент:');
    readln(g[n].name);
    write('Факултетен номер:');
    readln(g[n].fn);
    write('Среден успех:');
    readln(g[n].estim);
```

```

        end
    end;

    procedure info(g: group; n: integer); { Извеждане на информация за всички студенти }
}
    var i: integer;
    begin
        if n = 0
        then writeln('Масивът със студенти е празен !')
        else
            for i := 1 to n do begin
                writeln('Име на студент:', g[i].name);
                writeln('Факултетен номер:', g[i].fn);
                writeln('Среден успех:', g[i].estim:5:2);
                readln
            end
        end;
    end;

    function change(var g: group; n: integer): boolean; { Промяна на ср. успех на студент }
}
    var
        i: integer;
        fn: longint; { факултетен номер на търсения студент }
        find: boolean; { показва дали е открит студент с указания факултетен номер }
    begin
        find:= false;
        write('Факултетен номер на търсения студент:');
        readln(fn);
        for i := 1 to n do
            if fn = g[i].fn
            then begin
                find:= true;
                writeln('Име на студент:', g[i].name);
                writeln('Стар среден успех:', g[i].estim:5:2);
                write('Нов среден успех:');
                readln(g[i].estim)
            end;
        change := find
    end;

begin { Главна програма }
    n:= 0; { няма студенти в масива }
    repeat
        writeln('Меню');
        writeln('1. Добавяне на студент');
        writeln('2. Информация за всички студенти');
        writeln('3. Промяна на среден успех');
        writeln('0. Изход');
        write('Изберете:');
        readln(k);

```

```

case k of
  0: ;      { празен оператор }
  1: dobavi(g, n);
  2: info(g, n);
  3: if not change(g, n)
      then writeln('Няма такъв студент !');
      else writeln('Грешен избор !')
  end
until k = 0
end.

```

Забележка: Програми от този вид се наричат информационни системи.

Информационните системи се характеризират с това, че притежават архив (в този случай масива g) и имат средства за допълване (дейност 1) и промяна на единиците на архива (дейност 3), а също така могат да извършват справки в архива (дейност 3).

Понякога в един запис в едни случаи трябва да има едни полета, а в други случай други. Тогава се използват *вариантни записи*. Например в записа student (виж пример 9.1) полето estim служи за среден успех на студента. Възможно е обаче студента да е прекъснал и да няма среден успех. В този случай би трябвало да има друго поле от низ от символи, в което да се посочи причини за прекъсване.

Всички вариантни полета се декларират след постоянните полета на записа по следния начин:

```

име = record
  постоянни полета
  case ключ: изброим_тип of
    случай1: (полета1);
    случай2: (полета2);
    .....
    случайN: (полетаN)
  end;

```

където постоянните полета се декларират като при обикновените записи, полето ключ определя, кой от вариантите на записа се използва, случай1, случай2, ..., случайN са стойности (константи) на полето ключ, указващи при каква стойност на ключ какви вариантни полета се избират и полета1, полета2, ..., полетаN са полетата за съответния вариант. Може сред някои от тези полета също да имаме вариант и така се получават сложни вложени варианти.

Пример 9.4:

```

type
  student = record
    ime: string[50];
    fn: longint;
    case izpit: boolean of           { ключово поле показващо дали студента е
изпитан }
      true: ( lektor: string[50];   { име на преподавателя }

```

```

        estim: real );           { оценка }
    false: ( note: string[100]) { бележка обясняваща неявяване на изпит }
end;
var
    s1, s2: student;
begin
    s1.ime := 'Иван Петров Иванов';    { I студент }
    s1.fn := 992212;
    s1.izpit := true;
    s1.lektor := 'доц. Петров';
    s1.estim := 3.5;
    s2.ime := 'Ана Иванова Петрова';   { II студент }
    s2.fn := 992213;
    s2.izpit := false;
    s2.note := 'в майчинство';
    .....
end.

```

Подредбата на полетата в записа има значение само при представянето му в паметта на компютъра. Полетата се разполагат в паметта по реда на тяхното деклариране. Размерът на данна от тип запис е равен на сумата от дължините на полетата, от които тя се състои. При вариантите записи се взема дължината на най-дългия вариант. Ако в програмата трябва да определите дължината на един запис в брой байтове, може да се използва следната библиотечна функция:

function sizeof(тип): **integer**;

където параметърът тип е някой от декларираните в секция **type** типове или някой от вградените в езика типове. Например sizeof(**longint**) = 4, а за student от пример 9.4 sizeof(student) = 157.

Ако разгледате отново примери 9.2 и 9.3, може да забележите, че при обръщение към полетата на един запис трябва да се пише всеки път и името на записа. В пример 9.3 пък при обръщение към подполетата на полето *dostavka* трябва да се пише всеки път името на записа и името на полето, към чието подполета става обръщението. В дадена програмна част това може да бъде многократно и да доведе до досадно преписване на едни и същи неща. За този случай в езика Pascal съществува оператор **with**, чрез който се съкращава това досадно преписване. Този оператор има следния вид:

```

with име_на запис do
    команда

```

където име_на запис е името на записа, към чието полета се обръщаме, а в команда, която почти винаги е съставния оператор, се обръщаме направо към тези полета без да пишем името на записа. име_на_запис може да бъде обръщение към поле-запис. Тогава в команда се обръщаме към подполетата без да пишем името на записа и на неговото поле-запис

Пример 9.5 (извеждането на масива group в края на пример 9.2 ползвайки оператор **with**):

```

for i:= 1 to 20 do
  with group[i] do begin
    writeln('Име: ', name);
    writeln('Факултетен номер: ', .fn);
    writeln('Среден успех: ', estim:5:2)
  end;

```

Задача 9.2: Да се създаде информационна система, в която има следната информация за автомобили от транспортно предприятие: марка автомобил, номер, брой километри и състояние на автомобила, което е:

1. на път - краен пункт, разстояние, шофьор;
2. в ремонт - вид ремонт, цена;
3. в престой - брой дни.

Системата да може да извършва следните дейности: добавяне на нов автомобил, премахване на автомобил, пълна справка за всички автомобили, справка за автомобили с пробег в даден диапазон.

Алгоритъм: Тази задача е подобна на задача 9.1. Ще използваме запис с варианти за състоянието на автомобила - 1 на път - три полета, 2 в ремонт - две полета и 3 в престой - едно поле. При изтриването на автомобил от масива:

1. Открива се автомобила с номер еднакъв с този, който искаме да изтрием.
2. Ако такъв автомобил е открит и той не е последния, всички автомобили след него ги премества в масива един елемент напред.
3. Ако автомобил е изтрит, намаляваме броя на автомобилите с един.

Програма

```

program transport;
uses WinCrt;
const max = 50;
type
  automobil = record
    marka: string[20];
    nomer: string[15];
    probeg: longint;
    case sastojanie: integer of
      1:(punkt: string[20];
        razstojanie: integer;
        shofior: string[20]);
      2:(vid: string[20];
        cena: real);
      3:(brojdni: integer);
  end;
  autopark = array[1 .. max] of automobil;
var

```

```
k, n: integer;  
auto: autopark;
```

```
procedure dobavi(var a: autopark; var n: integer);  
begin  
  if n = max  
  then writeln('Автопаркът е пълен !')  
  else begin  
    n := n + 1;  
    with a[n] do begin  
      write('Марка на автомобила:');  
      readln(marka);  
      write('Номер:');  
      readln(nomer);  
      write('Пробег в км:');  
      readln(probег);  
      repeat  
        write('Състояние на автомобила 1. На път, 2. В ремонт, 3. В престой');  
        readln(sastojanie);  
        case sastojanie of  
          1: begin  
            write('Краен пункт:');  
            readln(punkt);  
            write('Разстояние в км:');  
            readln(razstojanie);  
            write('Шофьор:');  
            readln(shofjor)  
          end;  
          2: begin  
            write('Вид на ремонта:');  
            readln(vid);  
            write('Цена в лв:');  
            readln(cena)  
          end;  
          3: begin  
            write('Брой дни на престоя:');  
            readln(brojdni)  
          end;  
        else writeln('Грешен избор на състояние !')  
        end  
      until (0 < sastojanie) and (sastojanie < 4);  
    end  
  end  
end;  
end;
```

```
procedure delauto(var a: autopark; var n: integer);  
var  
  i: integer;  
  numb: string[15];  
  find: boolean;
```

```

begin
  if n = 0
  then writeln('Няма автомобили в автопарка !')
  else begin
    write('Номер на автомобил за изтриване:');
    readln(numb);
    find := false;
    for i := 1 to n do
      if find and (i < n)
      then a[i] := a[i + 1]
      else if numb = a[i].nomer      {автомобилът е открит}
      then begin
        find := true;
        if i < n
        then a[i] := a[i + 1]
        end;
      if find
      then n := n - 1
      else writeln('Няма автомобил с такъв номер !')
    end
  end;
end;

procedure info(a:autopark; n: integer);
var i: integer;
begin
  if n = 0
  then writeln('Няма автомобили в автопарка !')
  else
    for i := 1 to n do begin
      with a[i] do begin
        writeln('Марка на автомобила:', marka);
        writeln('Номер:', nomer);
        writeln('Пробег в км:', probeg);
        write('Състояние на автомобила - ');
        case sastojanie of
          1: begin
            writeln('на път. ');
            writeln('Краен пункт:', punkt);
            writeln('Разстояние в км:', razstojanie);
            writeln('Шофьор:', shofjor)
          end;
          2: begin
            writeln('в ремонт. ');
            writeln('Вид на ремонта:', vid);
            writeln('Цена в лв:', cena:10:2)
          end;
          3: begin
            writeln('в престой. ');
            writeln('Брой дни на престоя:', brojdni)
          end
        end
      end
    end
  end;
end

```

```

        end
    end
    readln
end
end;

```

```

procedure infoprobeg(a:autopark; n: integer);

```

```

    var

```

```

        i, pmn, pmx: integer;

```

```

        find: boolean;

```

```

    begin

```

```

        if n = 0

```

```

            then writeln('Няма автомобили в автопарка !')

```

```

        else begin

```

```

            repeat

```

```

                write('Въведете минимален и максимален пробег:');

```

```

                readln(pmn, pmx)

```

```

            until pmn < pmx;

```

```

            find := false;

```

```

            for i := 1 to n do begin

```

```

                with a[i] do begin

```

```

                    if (pmn <= probeg and (probeg <= pmx)

```

```

                        then begin

```

```

                            find := true

```

```

                            writeln('Марка на автомобила:', marka);

```

```

                            writeln('Номер:', nomer);

```

```

                            writeln('Пробег в км:', probeg);

```

```

                        end

```

```

                    end;

```

```

                readln

```

```

            end;

```

```

            if not find

```

```

                then writeln('Няма автомобили с пробег в указания диапазон !')

```

```

            end

```

```

        end;

```

```

begin                                { Главна програма }

```

```

    n:= 0;                             { няма автомобили в масива }

```

```

    repeat

```

```

        writeln('Меню');

```

```

        writeln('1. Добавяне на автомобил');

```

```

        writeln('2. Информация за всички автомобили');

```

```

        writeln('3. Изтриване на автомобил');

```

```

        writeln('4. Информация за автомобили в зададен диапазон на техния пробег');

```

```

        writeln('0. Изход');

```

```

        write('Изберете:');

```

```

        readln(k);

```

```

        case k of

```

```

            0: ;                        { празен оператор }

```

```

            1: dobavi(auto,n);

```

```
2: info(auto,n);
3: delauto(auto,n);
4: infoprobeg(auto,n);
  else writeln('Грешен избор !')
end
until k = 0
end.
```

Съдържание

8. Обработка на символна информация в Pascal. Символен тип и тип низ от символи. ASCII таблица в DOS и ANSI таблица в Windows. Операции с отделни символи и с низове от символи. Вградени подпрограми в Borland Pascal за работа със символи.

10. Файлове в Pascal. Файлове от определен тип. Достъп до данните във файла. Процедури и функции в Borland Pascal за работа с файлове от определен тип.
