ALLAHVERDIYEVA NAILƏ
NAMAZOV MANAFƏDDIN

C DILINDƏ PROQRAMLAŞDIRMA

MÜNDƏRİCAT

1. SADƏ PROQRAMLAR 9
PROQRAMLAŞDIRMANIN MAHIYYƏTI 9
PROQRAMLARIN HAZIRLAMA MƏRHƏLƏLƏRI 9
C DILINDƏ SADƏ PROQRAM 10
MƏTNIN EKRANA ÇIXARILMASI 10
PROQRAMIN YERINƏ YETIRILMƏSI 11
EKRANIN DAYANDIRILMASI 11
2. DƏYİŞƏNLƏR 13
VERILƏNLƏRIN TIPLƏRI VƏ DƏYIŞƏNLƏR 13
İKI ƏDƏDIN CƏMININ HESABLANMASI 13
CƏBRI IFADƏLƏR 15
Cəbri əməliyyatların üstünlük dərəcəsi 16
Mənimsətmə operatorları 17
İnkrement və dekrement 17
Cəbri ifadələrin qısa yazılışı 17
VERILƏNLƏRIN XARICETMƏ FORMATLARI 18
Tam ədədlər 18
Həqiqi ədədlər 18
3. ŞƏRTİ OPERATORLAR 20
IF – ELSE ŞƏRTI OPERATORU 20
MÜRƏKKƏB ŞƏRTLƏR 22
SWITCH OPERATORU (ÇOXVARIANTLI SEÇIM) 23
4. DÖVRLƏR 25
"FOR " DÖVR OPERATORU 25
Ədədin kvadratının hesablanması 26
"WHILE" DÖVR OPERATORU 27
"DO-WHILE" DÖVR OPERATORU 28
DÖVRÜN VAXTINDAN TEZ DAYANDIRILMASI 29
SIRALARIN CƏMININ HESABLANMASI 30
Sonlu elementdən ibarət cəmlər 30
Məhdudlaşdırıcı şərt əsasında sıraların hesablanması 31

5. PROQRAMLARIN DÜZƏNNƏMƏSİ METODLARI 33

DEV-C++ PROQRAMININ DÜZƏNNƏMƏ VASITƏLƏRI 33

Düzənnəmə nədir? 33

Təqib etmə (ing. traccing) 33

Proqramın bir hissəsinin dayandırılması 34

"Step by step" yerinə yetirmə 34

Dəyişənlərin qiymətlərinə baxış 35

Proqramın əl ilə icra edilməsi 36

Sərhəd qiymətlərinin yoxlanılması 36

6. QRAFİKİ REJİM 38

SADƏ QRAFIKI PROQRAM 38

ŞƏKIL ÇƏKMƏYƏ NECƏ BAŞLAMAQ LAZIMDIR? 38

Nöqtələrin koordinatları 38

Rəng 39

Ayrı-ayrı piksellərlə işləmə 40

Xətlər 40

Düzbucaqlar 40

Çevrə 41

İxtiyari oblastın rənglənməsi 41

Yazılar 41

PROQRAM NÜMUNƏSI 42

7. PROSEDURALAR 43

PROSEDURALI MƏSƏLƏNIN NÜMUNƏSI 43

8. FUNKSİYALAR 47

FUNKSIYALARIN PROSEDURALARDAN FƏRQI 47

MƏNTIQI FUNKSIYALAR 48

Ədəd sadədir, yoxsa yox? 49

İki qiymət qaytaran funksiyalar 49

9. PROQRAMIN STRUKTURU 51

PROQRAMIN TƏRKIB HISSƏLƏRI 51

QLOBAL VƏ LOKAL DƏYIŞƏNLƏR 51

PROQRAMIN TƏRTIBATI 52

Funksiyaların və proseduraların tərtibatı 52

Kənar boşluqlar *54*

10. ANİMASİYA 55

Animasiya nədir? 55

Obyektin hərəkət etməsi 55

İlkin təhlil *55*

Klaviatura ilə işləmə *56*

İstiqamət düymələri vasitəsi ilə idarəetmə 57

Sadə proqram *58*

Fasiləsiz hərəkət *59*

11. TƏSADÜFİ VƏ PSEVDO TƏSADÜFİ ƏDƏDLƏR 60

Təsadüfi ədədlər nədir? 60

Təsadüfi ədədlərin paylanması 60

Təsadüfi ədədlərlə işləyən funksiyalar 61

Verilmiş intervalda təsadüfi ədədlərin alınması 61

Ekranda qarın yağması 62

12. MASSİVLƏR 64

Əsas anlayışlar 64

Massiv nədir? *64*

Massivin elanı *64*

Massivin elementinə müraciət *65*

Massivlərin daxil edilməsi və xaric edilməsi 65

Massivin təsadüfi ədədlərlə doldurulması 67

Mətn fayllarla işləmə 68

Proqramdan fayllara necə müraciət etmək olar *68*

Müəyyən ölçülü massivlər *70*

Qeyri müəyyən ölçülü massivlər *71*

Binar fayllarla işləmə 72

Massivdə sadə axtarış 74

Verilmiş elementin axtarışı *74*

Müəyyən şərtə uyğun elementlərin axtarışı *75*

Verilmiş şərt əsasında massivin yaradılması *75*

Minimal element *76*

Massiv elementlərinin yerdəyişməsi 77

Yerdəyişmə ... *77*
Inversiya ... *78*
Dövrlü sürüşdürmə ... *78*
MASSIVLƏRIN ÇEŞIDLƏNMƏSI ... 79
Hava qabarcığı üsulu ... *79*
Minimal elementin seçilməsi üsulu ... *80*
MASSIVDƏ BINAR AXTARIŞI ... 81
PROSEDURALARDA VƏ FUNKSIYALARDA MASSIVLƏRIN ISTIFADƏSI ... 82
13. SİMVOLLAR SƏTİRLƏRİ ... 84
SIMVOL SƏTRI NƏDIR? ... 84
Sətirlərin elanı və başlanğıc qiymətlərin verilməsi ... *84*
STANDART DAXILETMƏ VƏ XARICETMƏ ... 85
FAYLLARLA IŞLƏMƏ ... 86
SƏTIRLƏRLƏ IŞLƏYƏN FUNKSIYALAR ... 88
Sətrin uzunluğu – strlen ... *88*
Sətirlərin müqayisəsi – strcmp ... *89*
Sətirlərin kopyalanması - strcpy ... *91*
Sətirlərin birləşdirilməsi - strcat ... *92*
Sətirlərdə axtarış ... *95*
Sətirlərin formatlanması ... *97*
Sətirdən daxiletmə ... *97*
FUNKSIYA VƏ PROSEDURALARDA SƏTIRLƏR ... 98
14. MATRİSLƏR (İKİ ÖLÇÜLÜ MASSİVLƏR) ... 100
MATRISA NƏDIR? ... 100
MATRISALARIN ELANI ... 101
Elementlərin başlanğıc qiymətləri ... *101*
Matrisaların yaddaşda yerləşməsi ... *101*
STANDART DAXILETMƏ VƏ XARICETMƏ ... 101
Klaviaturadan daxiletmə ... *101*
Təsadüfi ədədlərlə tamamlama ... *102*
Ekrana xaricetmə ... *102*
FAYLLARLA IŞLƏMƏ ... 103
Mətn faylları ... *103*

Binar fayllar ... *103*
MATRISALARLA IŞLƏMƏK ÜÇÜN ALQORITMLƏR ... 104
Matrisanın minimal elementinin təyini ... *104*
Ayrı-ayrı elementlərlə iş ... *105*
Sətirlərin və sütunların yerdəyişməsi ... *106*
İkiölçülü matrisanın birölçülü massivə çevrilməsi ... *106*
15. SİMVOL SƏTİRLƏRİNİN MASSİVLƏRİ ... 107
İkiölçülü simvol massivlərinin elanı və inisializasiyası ... *107*
Daxiletmə və xaricetmə ... *107*
Çeşidləmə ... *108*
16. YADDAŞIN İDARƏEDİLMƏSİ ... 109
GÖSTƏRICILƏR ... 109
YADDAŞIN DINAMIK AYRILMASI ... 110
Yaddaşın ayrılmasında yaranan səhvlər ... *111*
MATRISA ÜÇÜN YADDAŞIN AYRILMASI ... 112
Ölçüsü məlum olan sətir ... *112*
Ölçüsü məlum olmayan sətirlər ... *113*
17. REKURSİYA ... 114
REKURSIYA NƏDIR? ... 114
Rekursiv obyektlər ... *114*
Rekursiv prosedura və funksiyalar ... *114*
Dolayısı rekursiya ... *115*
SONSUZ REKURSIYA ... 115
REKURSIYADAN NƏ VAXT ISTIFADƏ ETMƏK LAZIM DEYIL ... 116
REKURSIYALI AXTARIŞ ... 117
REKURSIYALI FIQURLAR ... 118
Pifaqor ağacı ... *119*
18. STRUKTURLAR ... 120
STRUKTUR NƏDIR? ... 120
ELAN VƏ INISIALIZASIYA ... 120
STRUKTUR SAHƏLƏRI ILƏ IŞ ... 121
Adla müraciət ... *121*
Ünvanla müraciət ... *121*

DAXILETMƏ VƏ XARICETMƏ 121
Element-element daxiletmə və xaricetmə *121*
Binar faylla işləmə *122*
KOPYALAMA 122
STRUKTURLARIN MASSIVLƏRI 123
YADDAŞIN DINAMIK AYRILMASI 124
STRUKTURLARIN PROSEDURA PARAMETRLƏRINDƏ ISTIFADƏ EDILMƏSI 125
Parametrlərin qiymətlərinin ötürülməsi *125*
Parametrlərin istinadla ötürülməsi *126*
Parametrlərin ünvanla ötürülməsi *126*
AÇAR ƏSASINDA ÇEŞIDLƏMƏ 126
19. PROQRAMLARIN LAYİHƏLƏNDİRİLMƏSİ........ 128
PROQRAMLARIN LAYIHƏLƏNDIRMƏ MƏRHƏLƏLƏRI 128
Məsələnin qoyuluşu *128*
Verilənlər modelinin yaradılması *128*
Alqoritmin hazırlanması *128*
Proqramın hazırlanması *129*
Proqramın düzənnəməsi *129*
Sənədlərin hazırlanması *129*
Proqramın sınaqdan keçirilməsi *129*
Müşayiət etmə *130*
"AŞAĞIDAN YUXARIYA DOĞRU" PROQRAMLAŞDIRMA 130
STRUKTUR PROQRAMLAŞDIRMA 130
"Yuxarıdan aşağıya doğru" proqramlaşdırma *130*
Struktur proqramlaşdırmanın məqsədi *131*
Struktur proqramlaşdırmanın prinsipləri *131*
Strukturlu proqramlar *131*
20. QRAFİKLƏRİN ÇƏKİLMƏSİ........ 133
PROQRAMIN STRUKTURU 133
Proqramın tərtibatı *134*
FUNKSIYALARIN YAZILIŞ QAYDALARI 135
Birbaşa yazılmış funksiyalar *135*
Qeyri aşkar funksiya *135*

Polyar koordinatlarla verilmiş funksiyalar......136
Parametrik şəkildə verilmiş funksiyalar......136
Koordinatların çevrilməsi......137
KOORDINAT SISTEMI......137
Koordinat sisteminin təsviri......137
Miqyas və koordinatların çevrilməsi......138
Koordinat oxları......139
QRAFIKLƏRIN ÇƏKILMƏSI......140
Standart üsul......140
Polyar koordinatlarla verilmiş funksiyalar......142
Parametrik şəkildə verilmiş funksiyalar......142
KƏSIŞMƏ NÖQTƏLƏRIN TƏYIN EDILMƏSI......143
Birbaşa izafə seçim üsulu......143
Dixotomiya üsulu......143
Polyar koordinatlarla verilmiş funksiyalar......145
Parametrik şəkildə verilmiş funksiyalar......147
Ümumi hal......147
Şaquli xətlə kəsişmə......148
QAPALI OBLASTIN ŞTRIXLƏNMƏSI......148
QAPALI OBLASTIN SAHƏSI......150
Ümumi yanaşma......150
Düzbucaqlar üsulu......150
Trapesiyalar üsulu......151
Monte-Karlo üsulu......152
21. ƏDƏDİ ÜSULLAR......155
TAM ƏDƏDLI ALQORITMLƏR......155
Evklid alqoritmi (I)......155
Evklid alqoritmi (II)......155
Ən kiçik ortaq bölünən......156
Eratosfen xəlbiri......156
ÇOXMƏRTƏBƏLI TAM ƏDƏDLƏR......157
100! hesablanması......158
ÇOXHƏDLILƏR......159

Qorner sxemi......159
ARDICILLIQLAR VƏ SIRALAR......160
Ardıcıllıqlar......160
Rekurent ardıcıllıqlar......161
Oxşar məsələlər......162
Sıralar......162
Funksiyaların hesablanması......163
TƏNLIKLƏRIN ƏDƏDI HƏLLI......164
Xordalar üsulu......165
Nyuton üsulu (toxunanlar üsulu)......166
İterasiyalar üsulu......167
Funksiyanın parametr əvəzi istifadəsi......168
MÜƏYYƏN INTEQRALLARIN HESABLANMASI......169
ƏYRININ UZUNLUĞUNUN HESABLANMASI......170

1. SADƏ PROQRAMLAR

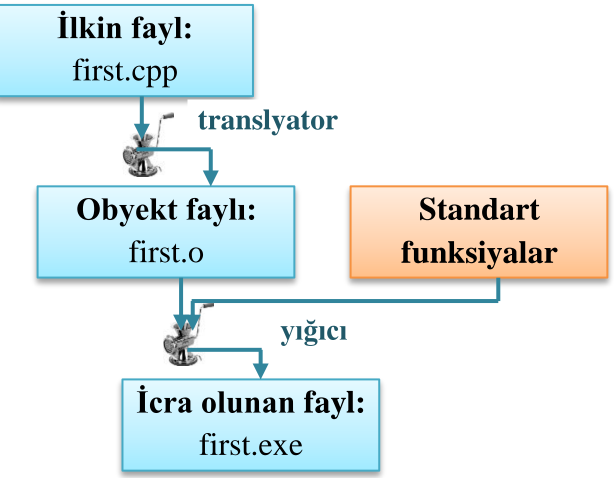
Proqramlaşdırmanın mahiyyəti

Bəzən elə təəssürat yaranır ki, bütün məsələləri kompüterdə mövcud olan hazır proqramlar vasitəsi ilə həll etmək olar. Çox vaxt bu belədir, lakin təcrübə göstərir ki, elə məsələlər var ki, onlar standart üsullarla həll olunmur. Belə məsələlərin həlli üçün xüsusi proqram yazmaq lazımdır.

Proqramların hazırlama mərhələləri

Əksəriyyət müasir proqramlaşdırma dillərdəki kimi C dilində yazılmış proqram iki mərhələ ilə yaradılır:

1. **Translyasiya** - proqram mətninin maşın dilinə çevrilməsi;
2. **Yığılma** – proqram komponentlərin və standart funksiyaların bir yerə yığılması.



Sadə proqramların hazırlanması bir mərhələli ola bilər, lakin mürəkkəb proyektlər iki mərhələyə emal olunurlar:

- adətən mürəkkəb proqram bir neçə ayrı-ayrı modullara bölünür. Bu modullar ayrıca və müxtəlif insanlarla düzəldilir, ona görə də, axırda bütün hazır modullar bir proyektə yığılmalıdır;
- bir modulda düzəliş etdikdən sonra yalnız onu translyasiya (maşın dilinə çevirmək) etmək olar;
- yığılma zamanı digər dillərdə yazılmış (məsələn, Assembler-də) modulları proqrama qoşmaq olar.

C dilinin translyatoru **kompilyator** adlanır. Kompilyatorlar bir dəfəyə bütün proqramı maşın koduna çevirirlər. Kompilyatorlardan fərqli olaraq **interpretatorlar** proqramı sətirbə-sətir maşın koduna çevirirlər. Kompilyatorlar daha sürətlə işləyirlər.

C dilində yazılmış ilkin fayl ***.c** və ya ***.cpp** genişlənməsinə malikdir (*.cpp genişlənməsi onu göstərir ki, proqramda C++ dilinin imkanlarından istifadə olunur). Bu adi mətn faylıdır, və proqramı bu fayla yazmaq üçün istənilən sadə mətn redaktorundan (məsələn, Notepad) istifadə etmək olar.

Translyator ilkin faylı maşın koduna çevirir və nəticədə eyni adlı **və** ***.o** genişlənməsi olan obyekt faylı yaradır. Buna baxmayaraq, obyekt faylı maşın dilindədir, onu icra etmək olmaz, çünki onun tərkibində standart funksiyalar (məsələn, daxiletmə-xaricetmə) yoxdur.

Yığıcı kitabxanalarda saxlanılan standart funksiyaları qoşur (onların genişlənməsi ***.a**-dır). Nəticədə *.exe genişlənməsi olan bir fayl yaranır. Bu fayl hazır icra oluna bilən proqramdır.

C dilində sadə proqram

Bu proqram cəmi 8 simvoldan ibarətdir :

```
main ()
{
}
```

Əsas proqram həmişə **main** adlanır (diqqətli olun – C dili böyük və kiçik hərfləri fərqləndirir və C dilində bütün standart operatorlar kiçik hərflərlə yazılır). Boş mötərizələr o deməkdir ki, **main** funksiyasının parametrləri yoxdur. Proqramın gövdəsi fiqurlu mötərizələr (**{**… **}**) içində yazılır. Belə ki, yuxarıdakı misalda mötərizələr içində heç nə yazılmayıb, bu o deməkdir ki, bu proqram heç bir iş görmür. Bu proqramı kompilyasiya edib, ondan icra olunan **exe**-faylı almaq olar.

Mətnin ekrana çıxarılması

Ekrana “Salam” sözünü çıxardan proqramı tərtib edək.

```
#include <stdio.h>
main ()
{
printf ( “Salam” );
}
```

stdio.h faylında saxlanılan standart daxiletmə və xaricetmə funksiyaların qoşulması

ekranda çap edən funksiyanın çağırışı

Qaydalar:

- Standart funksiyalardan istifadə etmək üçün bu funksiyanı translyatora tanıtdırmaq lazımdır, yəni o, bilməlidir ki, funksiyanın adı və parametrləri düz yazılıb, ya yox. Bu o deməkdir ki, proqrama funksiyanın *təsvirini* daxil etmək lazımdır. C dilində

standart funksiyaların təsviri ***.h** genişlənməsi olan başlıq fayllarda (ing. **header file**) saxlanılır (DevC++ proqramı üçün **C:\Dev-Cpp\include** qovluğu).

- Başlıq faylların qoşulması üçün preprosessorun **#include** direktivindən (göstərişindən) istifadə olunur. Faylın adı künclü mötərizələr (**< ... >**)arasında yazılır. Künclü mötərizələr içində boşluq olmamalıdır. Hər yeni başlıq faylın qoşulması üçün yeni **#include** direktivdən istifadə etmək lazımdır.
- Məlumatları ekrana çıxartmaq üçün **printf** funksiyasından istifadə olunur. Yuxarıdakı misalda bu funksiyanın bir parametri var – dırnaqlar arasında yazılmış cümləni ekrana çıxardan sətir.
- C dilinin istənilən operatordan sonra nöqtəli vergül (**;**) işarəsi qoyulmalıdır.

Proqramın yerinə yetirilməsi

Proqramı yerinə yetirmək üçün əvvəlcə onu translyator vasitəsi ilə maşın dilinə çevirmək, sonra isə yığıcının köməyi ilə standart funksiyaları qoşub icra olunan faylı yaratmaq lazımdır. Əvvəllər bütün bu əməliyyatları əmrləri ilə yazmaqla bir-bir yerinə yetirirdilər. Hal-hazırda bütün bu mərhələləri birləşdirib və xüsusi proqram təminatı vasitəsi ilə yerinə yetirirlər. Bu cürə proqramlar **IDE** (ing. **Integrated Development Environment**), yəni **proqramların yaradılması üçün inteqrasiya olunmuş mühit** adlanırlar. IDE-ya daxildir:

- mətn redaktoru;
- translyator;
- yığıcı;
- düzəndirici.

Bu mühitdə proqramın mətnini yığıb və bir düyməni basıb proqramı icra etmək olar (əgər proqramda səhv yoxdur).

DevC++ mühitində proqramı icra etmək üçün **F9** düyməsi basılmalıdır. Əgər proqramda səhvlər varsa, onda DevC++ ekranının aşağı hissəsində səhvlər haqqında məlumatlar əks olunacaqdır. Əgər bu sətirlərdən birini seçsək, onda proqramda səhv olduğu sətir seçilir.

Səhvləri aşkar etdikdə aşağıdakıları nəzərə almaq lazımdır:

- çox vaxt səhv həmin sətirdə yox, ondan əvvəlki sətirdə olur. Ona görə də o sətri də yoxlamaq lazımdır.
- çox vaxt bir səhv nəticəsində digər səhvlər də yaranır.

Ekranın dayandırılması

Əgər hazır olan proqramı yerinə yetirsək, onda məlum olacaqdır ki, proqram işləyib qurtarandan sonra nəticələrə baxmaq olmur. Nəticələrlə olan pəncərə tez bağlanır. Ona

görə də proqramın sonunda ekranı dayandıran funksiyadan istifadə etmək lazımdır. Bu funksiya **getch()** funksiyasıdır. O, klaviaturadan istənilən düymənin basılmasını gözləyir.

```
#include <stdio.h>
#include <conio.h>
main ()
{
printf ( "Salam");    // ekrana çıxarış
getch();              /* düymənin basılmasını gözləmək */
}
```

conio.h başlıq faylının qoşulması

Qaydalar:

- ekranın dayandırılması üçün **getch()** funksiyasından istifadə etmək lazımdır;
- bu funksiyanın təsviri **conio.h** başlıq faylında saxlanılır;
- ***//*** işarələri bir sətirli **şərhin** başlanması deməkdir. Bu cürə sətirlər translyator tərəfindən emal olunmur və proqrama aydınlıq gətirmək üçün istifadə edilir;
- Şərhi /* (şərhin başlanması) və */ (şərhin sonu) simvollar arasında da yazmaq olar. Bu halda şərh çox sətirli ola bilər.

2. DƏYİŞƏNLƏR

Verilənlərin tipləri və dəyişənlər

Verilənlərin emalı üçün onları yaddaşda saxlamaq lazımdır. Özü də, onlara hər hansı üsullarla müraciət etmək lazımdır. Adətən, insanlar bir-birinə adı ilə müraciət edirlər. Həmin üsuldan proqramlaşdırmada da istifadə olunur: yaddaşın hər xanasına (və ya bir neçə xanaya) xüsusi ad verilir. Bu addan istifadə edərək xanadan məlumatları oxumaq və yeni məlumatları yazmaq olar.

Kompüter yaddaşında müəyyən məlumatları özündə saxlayan adlı xanaya **dəyişən** deyilir. Proqramın yerinə yetirilməsi zamanı dəyişənin qiyməti dəyişə bilər. Xanaya yeni qiymət yazılarkən köhnə qiymət silinir.

Kompüter üçün yaddaşda olan bütün verilənlər - ədədlərdir (daha doğrusu, sıfırlar və birlərdir). Buna baxmayaraq məlumdur ki, kompüter tam ədədləri və kəsirləri fərqli üsullarla emal edir. Buna görə də hər proqramlaşdırma dilində verilənlərin müxtəlif tipləri var və onların emalı üçün müxtəlif üsullardan istifadə edilir. Məsələn,

- **Tam** dəyişənlər – **int** tipi (ing. *Integer* – tam). Bu tip dəyişənlər yaddaşda 4 bayt yer tuturlar;
- **Həqiqi** dəyişənlər – **float** tipi (ing. *Floating point* – sürüşən nöqtə).;
- **Simvollar – char** tipi (ing. *Character* – simvol). Bu tip dəyişənlər yaddaşda 1 bayt yer tuturlar.

Proqramda istifadə olunan bütün dəyişənləri elan etmək lazımdır. Kompüterə bildirmək lazımdır ki, yaddaşda həmin dəyişənə lazımı ölçüdə yer ayırsın və onu adlandırsın. Adətən, dəyişənləri proqramın əvvəlində elan edirlər. Elan üçün əvvəlcə tipin adı (**int**, **float** və ya **char**), sonra isə vergüllə ayrılmış bütün dəyişənlərin adları yazılmalıdır. Lazım gələrsə elan zamanı dəyişənə başlanğıc qiymət də vermək olar. Əgər dəyişənə heç bir qiymət verilmirsə, onda onun qiyməti qeyri müəyyən olur, yəni həmin xanada əvvəl yerləşən qiymət olur (“zibil”).

Misallar.

```
int a;      //tam a dəyişəni üçün yaddaş ayırmaq
float b, c; //iki həqiqi tipli b və c dəyişənləri
int Gr104, Gr07=23, Kt06; //üç tam dəyişəni,
                                  // Gr07 dəyişənə 23 qiyməti yazılır
float x=4.56, y, z;               // üç həqiqi tipli dəyişəni
                                  // x dəyişənə 4.56 ədədi yazılır
char c, c2='A', m;     // üç simvol tipli dəyişəni
                          // c2 dəyişənə ‘A’ simvolu yazılır
```

İki ədədin cəminin hesablanması

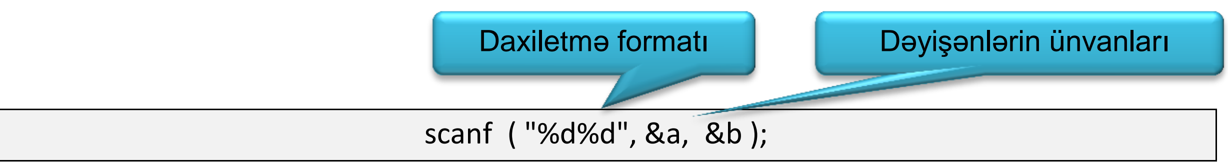
Misal. Klaviaturadan iki tam ədədi daxil edib, ekrana onların cəmini çıxartmaq.

Bu misalın həllini C dilində yazaq.

```
#include <stdio.h>
#include <conio.h>
main()
{
 int a, b, c; // dəyişənlərin elanı
 printf("İki tam ededi daxil edin\n");  //  daxiletmə üçün göstəriş
 scanf("%d%d", &a, &b);              // verilənlərin daxil edilməsi
 c = a + b;                            // hesablamalar (mənimsətmə operatoru)
 printf("%d+%d=%d", a, b, c);          // nəticənin xaric edilməsi
 getch();
}
```

Yeni məlumatlar

- Adətən proqram 4 hissədən ibarətdir:
    - Dəyişənlərin elanı;
    - Başlanğıc velilənlərin daxil edilməsi;
    - Verilənlərin emalı (hesablamalar);
    - Nəticənin xaric edilməsi.
- Verilənləri daxil etməzdən əvvəl ekrana əlavə məlumat çıxartmaq lazımdır. Bu daxiletmə üçün əlavə göstərişdir (bunu eləməyəndə kompüter verilənlərin daxil edilməsini gözləyir, lakin ekranda heç nə olmadığından istifadəçi bilmir, nə etmək lazımdır).
- **printf** funksiyasında **\n** simvolu yeni sətrin əvvəlinə keçmək deməkdir.
- Verilənlərin daxil edilməsi üçün **scanf** funksiyasından istifadə edilir.



- Daxiletmə formatı verilənlərin bir və ya bir neçə daxiletmə formatını sadalayan dırnaq arasında yazılan sətirdir:

%d Tam ədədin daxil edilməsi (**int** tipli dəyişən)

%f Həqiqi tipli ədədin daxil edilməsi (**float** tipli dəyişən)

%c Bir simvolun daxil edilməsi (**char** tipli dəyişən)

- Daxiletmə formatından sonra vergül qoyub daxil edilmiş qiymətləri yazmaq üçün yaddaş xanalarının ünvanlarını sadalamaq lazımdır. Fərqi hiss etmək üçün:

a yazılanda - a dəyişəninin qiymətidir
&a yazılanda – a dəyişəninin ünvanıdır

- Sətirdə formatların sayı siyahıda dəyişənlərin ünvanlarının sayına bərabər olmalıdır. Bundan əlavə, dəyişənin tipi elan olunmuş tiplə üst-üstə düşməlidir. Məsələn, əgər a və b - tam tipli dəyişənlərdirsə, onda aşağıdakı funksiyalar səhvdir.

scanf (“%d%d”, &a);	ikinci daxil edilmiş ədəd hara yazılmalıdır?
scanf (“%d%d”, &a, &b, &c);	**c** dəyişəni üçün format göstərilməyib
scanf (“%f%f”, &a, &b);	Tam dəyişənləri həqiqi formatla daxil etmək olmaz

- Hesablamalar üçün **mənimsətmə operatorundan** istifadə edilir. Operatorun sağ tərəfində hesablanan cəbri ifadə yazılır, bərabər işarəsinin sol tərəfində isə nəticəni mənimsəyən dəyişənin adı yazılır.
  c=a+b; // a və b dəyişənlərin cəmini c dəyişənə yazmaq
- Ədədləri və dəyişənlərin qiymətlərini ekrana çıxartmaq üçün **printf** funksiyasından istifadə etmək lazımdır.

```
printf ( "Netice: %d + %d = %d \n", a, b, c );
```

Çap üçün verilənlər

Tam ədədləri çıxartmaq

Bu simvolları necə var çıxartmaq

- **printf** funksiyasının parametrləri **scanf** funksiyasına oxşardır. Əvvəlcə simvollar sətri (xaricetmə formatı) yazılır. Sətirdə aşağıdakı xüsusi simvollardan istifadə etmək olar:

%d Tam ədədin xaric edilməsi

%f Həqiqi tipli ədədin xaric edilməsi

%c Bir simvolun xaric edilməsi

%s Simvollar sətrinin xaric edilməsi

\n Yeni sətrin əvvəlinə keçid

Digər simvollar (digər xüsusi əmrlərdən başqa) dəyişilməz olaraq ekranda əks olunur.

- Format sətrindən sonra vergül qoyulur və ondan sonra ekranda əks olunacaq dəyişənlərin adları, ədədlər və ya ifadələr yazılır.

```
printf ( "Netice: %d + %d = %d \n", a, 5, a+5 );
```

- **printf** funksiyasında dəyişənlərin sayı və tipləri formata uyğun olmalıdır.

Cəbri ifadələr

Mənimsətmə operatorun sağ tərəfində yerləşən cəbri ifadələr aşağıdakılardan ibarət ola bilərlər:

- tam və həqiqi ədədlər (tam və kəsr hissəsinin ayırıcısı nöqtədir)
- cəbri əməliyyatların işarələri
  \+ - toplama, çıxma

* / vurma, bölmə
% bölmə qalığı

- standart funksiyalara müraciət

abs(i) tam i ədədin mütləq qiyməti (modulu)- $|i|$
fabs(x) həqiqi x ədədin modulu - $|x|$
sqrt(x) həqiqi x ədədin kök altısı -$\sqrt{x}$
pow(x,y) y tərtibli x - x^y

- əməliyyatların ardıcıllığını dəyişmək üçün mötərizələrdən istifadə edilir.

C dilində bölmə əməliyyatının bəzi spesifik xüsusiyyətləri vardir və onları nəzərə almaq lazımdır:

İki tam ədədin bölünməsi nəticəsində bölmə qalığı nəzərə alınmır, yəni nəticə tam tipli alınır. Məsələn, 7/4 ifadənin nəticəsi 1 olacaqdır. Əgər nəticə həqiqi tipdə olmalıdırsa və qalıq da nəzərə alılmalıdırsa, onda bölən və ya bölünən ədədlərdən birini həqiqi tipə çevirmək lazımdır. Məsələn:

```
 int i, n;
 float x;
i=7;
x=i / 4;           //   x=1, iki tam ədədlərin bölünməsi
x= i / 4. ;        //  x=1.75, tam ədəd həqiqi ədədə bölünür
x= ( float) i / 4;   // x=1.75, həqiqi ədəd tam ədədə bölünür
n= 7. / 4. ;       // n=1, nəticə tam ədədə yazılır
```

Əgər a dəyişənin b dəyişənə bölmə nəticəsində qalıq hesablanmalıdır və nəticə AB dəyişənə yazılmalıdırsa, onda % əməliyyatından istifadə etmək lazımdır.

```
AB=a % b ;
```

Cəbri əməliyyatların üstünlük dərəcəsi

Proqramlaşdırma dillərində cəbri ifadələr bir mərtəbəli yazılırlar, ona görə də əməliyyatların **prioritetini** (üstünlük dərəcəsini), yəni yerinə yetirilmə ardıcıllığını bilmək lazımdır. Əvvəlcə

- mötərizələrdə olan əməliyyatlar yerinə yetirilir;
- funksiyalar hesablanır;
- soldan sağa doğru vurma, bölmə, qalığın tapılması əməliyyatları yerinə yetirilir;
- soldan sağa doğru toplama və çıxma əməliyyatları yerinə yetirilir.

Məsələn:

```
      2   1     5  4       3      8      6    7
x = ( a + 5 * b ) * fabs (c + d ) – ( 3 * b – c ) ;
```

Əməliyyatların ardıcıllığını dəyişmək məqsədilə mötərizələrdən istifadə olunur. Aşağıdakı ifadəyə baxaq:

$$y = \frac{4x+5}{(2x-15z)(3z-3)} - \frac{5x}{x+z+3}$$

İfadənin kompüterdə yazılışı belə olacaqdır:

```
y = ( 4 * x + 5 ) / ( ( 2 * x - 1 5 * z ) * ( 3 * z - 3 ) ) - 5 * x / ( x + z + 3 ) ;
```

Mənimsətmə operatorları

Proqramlaşdırmada tez-tez mənimsətmə operatorlarından istifadə edilir. Məsələn,

```
i = i + 1 ;
```

Riyazi baxımından bu tənlik mənasızdır, lakin informatika baxımından bu operator ***i*** dəyişənin qiymətini bir vahid artırır. **i**-nin köhnə qiyməti götürülür, bir vahid artırılır və köhnə qiymətinin yerinə yazılır.

İnkrement və dekrement

C dilində hər hansı bir dəyişənin qiymətini bir vahid artırılması üçün (**inkrement**) xüsusi operatorlar təyin olunub.

```
i ++ ;    // və ya ...
++ i ;
```

Həmin operatorları mənimsətmə operatoru vasitəsi ilə yazmaq olar.

```
i = i + 1 ;
```

C dilində hər hansı bir dəyişənin qiymətini bir vahid azaldılması üçün (**dekrement**) xüsusi operatorlar təyin olunub.

```
i - - ;    // və ya ...
- - i ;
```

Həmin operatorları mənimsətmə operatoru vasitəsi ilə yazmaq olar.

```
i = i - 1 ;
```

İnkrement operatorların birinci forması postfiks, ikinci forması isə prefiks adlanır. Əgər bu operatorlar hər-hansı mürəkkəb və ya şərti operatorların daxilində istifadə olunursa, onda prefiks və postfiks formaları fərqli nəticə verirlər.

Cəbri ifadələrin qısa yazılışı

Əgər hər-hansı dəyişənin qiyməti dəyişilməlidir və alınmış yeni qiymət köhnə qiyməti əvəz etməlidirsə, onda mənimsətmə operatorun qısa variantından istifadə etmək rahatdır.

Qısa yazılış	Tam yazılış
x += a;	x = x + a;
x -= a;	x = x - a;
x *= a;	x = x * a;
x /= a;	x = x / a;
x %= a;	x = x % a;

Verilənlərin xaricetmə formatları

Tam ədədlər

scanf və **printf** funksiyaların birinci parametri verilənlərin daxiletmə və xaricetmə formatını müəyyən edən simvollar sətri olmalıdır. Verilənləri daxil edən **scanf** funksiyasında tam ədədlər üçün **%d**, həqiqi ədədlər üçün **%f** və simvol üçün **%c** formatı göstərilməlidir. **printf** funksiyasının format hissəsi daha geniş imkanlarına malikdir. O, ekrana çıxarılan verilənlərə ayrılmış sahələrin ölçülərini müəyyən edə bilər.

Aşağıdakı cədvəldə **1234** ədədin ekrana çıxarılmasının müxtəlif formatları göstərilmişdir.

Operatorun yazılışı	Nəticə	Şərh
printf(“[%d]”, 1234) ;	[1234]	Minimal mümkün olan sahə
printf(“[%6d]”, 1234) ;	[1234]	6 pozisiya, sağdan düzəlmə
printf(“[%-6d]”, 1234) ;	[1234]	6 pozisiya, soldan düzəlmə
printf(“[%2d]”, 1234) ;	[1234]	Ədəd ona ayrılmış sahəyə (2 pozisiya) yerləşmədiyinə görə xaricetmə sahəsi artırılır

Simvolların xaric edilməsi üçün eyni format üsullarından istifadə edilir, lakin **%d %c-** ilə əvəz olunur.

Həqiqi ədədlər

Həqiqi ədədlərin daxil və xaric edilməsi üçün üç formatdan istifadə oluna bilər: **%f, %e və %g**. %f formatı üçün misallar aşağıdakı cədvəldə göstərilmişdir:

Operatorun yazılışı	Nəticə	Şərh
printf(“[%f]”, 123.45) ;	[123.450000]	Minimal mümkün olan sahə, kəsr hissəsində 6 rəqəm
printf(“[%9.3f]”, 123.45) ;	[123.450]	Cəmi 9 pozisiya, onlardan 3-ü kəsr hissəsinindi. Sağdan düzəlmə.
printf(“[%-9.3f]”, 123.45) ;	[123.450]	Cəmi 9 pozisiya, onlardan 3-ü kəsr hissəsinindi. Soldan düzəlmə.
printf(“[%6.4f]”, 123.45) ;	[123.4500]	Ədəd ona ayrılmış 6 pozisiyaya yerləşmədiyinə görə (kəsr hissəsinə 4 yer ayrılır) xaricetmə sahəsi artırılır

%e formatı elmi hesablamalarda çox böyük və ya çox kiçik ədədlərin (atom ölçüləri, Günəşə qədər məsafə) xaric edilməsi üçün istifadə olunur. Bu format **standart** və ya **sürüşən nöqtəli təsvir** adlanır. Bu formata əsasən ədəd *mantissa* və *tərtibdən* ibarətdir. Məsələn, **123.45** ədədi standart formatda belə yazıla bilər: $123.45=1.2345 \cdot 10^2$. Burada **1.2345** – mantissa (mantissa həmişə 1÷10 arası dəyişməlidir), **2** – tərtibdir. %e formatı əsasında ədədin təsviri üçün ümumi pozisiyaların və mantissanın kəsr hissəsində rəqəmlərin sayını təyin etmək olar. Bu formatda tərtib həmişə 2 rəqəmli olur, tərtibin önündə **e** hərfi dayanır. Tərtibin işarəsi həmişə yazılır (- və ya +).

Operatorun yazılışı	Nəticə	Şərh
printf(“[%e]”, 123.45) ;	[1.234500e+02]	Minimal mümkün olan sahə, mantissanın kəsr hissəsində 6 rəqəm

printf("[%12.3e]", 123.45) ;	[1.234e+02]	Cəmi 12 pozisiya, onlardan 3-ü mantissanın kəsr hissəsinindi. Sağdan düzəlmə.
printf("[%-12.3e]", 123.45) ;	[1.234e+02]	Cəmi 12 pozisiya, onlardan 3-ü mantissanın kəsr hissəsinindi. Soldan düzəlmə.
printf("[%6.2e]", 123.45) ;	[1.23e+02]	Ədəd ona ayrılmış 6 pozisiyaya yerləşmədiyinə görə (mantissanın kəsr hissəsinə 2 yer ayrılır) xaricetmə sahəsi artırılır.

%g formatı kəsr hissəsinin sonunda artıq sıfırları götürmək və avtomatik formatın (onluq və ya sürüşən nöqtəli format) seçilməsi üçün istifadə edilir. Çox kiçik və ya çox böyük ədədlərə sürüşən nöqtəli format tətbiq olunur. %g formatında pozisiyaların ümumi sayı və vacib rəqəmlərin sayı göstərilir.

Operatorun yazılışı	Nəticə	Şərh
printf("[%g]", 12345) ; printf("[%g]", 123.45) ; printf("[%g]", 0.000012345) ;	[12345] [123.45] [1.2345e-05]	Minimal mümkün olan sahə, vacib rəqəmlərin sayı 6-dan çox deyil
printf("[%10.3g]", 12345) ; printf("[%10.3g]", 123.45) ; printf("[%10.3g]", 0.000012345) ;	[1.23e+04] [123] [1.23e-05]	Cəmi 10 pozisiya, onlardan 3-ü rəqəmlərin sayıdır. Sağdan düzəlmə. Soldan düzəlməni etmək üçün "%-10.3g" formatından istifadə etmək lazımdır.

3. ŞƏRTİ OPERATORLAR

Sadə proqramlarda bütün əmrlər ardıcıl olaraq yerinə yetirilir. Bu cürə alqoritmlər **xətti alqoritmlər** adlanır. Lakin, bəzi hallarda müəyyən şərtdən asılı olaraq bir neçə variantdan seçim etməyə lazım gəlir. Əgər şərt doğrudursa, onda bir qrup əməliyyat, əks halda – digər qrup əməliyyat yerinə yetirilir. Bundan ötrü **budaqlanan alqoritmlərdən** istifadə edilir. Proqramlaşdırma dillərində budaqlanan alqoritmlər **şərti operatorlarla** təsvir olunurlar. C dilində iki cürə şərti operator mövcuddur:

- iki variantdan birini seçən **if – else** operatoru
- bir neçə variantdan birini seçən çoxvariantlı seçim - **switch** operatoru

if – else şərti operatoru

Misal. Klaviaturadan daxil olunmuş ədədlərdən ən böyününü təyin etmək.

Tapşırığa əsasən, cavab iki cürə ola bilər: əgər birinci ədəd ikincidən böyükdürsə, onda ekrana birinci ədədi, əks halda – ikinci ədədi çıxartmaq lazımdır. Aşağıda bu məsələnin iki həll variantı göstərilmişdir: birincidə nəticə dərhal ekrana çıxarılır, ikincidə isə nəticə **Max** adlanan dəyişənə yazılır, sonra ekrana çıxardılır.

```
#include <stdio.h>
#include <conio.h>
main()
{
float A, B;
printf (“A ve B-ni daxil edin: ”);
scanf ( “%f%f”, &A, &B );
if ( A > B )
  {
  printf ( “ Boyuk eded %f” , A );
  }
else
  {
  printf ( “ Boyuk eded %f”, B );
   }
getch ();
}
```

```
#include <stdio.h>
#include <conio.h>
main()
{
float A, B, Max;
printf (“A ve B-ni daxil edin: ”);
scanf ( “%f%f”, &A, &B );
if ( A > B )   // başlıq
  {
   Max = A;  // “əgər” bloku
   }
else
   {
   Max = B;  // “əks halda” bloku
   }
printf ( “ Boyuk eded %f”, Max );
getch ();
}
```

- Şərti operatorun yazılış qaydası aşağıdakı kimidir:

```
if ( şərt )  // şərti operatorun başlığı
   {
   …  //  "əgər" bloku – başlıqdakı şərt doğru olduqda,
        // bu blokun operatorları yerinə yetirilir
    }
else
  {
   …  //  "əks hal" bloku - başlıqdakı şərt yalan olduqda,
        // bu blokun operatorları yerinə yetirilir
   }
```

- **if –else** operatoru – vahid bir operatordur, ona görə də "əgər" blokunu yekunlaşdıran mötərizə (}) və **else** sözün arasında heç bir operator yazıla bilməz;
- **else** sözündən sonra heç vaxt şərt yazılmır. "əks hal" bloku başlıqdakı şərt yalan olduqda yerinə yetirilir;
- əgər "əgər" və "əks hal" blokların tərkibində bir operator yazılırsa, onda fiqur mötərizələri yazmamaq da olar;
- Şərtdə aşağıdakı müqayisə operatorlardan istifadə etmək olar:

< böyük
> kiçik
<= kiçik və ya bərabər
>= böyük və ya bərabər
== bərabər
!= bərabər deyil

- C dilində sıfıra bərabər olmayan ədəd doğru şərt deməkdir, sıfır isə - yalan şərtdir;
- Əgər "əks hal" blokunda heç nə etmək lazım deyilsə, onda "əks hal" blokunu yazmamaq olar. Məsələn, əgər a≠0, onda c=b/a (əks halda heç nə etmək lazım deyil). if –else operatorun qısaldılmış variantı belədir:

```
if ( şərt )
  {
  …    // şərt doğru olduqda bu operatorlar yerinə yetiriləcəkdir
   }
```

Əvvəlki məsələnin həlli belə ola bilərdi:

```
#include <stdio.h>
#include <conio.h>
main()
{
float A, B, Max;
printf ( "A ve B-ni daxil edin: " );
scanf ( "%f%f", &A, &B );
Max = A ;
```

```
if ( B > A )
   Max =  B;
printf ( "Boyuk eded %f", Max );
getch ();
}
```

- "əgər" və "əks hal" bloklarının tərkibində digər operatorlar ola bilər. Bir **if – else** operatorun tərkibində digər **if – else** operatoru ola bilər. Belə olduğu halda **else** operatoru ən yaxın if operatoruna aid edilir.

```
if ( A > 300)
   if ( A > 600 )
      printf ( "Sizin qebul baliniz yuksekdir! " );
      else
      printf ( "Siz kifayyet qeder bal yigmisiniz." );
else
   printf ( "Sizin baliniz asagidir!" );
```

- Proqramı asanlıqla oxumaq üçün bütün daxili if-else operatorları 2-3 simvol sağa sürüşdürülür.

Mürəkkəb şərtlər

Sadə şərtlər bir müqayisə operatorundan (<, >, <=, >=, ...) ibarətdir. Bəzən şərtlər 2 və daha çox sadə şərtlərdən ibarət olur. Məsələn, əgər hər hansı bir firma yaşı yalnız 25-40 arasında dəyişən işçiləri işə götürürsə, onda proqram aşağıdakı kimi yazılmalıdır:

```
#include <stdlib.h>
#include <conio.h>
main ()
{
int age;
printf ( "\nYasinizi daxil edin: ");
scanf ( "%d", &age );
if ( 25 <= age && age <= 40 )    // mürəkkəb şərt
   printf ( "Uyğundur! " );
else
   printf ( "Uyğun deyil! " );
getch ();
}
```

- **Mürəkkəb şərtlər** 2 və daha çox sadə şərtlərdən ibarət olurlar. Onlar **məntiqi əməliyyat** işarələri ilə birləşdirilir:

- **VƏ** (&&) əməliyyatı (**AND**). Bu əməliyyatda şərtlər eyni zamanda doğru olmalıdırlar

şərt1 && şərt2

Bu əməliyyatı aşağıdakı **doğruluq cədvəli** adlanan cədvəllə təsvir etmək olar.

şərt1	şərt2	şərt1 && şərt2
yalan (0)	yalan (0)	yalan (0)
yalan (0)	doğru (1)	yalan (0)
doğru (1)	yalan (0)	yalan (0)
doğru (1)	doğru (1)	doğru (1)

- **VƏ YA** (||) əməliyyatı (**OR**). Bu əməliyyatda heç olmasa bir şərtin doğru olması tələb olunur.

şərt1 || şərt2

Doğruluq cədvəli aşağıdakı kimidir:

şərt1	şərt2	şərt1 | | şərt2
yalan (0)	yalan (0)	yalan (0)
yalan (0)	doğru (1)	doğru (1)
doğru (1)	yalan (0)	doğru (1)
doğru (1)	doğru (1)	doğru (1)

Mürəkkəb şərtlərdə bəzən **YOX** (ing. **NOT**) əməliyyatından istifadə edilir (inkaretmə).

! şərt

Məsələn, aşağıdakı şərtlər eynidir:

A > B ⇔ ! (A <= B)

Məntiqi və müqayisə əməliyyatlarının yerinə yetirilmə ardıcıllığı (üstünlük dərəcəsi):

- mötərizələrdəki əməliyyatlar, sonra ...
- NOT əməliyyatı, sonra ...
- müqayisə əməliyyatları >, >=, <, <=, ==, !=, sonra ...
- && əməliyyatı, sonra ...
- || əməliyyatı

Əməliyyatların ardıcıllığını dəyişmək üçün mötərizələrdən istifadə olunur.

switch operatoru (çoxvariantlı seçim)

Əgər tam və ya simvol tipli dəyişənin qiymətindən asılı olaraq bir neçə variantdan biri seçilməlidirsə, onda bir-birinin daxilində yerləşən bir neçə **if** operatorundan istifadə etmək olar, lakin bunun üçün nəzərdə tutulmuş xüsusi **switch** operatorundan istifadə etmək daha əlverişlidir.

Misal. Klaviaturadan hərflə imtahan qiyməti daxil olunur. Hərflə yazılmış qiymətə əsasən ekrana imtahan qiymətini yazılı çıxartmaq.

```
#include <stdio.h>
#include <conio.h>
main ()
```

```
{
char c;
printf ( “\nİmtahan qiymetini herfle daxil edin: “ );
scanf ( “%c”, &c );    // İmtahan qiymətini daxil edin
switch ( c )             //  seçim operatorun başlığı
  {
   case ‘A’ : printf ( “\nEla” ); break;
   case ‘B’ : printf ( “\nCox yaxsi” ); break;
   case ‘C’ : printf ( “\nYaxsi” ); break;
   case ‘D’ : printf ( “\nQenaetbexsh” ); break;
   case ‘E’ : printf ( “\nKafi” ); break;
   case ‘F’ : printf (“\nQeyri kafi” ); break;
  default :   printf (“\nBele qiymet yoxdur!” );   // susmaya görə
  }
getch();
}
```

- Çoxvariantlı seçim operatoru – **switch** başlıqdan və fiqurlu mötərizələrdə yazılmış operatorun gövdəsindən ibarətdir.
- Başlıqda **switch** açar sözündən sonra mötərizələrin içində dəyişənin (tam və ya simvol tipli) adı yazılır. Dəyişənin qiymətindən asılı olaraq bir neçə variantdan seçim baş verir.
- Hər bir varianta **case** bölməsi uyğundur. **case** sözündən sonra dəyişənin mümkün qiymətlərdən biri yazılır və qoşa nöqtələr qoyulur. Əgər dəyişənin qiyməti **case**-də yazılan qiymətlərin hər hansı ilə üst-üstə düşürsə, onda proqram həmin **case** bölməsinə keçir və orada olan operatorları yerinə yetirir.
- **break** operatoru **switch** operatorun gövdəsindən çıxmaq üçün nəzərdə tutulub. Əgər bütün **break** operatorları silinərsə, onda, məsələn, “A” hərfi daxil olduqda çapa bütün variantlar çıxarılacaqdır.

Ela

Cox yaxsi

Yaxsi

Qenaetbexsh

Kafi

Qeyri kafi

Bele qiymet yoxdur!

- Əgər dəyişənin qiyməti **case**-də yazılan heç bir qiymətlə üst-üstə düşmürsə, onda proqram **default** bölməsinə keçir. **default** bölməsi olmaya da bilər.
- Hər **case** bölməsində 2 qiymət ola bilər. Məsələn, əgər proqramın həm kiçik, həm də böyük həriflərə reaksiyası eyni olmalıdırsa, onda **switch** operatoru belə yazılmalıdır:

```
switch ( c )
  {
   case ‘A’ :
   case ‘a’ :
```

```
        printf ( "\nEla" ); break;
   case 'B' :
   case 'b' :
        printf ( "\nCox yaxsi" ); break;
............................
    }
```

4. DÖVRLƏR

Bəzən eyni məlumatları bir neçə dəfə ekrana çıxartmaq lazım gəlir. Misal olaraq, “Salam!” sözünü 10 dəfə ekranda çap edək. Əlbəttə, 10 dənə **printf** operatorundan istifadə etməklə bu tapşırığı yerinə yetirmək olar. Lakin bu vəziyyətdən çıxış yolu deyil, çünki ola bilər ki, bu sözü 10 yox, 200 dəfə çap etməyə lazım gələr. Onda proqram çox uzanacaqdır. Ona görə də **dövrlərdən** istifadə etmək lazım gəlir.

Bir neçə dəfə yerinə yetirilən operatorlar ardıcıllığı **dövr** adlanır.

C dilində bir neçə dövr operatoru var.

“for “ dövr operatoru

Hər hansı bir əməliyyatın neçə dəfə təkrar olunduğu adətən bilinir. Bəzi proqramlaşdırma dillərində bunun üçün **repeat** (lazımı qədər təkrar et) dövründən istifadə edilir. Yaddaşda bir xana ayrılır və həmin xanaya təkrarlamaların sayı yazılır. Proqram dövrü bir dəfə yerinə yetirəndə, xananın tərkibi (*sayğac*) bir vahid azalır. Xanada sıfır olanda dövr öz işini bitirir.

C dilində **repeat** əvəzinə **for** operatorundan istifadə edilir. f**or** dövrü sayğac-xananı gizlətmir, onu elan etməyə tələb edir (yaddaşda yer ayırmaq üçün), və, hətta, onun qiymətini dövrün gövdəsində istifadə etməyə imkan yaradır. Aşağıdakı proqramda “Salam!” sözü 10 dəfə çap olunur.

```
#include <stdio.h>
#include <conio.h>
main ()
{
int i ;                          //  dövr dəyişəninin elanı
for ( i=1 ;  I <=10 ; i++ )  // dövrün başlığı
   {                             // dövrün başlanması
   printf ( "Salam!" );     // dövrün gövdəsi
   }                           // dövrün sonu
getch () ;
}
```

- Dövr təkrarlamaların sayı məlum olduqda və ya hesablana bildikdə **for** dövrü operatorundan istifadə edilir;

- **for** dövrü operatoru başlıqdan və gövdədən ibarətdir;
- Başlıqda **for** sözündən sonra nöqtəli vergüllə ayrılmış 3 ifadə yazılır:
    - **başlanğıc qiymətlər**: mənimsətmə operatorları. Onlar yalnız 1 dəfə dövr başlayanda yerinə yetirilir;
    - **dövrün növbəti addımına keçmək üçün şərt**: əgər şərt yalandırsa, onda dövr öz işini dayandırır; əgər şərt əvvəldən yalandırsa, onda dövr heç bir dəfə də yerinə yetirilmir;
    - **dövrün hər addımında yerinə yetirilən əməliyyatlar**. Çox vaxt bunlar mənimsətmə operatorlardır.
- Başlığın hər bir hissəsində vergüllə ayrılmış bir neçə operator ola bilər. Məsələn, başlıqlar aşağıdakı kimi ola bilərlər:

```
for ( i=0 ;i<10 ; i++ ) { … }
for ( i=0, x=1.;  i<10;  I +=2, x *=0.1 ) { …  }
```

- Dövrün gövdəsi fiqur mötərizələr içərisində yazılmalıdır; əgər dövrün gövdəsində yalnız bir operator varsa, mötərizələri qoymamaq olar.
- Dövrün gövdəsində digər operatorlar ola bilər, o cümlədən də dövrlər (*iç-içə dövrlər*).
- Proqramı yaxşı başa düşmək üçün dövrün gövdəsini mötərizələrlə birlikdə 2-3 addım sağa sürüşdürülər.

Ədədin kvadratının hesablanması

Misal: Klaviaturadan hər hansı bir natural ədəd (**N)** daxil olunur. Ekrana 1-dən N-ə qədər bütün tam ədədlərin kvadratlarını aşağıda göstərilən şəkildə çıxartmaq lazımdır:

1 kvadratı bərabərdir 1

2 kvadratı bərabərdir 4

……

```
#include <stdio.h>
#include <conio.h>
main ()
{
int i, N ;                                      //  i – dövrün dəyişənidir
printf ( “N-in qiymetini daxil edin: “) ;    //  daxiletmə üçün göstəriş
scanf ( “%d”, &N ) ;                          // klaviaturadan N-nin daxil edilməsi
for ( i=1 ; i <= N ; i++ )                    // dövrün başlığı
   {
   printf ( “ %d kvadrati beraberdir %d\n”, i, i*i );
   }
getch ();
}
```

Proqramda 2 dəyişən elan olunub: **N** – maksimal ədəd və **i** - əlavə dəyişən, hansı ki dövrdə, ardıcıl olaraq, 1-dən N-ə kimi qiymət alır. N-nin qiymətini daxil etmək üçün göstəriş

verilir (**printf** operatoru). **scanf** operatoru isə %d formatından istifadə edərək N-nin qiymətini daxil edir.

Dövrə daxil olduqda **i=1** operatoru yerinə yetirilir, sonra isə hər addımda i-nin qiyməti bir vahid artirilir (**i++**). Dövr şərt **i<=N** doğru olana kimi işləyir. Dövrün gövdəsində yerləşən yeganə operator ədədin özünü və onun kvadratını verilmiş formata əsasən çapa göndərir. Kvadrata yüksəltmək üçün vurmadan istifadə etməyə daha məsləhətdir.

"while" dövr operatoru

Bəzən müəyyən əməliyyatın neçə dəfə yerinə yetirilməsi haqqında heç bir şey məlum deyil, lakin hansı şərt əsasında o yerinə yetirilməlidir - bu şərti təyin etmək olur. "while" sözü "hələ ki" kimi tərcümə olunur, yəni hələ ki şərt doğrudur, bu operatoru yerinə yetir. Belə işləyən dövr "**while" dövr operatoru** adlanır.

Misal. Tam ədədi daxil edərək onun tərkibində olan rəqəmlərin sayını təyin edin.

Bu məsələni həll etmək üçün aşağıdakı alqoritmdən istifadə etmək lazımdır.

Ədəd, ardıcıl olaraq, 10-a bölünür və qalıq nəzərə alınmır. Bu əməliyyat bölmə nəticəsi sıfır alınana kimi davam etdirilir. Xüsusi dəyişənin (sayğacın) köməyi ilə bölmələrin sayını (ədəddə olan rəqəmlərin sayını) hesablayırıq. Aydındır ki, bu məsələnin həllində **for** dövr operatorundan istifadə etmək olmaz, çünki dövrlərin sayı məlum deyil. Ona görə də məsələnin həlli üçün **while** dövr operatorundan istifadə olunmalıdır.

```
#include <stdio.h>
#include <conio.h>
main ()
{
int N ;                              // daxil edilmiş ədəd
int count=0;                         // sayğac tipli dəyişən
printf ( "\nEdedi daxil edin: " );   // daxiletmə üçün göstəriş
scanf ("%d", &N );                   //  klaviaturadan N ədədin oxunması
while ( N > 0 )                      // dövrün başlığı (hələ ki N>0 )
    {                                // dövrün başlangıcı
    N /=10 ;                         // axırıncı rəqəmin aradan götürülməsi
     count ++;                       // sayğacın 1 vahid artırılması   <- Dövrün gövdəsi
    }                                // dövrün sonu
printf ( "Bu ededin %d reqemi var\n", count ) ;
getch ();
}
```

- **while** dövr operatoru o vaxt istifadə edilir ki, nə vaxt dövrdəki addımların sayı məlum deyil və ya hesablana bilməz.
- **while** dövrü başlıqdan və gövdədən ibarətdir.

- Başlıqda **while** sözündən sonra mötərizədə dövrü davam etdirmək üçün şərt yazılır. Şərt yalan olanda, dövr öz işini dayandırır.
- Şərtdə müqayisə və məntiqi operatorlardan istifadə etmək olar:

<, > böyük, kiçik
<=, >= kiçik bərabər, böyük bərabər
== bərabər
!= bərabər deyil

- Əgər dövr başlayanda şərt artıq yalandırsa, onda dövr yerinə yetirilmir.
- Əgər şərt heç vaxt yalan olmursa, onda dövr sonsuz davam edir. Bu ciddi məntiqi səhvdir.
- C dilində sıfıra bərabər olmayan istənilən ədəd *doğru*, sıfır isə - *yalan* şərt deməkdir.

```
while ( 1 ) { … }   // sonsuz dövr
while ( 0 ) { … }   // dövr heç bir dəfə də yerinə yetirilməyəcəkdir
```

- Dövrün gövdəsi fiqur mötərizələr içində yazılır. Əgər gövdədə bir operator varsa, onda mötərizələr lazım deyil.
- Dövrün gövdəsində digər operatorlar, həmçinin dövrlər də ola bilər.
- Proqramı asanlıqla başa düşmək üçün dövrün gövdəsi mötərizələrlə birlikdə 2-3 simvol sağa sürüşdürülür.

"do-while" dövr operatoru

Bəzən elə hallar olur ki, dövrü heç olmasa bir dəfə yerinə yetirib, sonra isə müəyyən şərt əsasında onu davam etdirmək lazımdır. Bundan ötrü **do-while** dövr operatorundan istifadə olunur. Burada, əvvəlcə dövrün gövdəsindəki operatorlar yerinə yetirilir, sonra şərt yoxlanılır, yəni şərt axırda yoxlanılır.

Misal. Natural ədədi daxil edərək onun rəqəmlərin cəmini hesablayıb ekranda əks etdirmək tələb olunur. Proqramı elə təşkil etmək lazımdır ki, mənfi və ya sıfır ədədi daxil etmək mümkün olmasın.

İstənilən proqram elə yazılmalıdır ki, düzgün olmayan verilənləri proqrama daxil etmək mümkün olmasın. Belə ki, istifadəçi səhv verilənləri bir neçə dəfə daxil edə bilər, ona görə də müəyyən şərt əsasında dövr təşkil olunmalıdır. Digər tərəfdən, ədədi heç olmasa bir dəfə daxil etmək lazımdır. Ona görə də do-while dövr operatorundan istifadə olunmalıdır.

Əvvəlki proqramdan fərqli olaraq indi hər addımda bölmə qalığını təyin etmək lazımdır. Ədədin sonuncu rəqəmi onun 10-a bölmə qalığına bərabərdir. Qalıqları aldıqca onları əlavə dəyişənin içində cəmləmək lazımdır.

```
#include <stdio.h>
#include <conio.h>
main ()
{
int N, sum;              // sum – rəqəmlərin cəmidir
sum = 0;                //  əvvəlcə cəmi sıfırlaşdırırıq
```

```
do {                          // dövrün başlığı
   printf ( "\nNatural ededi daxil edin: ") ;
   scanf ( "%d", &N ) ;
   }
while ( N <=0 ) ;       // dövrün şərti (hələ ki N <=0)
while ( N > 0 ) {
   sum +=N % 10 ;
   N /=10 ;
   }
printf ( "Verilmis ededde reqemlerin cemi beraberdir  %d\n", sum ) ;
getch () ;
}
```

- Əgər dövrdəki təkrarlamaların sayı qabaqcadan məlum deyilsə, onda **do-while** dövr operatorundan istifadə edilir;
- Dövr başlıqdan (**do**), gövdədən və həlledici şərtdən ibarətdir;
- **while** sözündən sonra mötərizələrin içində şərt yazılır. Hələ ki şərt doğrudur, dövr davam edir, əgər şərt yalan olursa, dövr öz işini dayandırır;
- şərt dövrün sonunda yoxlanılır, yəni **dövr heç olmasa bir dəfə təkrarlanır**.
- Əgər şərt heç vaxt yalan olmursa, onda dövr sonsuz davam edir. Bu ciddi məntiqi səhvdir.
- Dövrün gövdəsi fiqur mötərizələr içində yazılır. Əgər gövdədə bir operator varsa, onda mötərizələr lazım deyil.
- Dövrün gövdəsində digər operatorlar, həmçinin dövrlər də ola bilər.
- Proqramı asanlıqla başa düşmək üçün dövrün gövdəsi mötərizələrlə birlikdə 2-3 simvol sağa sürüşdürülür.

Dövrün vaxtından tez dayandırılması

Bəzən dövrün sonunu gözləmədən onu vaxtından tez dayandırmaq lazım gəlir. Bunun üçün xüsusi **break** operatorundan istifadə edirlər. Bəzən də dövrdən çıxmayaraq cari dövrü dayandırıb, növbəti addıma keçmək lazım gəlir. Bunun üçün **continue** operatorundan istifadə edirlər.

Misal. Klaviaturadan 2 tam ədəd daxil etməklə və bu iki ədədin bir-birinə bölməklə qalıq və qisməti hesablayıb ekrana çıxartmaq tələb olunur. Proqram dövrdə işləməlidir, yəni 2 ədədi daxil edib, hesabatı aparıb, nəticəni ekrana çıxardıb, yenə də ədədləri daxil etməlidir və s. Əgər hər iki ədəd sıfıra bərabərdirsə, onda proqramı dayandırmaq lazımdır. Əgər yalnız 2-ci ədəd sıfıra bərabərdirsə, onda proqram səhv haqqında məlumat verməlidir.

Bu məsələnin incəliyi ondan ibarətdir ki, dövrə daxil olarkən bilmək olmur cari sikl axira kimi yerinə yetiriləcəkdir, ya yox. Lazımı informasiya klaviaturadan verilənləri daxil edərkən əldə olunur. Ona görə də bu məsələdə sonsuz dövrdən istifadə olunur: **while (1) { … }** (bildiyiniz kimi, C dilində **1** doğru şərt sayılır). Bu cürə dövrdən çıxmaq üçün xüsusi **break** operatorundan istifadə edilir.

Eyni zamanda, əgər ikinci ədəd sıfır olarsa, onda dövrün qalan hissəsini yerinə yetirmək lazım deyil. Bundan ötrü **continue** operatoru nəzərdə tutulub.

```
#include <stdio.h>
#include <conio.h>
main ()
{
int A, B;
while ( 1 )                    // sonsuz dövr
   {
    printf ( "\nIki ededi daxil edin :" );
    scanf ( "%d%d", &A, &B );
    if ( A == 0 && B == 0 ) break;   // dövrdən çıxış
    if ( B == 0 )
       { printf ( "Sifira bolme! " );
        continue ;     // vaxtından tez dövrün növbəti addımına keçid
        }
    printf ( "Qismet %d   qaliq %d", A/B, A%B );
    }
getch ();
}
```

- Əgər dövr fasiləsiz işləməlidir və müəyyən şərt əsasında öz işini dayandırmalıdır, onda daxilində **break** operatoru olan **sonsuz** dövrdən istifadə etmək lazımdır.

```
while ( 1 ) {
...
if ( cixis_uchun_shert) break;
....
}
```

- **break** operatoru vasitəsi ilə istənilən dövrlərdən çıxmaq olar: **for, while, do-while**.
- Dövrün hər hansı addımını vaxtından tez dayandırıb növbəti addıma keçmək üçün **continue** operatorundan istifadə edilir.

Sıraların cəminin hesablanması

Sonlu elementdən ibarət cəmlər

Misal. Birinci 20 elementin cəmini hesablayın:

$$S = \frac{1}{2} - \frac{2}{4} + \frac{3}{8} - \frac{4}{16} + \cdots$$

Bu məsələni həll etmək üçün elementlərin dəyişmə qaydasını təyin etmək lazımdır. Misala baxaraq qeyd etmək olar ki:

- hər element kəsrdir;
- növbəti elementə keçərək surət bir vahid artır;
- növbəti elementə keçərək məxrəc 2 dəfə artır;

- kəsrlərin önündə yazılan işarələr növbələşdirilir (+, -, +, və s.)

Sıranın istənilən elementini aşağıdakı kimi təsvir etmək olar:

$$a_i = \frac{zc}{d}$$

Burada **z, c** və **d** dəyişənlərin qiymətləri (birinci 5 element üçün) aşağıdakı cədvələ əsasən dəyişir:

i	**1**	**2**	**3**	**4**	**5**
z	1	-1	1	-1	1
c	1	2	3	4	5
d	2	4	8	16	32

z dəyişəninin işarəsi dəyişir. Bu əməliyyatı **z=-z** kimi yazmaq olar. **c** dəyişənin qiyməti bir vahid artır, yəni **c++** yazılmalıdır. **d** dəyişəni 2-ya vurulur, yəni **d=d*2**. Bu məsələnin həlli alqoritmini aşağıdakı addımlar kimi yazmaq olar:

- **S** dəyişənə sıfır qiyməti vermək; bu xanada cəm saxlanılacaqdır;
- z, c və d dəyişənlərə başlanğıc qiymətləri (birinci element üçün) yazmaq:

  z=1, c=1, d=2.
- 20 dəfə təkrarlamaq:
  - cəmə növbəti elementin qiymətini əlavə etmək;
  - növbəti element üçün **z, c** və **d** dəyişənlərin qiymətlərini dəyişmək.

```
#include <stdio.h>
main ()
{
float S, z, c, d;
int i;
S=0; z=1; c=1; d=2;   // başlanğıc qiymətlər
for ( i = 1; I <=20; i ++ )
   {
   S = S + z*c/d;        // elementi cəmə əlavə etmək
   z = - z;              // z, c, d dəyişənlərin qiymətlərinin dəyişdirilməsi
   c ++;
   d = d * 2;
   }
printf ( “Cem S= %f”, S );
}
```

Məhdudlaşdırıcı şərt əsasında sıraların hesablanması

Elementlərin sayı qabaqcadan məlum olmayan daha mürəkkəb məsələyə baxaq.

Misal. Mütləq qiyməti 0.001-dən aşağı olmayan sıra elementlərinin cəmini təyin etmək:

$$S = \frac{1}{2} - \frac{2}{4} + \frac{3}{8} - \frac{4}{16} + \cdots$$

Bu məsələni o vaxt həll etmək olar ki, nə vaxt sıra elementlərinin mütləq qiymətləri azalmaya doğru dəyişsin və sıfıra yaxınlaşsın. Belə ki, bilinmir neçə element cəmə daxil

olacaqdır, ona görə **while** (və ya **do-while**) dövr operatorundan istifadə etmək lazımdır. Məsələnin həllinin bir variantı aşağıda göstərilmişdir:

```
 #include <stdlib.h>
main ()
{
float S, z, c, d, a ;
S = 0; z = 1; c = 1; d = 2 ;    // başlanğıc qiymətlər
a = 1 ;                         // 0.001-dən böyük istənilən ədəd
while ( a >= 0.001 )
  {
  a = c/d ;                     // elementin mütləq qiymətinin hesablanması
  S = S + z*a;                  // elementi cəmə əlavə edilməsi
  z = - z;                      // z, c, d dəyişənlərinin hesablanması
  c ++;
  d = d * 2;
  }
printf ( "Cem S =  %f", S );
}
```

a-nın qiyməti 0.001-dən aşağı olduğu zaman dövr öz işini bitirəcəkdir. Dövrü başlamaq üçün a-ya 0.001-dən böyük istənilən qiymət yazıla bilər.

5. PROQRAMLARIN DÜZƏNNƏMƏSİ METODLARI

Dev-C++ proqramının düzənnəmə vasitələri

Düzənnəmə nədir?

"Düzənnəmə" sözü "proqramda səhvlərin axtarışı və düzəldilməsi" deməkdir. İngilis dilində bu söz *debugging* (hərfi-tərcümədə "böcəklərin çıxarılması") deməkdir. Əfsanəyə görə 1940-ci ildə Mark II kompüterin kontaktlarına böcək (güvə) düşmüş, və bu böcəyə görə kompüter sıradan çıxmışdır.

Proqramlarda üç növ xətalar olur:

- *sintaktik xətalar* – operatorların səhv yazılışı (məsələn, **printf** operatorun əvəzinə **print** operatorun yazılışı); bu xətaları asanlıqla düzəltmək olur, çünki onları translyator təyin edir, hətta səhv olduğu sətri də göstərir;
- *yerinə yetirilmə zamanı ortaya çıxan xətalar* – proqramın icrası zamanı hər hansı "qəza" halı, məsələn, sıfıra bölmə;
- *məntiqi xətalar* – alqoritmdəki xətalar (proqram işləyir, lakin səhv nəticə verir). Bu xətaları üzə çıxartmaq çox çətindir.

Beləliklə, düzənnəmə zamanı ən çətin məsələ - səhv yazılmış operatoru təyin etməkdir. Təəssüf ki, bu məsələni avtomatlaşdırmaq olmur, onu yalnız insan həll edə bilər. Bu işi asanlaşdırmaq üçün "**debugger**" adlanan xüsusi proqramları icad ediblər.

Təqib etmə (ing. *traccing*)

Proqramın icra olunması zamanı siqnal xarakterli mesajların proqramın müəyyən nöqtələrində ekrana çıxarılması **təqib etmə (traccing)** və ya **izləmə** adlanır. Təqib etmə nə üçün istifadə edilir?

Birincisi, ekranda bu cürə mesajın çıxarılması bu o deməkdir ki, proqram bu nöqtəyə qəlib çatıb (vaxtından tez işini dayandırmayıb və ya sonsuz dövrə düşməyib). İkincisi, bu mesajlarda nəinki mətn, hətta dəyişənlərin qiymətlərini əks etdirmək olar. Bu isə proqramın düzgün gedişinə nəzarət etməyə imkan yaradır. Əgər, məsələn, 2-ci yoxlama nöqtəsində bütün dəyişənlərin qiymətləri düzgündürsə, 3-cü nöqtədə isə yox, onda xətanı bu iki nöqtə arasında axtarmaq lazımdır.

Aşağıdakı proqramda 3 nöqtədə **traccing** (əlavə çap operatorları) operatorları yerləşdirilib.

```
main ()
{
int i, X;
printf ( "Tam ededi daxil edin: \n" );
scanf ( "%d", &X ) ;
printf ( "Daxil edilmish eded = %d\n ", X);      // 1-ci nöqtə
for ( i = 1 ; i<10; i++ )
   {
    printf ( " Dovrde: i= %d, X=%d\n", i, X ) ;   // 2-ci nöqtə
      .....
```

```
      }
printf ( " Dovrden sonra: X= %d\n", X );        //3-cü nöqtə
...
}
```

1-ci nöqtədə ekrana **X** dəyişənində yerləşən qiymət çıxarılacaqdır. Bu qiymət klaviaturadan daxil edilmiş qiymətlə üst-üstə düşməlidir.

2-ci nöqtədə dövrün hər addımında alınmış qiymətlər ekrana çıxarılır. Bu isə dövrün düz işləməsi haqqında xəbər verir.

3-cü nöqtədə dövr bitdikdən sonra X-in qiyməti ekrana çıxarılır. Bu nöqtə də nəzarəti artırır.

Qeyd edək ki, *traccing* üçün çap operatorları proqramın istənilən yerində və istənilən zaman yerləşdirmək olar. Bunun üçün *debugger* proqramı lazım deyil.

Proqramın bir hissəsinin dayandırılması

Bəzən belə hallara rast gəlmək olur ki, proqramı təkmilləşdirəndən sonra işləyən proqram artıq işləmir. Bu vəziyyətdə xətanı aşkar etmək üçün proqramda yeni əlavə olunmuş kodu silib, sonra onu hissə-hissə əlavə etmək lazımdır. Operatorları əlavə edərkən "proqramı sıradan çıxardan" operatoru asanlıqla aşkar etmək olacaqdır. Proqramdan yeni kodu silməsək də olar, sadəcə onu şərh kimi qeyd etmək olar.

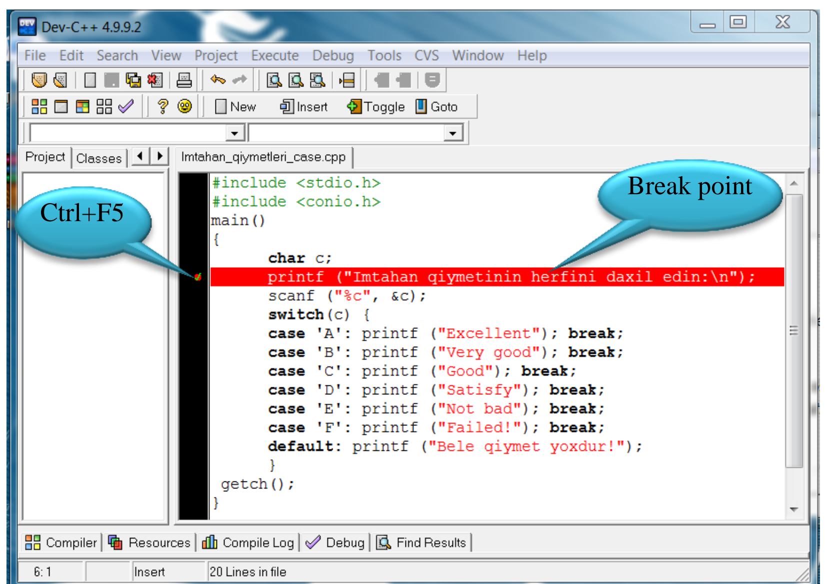
Proqramın bir sətrinin söndürülməsi üçün sətrinin qabağına **//** simvolları qoymaq lazımdır. Bir neçə operatordan ibarət blokun söndürülməsi üçün çox sətirli şərhdən istifadə edilir. Çoxsətirli şərh **/*** simvollarından başlayır ***/** simvolları ilə bitirilir.

```
main ()
{
 int i, X ;
printf ( "Ededi daxil edin: \n" ) ;
scanf ( "%d", &X );
// X *=X+2 ;
 for (i=1; i<10; i++ ) X *=i;
   /* while ( X > 5 )  {
       i = i * X;
       }    */
....
}
```

"Step by step" yerinə yetirmə

Adətən, proqram əvvəldən axıra kimi dayanmadan yerinə yetirilir. Düzənmənin ən yaxşı yolu – hər sətrin sonunda dayanıb, yaddaşa yazılan dəyişənlərin qiymətlərini yoxlayaraq proqramı addım-addım (step by step) yerinə yetirməkdir. Bu məqsəd üçün **debugger** adlanan xüsusi proqramlar nəzərdə tutulub.

DevC++ proqram təminatı **GDB debugger** ilə təchiz olunub. Əvvəlcə *dayanma nöqtələri (**break point**)*, yəni proqramın dayanması üçün lazımı sətirlər qeyd olunur. Bunun üçün lazımı sətrin sol tərəfində qara fonda mausla vurmaq lazımdır. Təkrar vurma *dayanma nöqtəni* götürür. Ctrl+F5 düymələrin kombinasiyasından istifadə edərək həmin əməliyyatı yerinə yetirmək olar.



Əgər proqramda heç olmasa bir dayanma nöqtəsi varsa, onda **F8** düyməsini vurmaqla bu proqramı düzənnəmə rejiminə (debugger) keçirmək olar. Debugger proqramı ən birinci **break point** nöqtəsində dayanacaqdır. Bundan sonra **F7** düyməsini sıxmaqla proqramı addım-addım (step by step) yerinə yetirmək olar.

Bu rejimdə proqrama daxil olunan proseduralara daxil olmaq olmur (yalnız əsas proqram yerinə yetirilir). Prosedura və ya funksiyaya daxil olmaq üçün **Shift+F7** kombinasiyasından istifadə olunur.

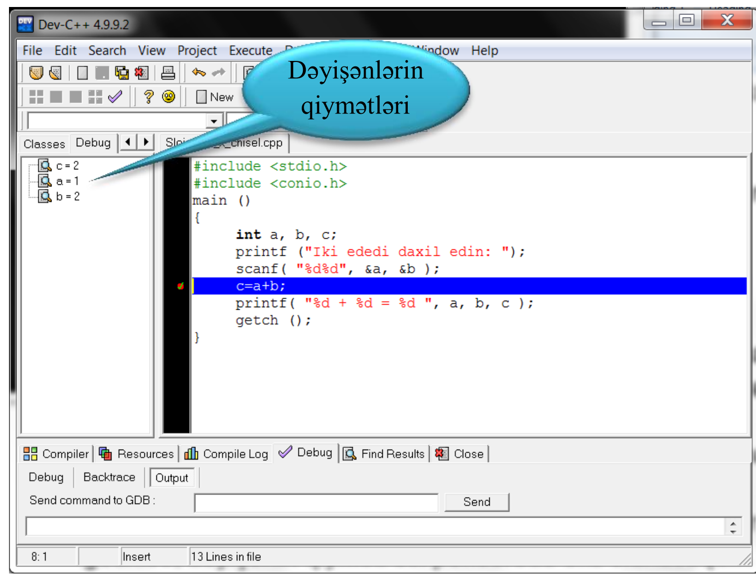
Ctrl+F7 kombinasiyasından istifadə edərək proqram növbəti **break point** nöqtəsinə qədər yerinə yetirilir.

Dəyişənlərin qiymətlərinə baxış

Proqramın addım-addım yerinə yetirilməsi zamanı hansı operatorlar, neçə dəfə və hansı ardıcıllıqda yerinə yetirilirsə - məhz bu məlumatları əldə etmək olar. Lakin çox vaxt proqramda xətanı təyin etmək üçün bu kifayət olmur.

Proqramın icrası zamanı dəyişənlərin qiymətlərinə baxış aləti xətaların təyini üçün çox güclü vasitədir.

Əgər debugger rejimində mausun göstəricisini hər hansı bir dəyişənin üzərinə gətirilsə, dəyişənin qiyməti sol pəncərədə əks olunacaqdır.



Proqramın əl ilə icra edilməsi

Əgər proqramda xətaları aşkar etmək üçün yuxarıda göstərilmiş üsullar kömək edə bilmirsə, onda nəticələri qeyd edərək proqramı kağız üzərində yerinə yetirmək lazımdır.

Adətən, bir cədvəl qurulur və dəyişənlərin bütün qiymətləri bu cədvələ yazılır. Dəyişənlərin məlum olmayan qiymətlərini “**?**” işarəsi ilə ifadə edirlər.

Daxil edilmiş natural ədədin sadə və ya mürəkkəb olmasını təyin edən proqramı (proqram səhv işləyir!) nəzərdən keçirək. **N=5** ədədi üçün proqram “eded murekkebdir” cavabi verir. Bu isə səhvdir. Cədvəli qurub xətanı aşkar etməyə çalışaq.

```
#include <stdio.h>
main ()
{   int N, i, count = 0 ;
    printf ( “Ededi daxil edin: “) ;
   scanf ( “%d”, &N );
   for (i=2; i <=N; i ++ )
     if  ( N % i == 0)   count ++;
  if ( count ==0 )
        printf ( “Eded sadedir” );
  else printf ( “Eded murekkebdir” );
 }
```

N	i	count
5	?	0
	2	
	3	
	4	
	5	1

Dəyişənlərin qiymətlərini bir-bir dəyişərək, araşdırmaq olur ki, ədəd özü-özünə bölünür və nəticədə **count** dəyişəninin qiyməti sıfırdan fərqli olur. İndi isə, xətanın səbəbini araşdırandan sonra səhvi düzəltmək asan olur. Bunun üçün dövrdəki şərt belə dəyişməlidir: **i<N.**

Sərhəd qiymətlərinin yoxlanılması

Proqramı və ya funksiyanı sınaqdan keçirərkən onun işini giriş verilənlərin dəyişmə diapazonunun sərhədlərində yoxlamaq lazımdır. Məsələn, ədədlərin sadə və ya mürəkkəb olmasını təyin edən proqram üçün bu sərhəd qiymət N=2-dir (1 nə sadə, nə də

mürəkkəbdir). Yəni, proqramı sınaqdan keçirən zaman giriş dəyişənlərin sərhəd qiymətləri yoxlanılmalıdır.

6. QRAFİKİ REJİM

Sadə qrafiki proqram

C dilində qrafiki proqram “sendviç” strukturuna bənzəyir.

qrafiki rejimin qoşulması

şəkillərin çəkilməsi

qrafiki rejimin söndürülməsi

Şəkillərin çəkilməsi üçün xüsusi pəncərə açan sadə bir proqramı tərtib edək:

```
#include <graphics.h>
#include <conio.h>
main ()
{
initwindow ( 400, 300 ); //  400x300 ölçüdə qrafiki pəncərənin açılması
// ....  burada şəkil çəkmək olar
getch ();
closegraph ();    //  pəncərəni bağla
}
```

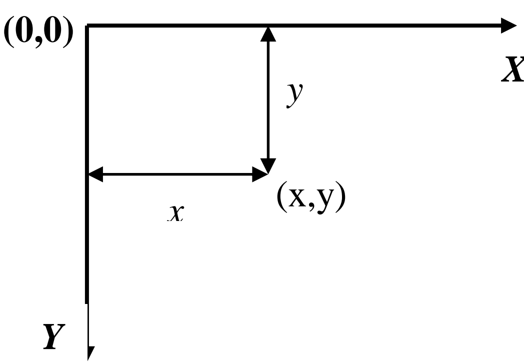
- Qrafiki funksiyalardan istifadə etmək üçün **graphics.h** başlıq faylını qoşmaq lazımdır.
- **initwindow** funksiyası şəkil çəkmək üçün əlavə pəncərə açır. Mötərizədə piksellərlə pəncərənin enini və hündürlüyünü göstərmək lazımdır.
- **closegraph** funksiyasının köməyi ilə qrafiki rejim dayandırılır.

Şəkil çəkməyə necə başlamaq lazımdır?

Nöqtələrin koordinatları

Ekranda şəkil çəkmək üçün koordinatları təyin etməyə bacarmaq lazımdır.

- Koordinatların başlanğıcı, yəni (0,0) nöqtəsi pəncərənin sol yuxarı küncündə yerləşir.
- Riyazi koordinat sistemindən fərqli olaraq **X** oxu sağa, **Y** oxu isə aşağı istiqamətləndirilib.
- İstənilən nöqtə üçün **x** koordinatı – pəncərənin sol, **y** koordinatı isə yuxarı sərhədinə kimi məsafədir.



Rəng

16 standart rəng üçün ədədi və simvolik işarələr təyin olunub:

0	BLACK	qara	8	DARKGRAY	tünd boz
1	BLUE	göy	9	LIGHTBLUE	açıq göy
2	GREEN	yaşıl	10	LIGHTGREEN	açıq yaşıl
3	CYAN	mavi	11	LIGHTCYAN	açıq mavi
4	RED	qırmızı	12	LIGHTRED	açıq qırmızı
5	MAGENTA	bənövşəyi	13	LIGHTMAGENTA	açıq bənövşəyi
6	BROWN	qəhvəyi	14	YELLOW	sarı
7	LIGHTGRAY	açıq boz	15	WHITE	ağ

Bundan əlavə, bütün rəng palitrasından istifadə etmək olar (*True Color* rejimi, təbii rəng). Bu halda istənilən rəng 3 komponentdən ibarət olur: qirmızı (**R**), yaşıl (**G**) və göy (**B**). Rəngin hər komponenti 0 – 255 aralığında dəyişən tam ədəddir (cəmi 256 variant). Beləliklə RBG kombinasiyası üçün 256^3=16777216 rəng alınır. Rəngləri vermək üçün **COLOR** funksiyasından istifadə edilir. Bu funksiyanın 3 parametri var. Onlar **R, G, B** rənglərinin intensivliyini təyin edirlər və mötərizədə vergüllə ayrılmış yazılırlar. Məsələn, bəzi rəngləri belə təyin etmək olar:

```
COLOR (0,0,0)            qara
COLOR (255,0,0)          qırmızı
COLOR (0, 255, 0)        yaşıl
COLOR (0, 0, 255)        göy
COLOR (255, 255, 255)    ağ
COLOR (100, 100, 100)    boz
COLOR (255, 0, 255)      bənövşəyi
```

setcolor funksiyası xətlərin rəngini təyin edir:

```
setcolor ( 10 ) ;  // açıq yaşıl rəngi təyin etmək
```

Bu əmrdən sonra çəkilən bütün xətlər, düzbucaqlar, çevrələr və s. açıq yaşıl rəngli olacaqdır. Bu funksiya rəngi rəng palitrasından da götürə bilər:

```
setcolor ( COLOR(255, 0, 255) ) ;   // bənövşəyi rəngi təyin etmək
```

Şəkil çəkmək üçün açılmış pəncərə ağ rəngdə olur.

Ayrı-ayrı piksellərlə işləmə

Şəkil çəkmək üçün standart funksiyalardan istifadə edilir. **putpixel** funksiyasından istifadə edərək hər piksel üçün onun rəngini vermək olar:

```
putpixel ( x, y, 14 );    // (x,y) nöqtəsini sarı rəngləmək
```

getpixel funksiyası vasitəsi ilə pəncərədə yerləşən istənilən pikselin rəngini təyin etmək olar.

```
n = getpixel ( x, y);   // (x,y) nöqtəsinin rəngini n dəyişənə yazmaq
```

Xətlər

Parçanı **line** əmri vasitəsi ilə çəkmək olar:

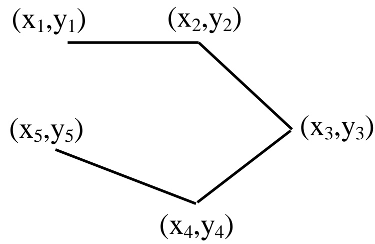
```
line ( x1, y1, x2, y2 );    // (x1,y1)-(x2,y2) parçası
```

Parçanı başqa üsulla çəkmək olar. Əvvəlcə **moveto** əmrindən istifadə edərək kursoru **(x1,y1)** nöqtəsinə gətirmək, sonra isə **lineto** əmri vasitəsi ilə **(x2,y2)** nöqtəsinə qədər parça çəkmək:

```
moveto ( x1, y1 ) ; //    kursor (x1,y1) nöqtəsinə gətirilir
lineto (x2, y2 ) ; //    (x2,y2) nöqtəyə qədər xətt çəkilir
```

lineto əmrindən sonra kursor növbəti (x2,y2) nöqtəsinə keçir. Qırıq xətlərin çəkilməsində bu əmrdən istifadə etmək daha əlverişlidir:

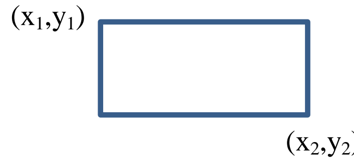
```
setcolor (12) ;  // qırmızı rəng
moveto (x1, y1) ; // kursoru (x1,y1) nöqtəsinə gətir
lineto (x2, y2 );  // 2-ci nöqtəyə qədər parça çəkilir
lineto (x3, y3 );  // 3-cü nöqtəyə qədər parça çəkilir
lineto (x4, y4 );  // .....
lineto (x5, y5);
```



Düzbucaqlar

Düzbucağı çəkmək üçün iki qarşı bucağın koordinatlarını vermək lazımdır (adətən sol yuxarı və sağ aşağı bucaqlar seçilir). Konturun rəngi **setcolor** funksiya vasitəsi ilə təyin olunur, düzbucağın özü isə **rectangle** əmri vasitəsi ilə çəkilir:

```
setcolor ( 9) ;
rectangle (x1, y1, x2, y2);
```



Rəngli düzbucağı çəkmək üçün **bar** əmrindən istifadə edilir. Rəngi və rəng çəkilməsinin üslubunu (ing. fill style) təyin etmək üçün **setfillstyle** funksiyasından istifadə etmək lazımdır.

```
setfillstyle ( 1, 12 );   // 1-ci üslub, 12-ci rəng
bar (x1, y1, x2, y2);
```

setfillstyle funksiyasının 2 parametri var. Birinci parametr rəng çəkmə üslubunu, ikinci parametr isə rəngi təyin edir.

0		rəng yoxdur
1		bircinsli rəng
3,4,5,6		əyri xətlər
7,8		tor
9,10,11		nöqtəli naxışlar

Çevrə

Çevrəni çəkmək üçün **circle** funksiyasından istifadə edirlər:

```
setcolor ( COLOR (0, 255, 0) );    // yaşıl rəng
circle (x, y, R );
```

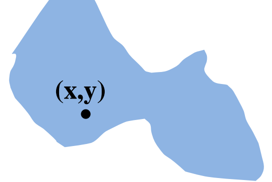
circle funksiyasının üç parametri var: birinci iki parametr mərkəzin koordinatlarıdır, üçüncü parametr isə - radiusdur. Parametrlər kimi ədədlər, dəyişənlərin adları və ya cəbri ifadələr ola bilər:

```
circle (200, y0+25, R );
```

İxtiyari oblastın rənglənməsi

Bəzən ixtiyari çəkilmiş fiquru rəngləmək lazım gəlir. Bunu **floodfill** funksiyası vasitəsi ilə etmək olar:

```
setfillstyle ( 1 , 11 );  // üslub 1, rəng 11
floodfill (x, y, 0);    // 0 rənginin sərhədinə kimi rəngləmək
```

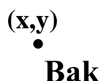


Fiquru rəngləmək üçün onun daxilində istənilən nöqtənin **(x,y)** koordinatlarını bilmək lazımdır. Bundan başqa, fiqurun sərhədi bir rəngli və kəsilməz olmalıdır. Sərhədin rəngi **floodfill** funksiyasının axırıncı parametridir.

Yazılar

outtextxy funksiyası pəncərənin istənilən yerində yazını çıxartmağa imkan verir. Bu funksiyaya yazının yuxarı sol küncünün (x,y) koordinatlarını vermək lazımdır. Yazının rəngi **setcolor** funksiyası ilə tənzimlənir:

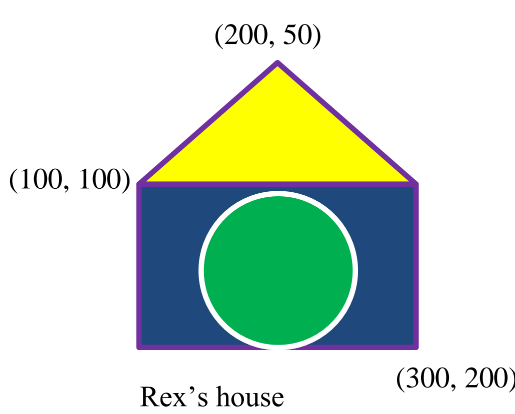
```
setcolor ( 9 );
outtextxy (x, y, “Bakı” );
```



Proqram nümunəsi

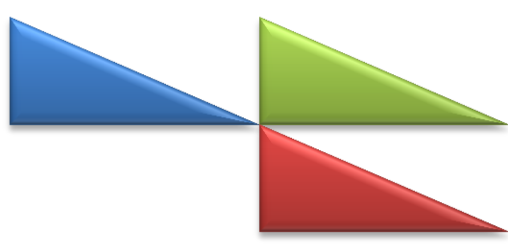
Standart qrafiki funksiyalardan istifadə edərək evcik çəkən bir proqram yazaq.

```
#include <graphics.h>
#include <conio.h>
main ()
{
    initwindow (440, 300);
    setfillstyle (1,9);
    bar (100, 100, 300, 200);   // göy düzbucaq
    setcolor (13);             // bənövşəyi kontur
    rectangle (100, 100, 300, 200);
    moveto (100, 100);        // damın çəkilməsi
    lineto (200,50);
    lineto (300, 100);
    setfillstyle (1,14);          // dam sarı rəngdə
    floodfill (200, 75, 13);
    setcolor (15);
    circle (200, 150, 50);      // ağ çevrə
    setfillstyle (1, 10);
    floodfill (200, 150, 150);   // çevrənin içi yaşıl
    setcolor (12);
    outtextxy (100, 230, "Rex's house");
 getch();
 closegraph();
 }
```



7. PROSEDURALAR

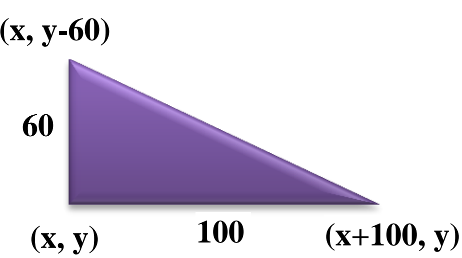
Proseduralı məsələnin nümunəsi



Proqramlarda tez-tez təkrar olunan obyektlərə rast gəlmək olur. Məsələn, şəkildəki 3 eyni üçbucaq. Ekranda fərqli rənglərlə eyni üç üçbucağı çəkən proqramı tərtib edək.

Aydındır ki, bu 3 üçbucağı ayrı-ayrı çəkmək olar, lakin onların oxşarlığından istifadə edərək bunu başqa cürə etmək olar.

Əvvəlcə verilmiş fiqurların **ümumi** (ölçülər, dönmə bucağı) və **fərqli** (rəng, koordinatlar) parametrlərini təyin etmək lazımdır. Əgər üçbucaqların ölçüləri və bucağı məlumdursa, onda onların çəkilməsi üçün üçbucağa məxsus olan bir nöqtənin (adətən, üçbucağın təpə nöqtələrindən biri) koordinatların qiymətləri bilinməlidir.



Tutaq ki, **(x,y)** – aşağı sol bucağın koordinatlarıdır. Üçbucağın eni və hündürlüyü, uyğun olaraq, 100 və 60-dır. Bu məlumatlara əsaslanaraq digər bucaqların koordinatlarını hesablamaq asandır: **(x, y-60)** və **(x+100, y)** (nəzərə alaq ki, **Y** oxu aşağı istiqamətləndirilib).

Üçbucağı çəkən **Tr** adlanan yeni əmr daxil edək. Bu əmrə aşağıdakı kimi müraciət olunacaqdır:

```
Tr ( x, y, c );
```

Burada c – üçbucağın rəngidir.

Kompüter (translyator) bu əmri tanımır və, ona görə də, onu yerinə yetirə bilməz. Deməli, bu əmri məlum olan əmrlər vasitəsi ilə təsvir etmək lazımdır. **Tr** əmri aşağıdakı kimi olmalıdır:

```
void Tr (int x, int y, int c )
{
   moveto (x, y );   // kursor aşağı kuncə qoyulur
   lineto (x, y-60);   // konturu çəkirik
   lineto (x+100, y);
   lineto (x, y) ;
   setfillstyle (1, c );    // rəngi və rəngləmə üslubu seçirik
   floodfill (x+20, y-20, 15 ); //  üçbucağı rəngləyirik
 }
```

Proqramlaşdırmada yeni əmrlərin yaradılması **alt proqramlar** və ya **proseduralar** adlanır.

> Proqramın bir-neçə yerində rast gələn müəyyən əməliyyatlarının yerinə yetirilməsi üçün nəzərdə tutulan əlavə proqram kodu (alt proqram) **prosedura** adlanır.

Proseduranın tərtibatı əsas proqramın tərtibatına oxşayır, lakin onun adı **main** yox, başqadır. Prosedura başlıqdan, fiqur mötərizələr içində yazılmış **gövdədən** ibarətdir. Proseduranın gövdəsində müraciət zamanı yerinə yetirilən əmrlər yazılır. Burada yalnız translyatora məlum olan əmrlərdən istifadə etmək olar.

Proseduranın başlığını nəzərdən keçirək:

```
void Tr (int x, int y, int c )
```

void sözü o deməkdir ki, bu prosedura müəyyən əməliyyatları yerinə yetirir (məsələn, nə isə çəkir) və heç nə hesablamır. Mötərizədə vergüllə ayrılmış proseduranın **parametrləri** yazılır.

Proseduranın yerinə yetirilməsi üçün lazım olan əlavə məlumatlar **parametrlər** adlanır.

Prosedura parametrsiz də ola bilər, lakin o həmişə eyni əməliyyatları yerinə yetirəcəkdir. Məsələn, ekranın konkret yerində həmişə göy üçbucaq çəkəcəkdir. Parametrlərin hesabına prosedura tərəfindən görünən iş fərqli olur. Məsələn, üçbucağın rəngi, yeri və ölçüləri dəyişə bilər.

Parametrlərə qiymətləri qabaqcadan məlum olmayan dəyişənlər daxildir. Onlar **formal parametrlər** adlanır. Proseduranın başlığında formal parametrlərin *tipi* və *adı* göstərilməlidir.

Proseduraya necə müraciət etmək olar? Tutaq ki, göy üçbucağın aşağı sol bucağını **(100, 100)** nöqtəsində yerləşdirmək lazımdır. Nəzərə alaraq ki, üçbucağın eni və hündürlüyü, uyğun olaraq, 100 və 60-dir, digər üçbucaqların koordinatlarını da hesablamaq olar: **(200, 100)** və **(200, 160)**. Yuxarıdakı şəkildə göstərilmiş 3 üçbucağı təsvir etmək üçün proqram aşağıdakı kimi olmalıdır:

```
#include <conio.h>
#include <graphics.h>
void Tr (int x, int y, int c )
{
   moveto (x, y );   // kursor aşağı kuncə qoyulur
   lineto (x, y-60);   // konturu çəkirik
   lineto (x+100, y);
   lineto (x, y) ;
   setfillstyle (1, c );    // rəngi və rəngləmə üslubu seçirik
   floodfill (x+20, y-20, 15 ); //  üçbucağı rəngləyirik
 }
main ()
{
  initwindow (400, 300) ;
  Tr (100, 100, COLOR(0, 0, 255)) ;
  Tr (200, 100, COLOR(0, 255, 0)) ;
  Tr (200, 160, COLOR(255, 0, 0)) ;
  getch () ;
  closegraph () ;
}
```

Yuxarıdakı proqramı nəzərdən keçirək. Prosedura əsas proqramdan əvvəl yazılır. O, əsas proqramın daxilində yox, ayrıca blok şəkilində tərtib olunur. Aşağıdakı sətir **proseduranın çağırışı** adlanır:

```
Tr (100, 100, COLOR(0, 0, 255)) ;
```

Proseduranın çağırışında mötərizələr içində bütün parametrlərin faktiki qiymətləri (**faktiki parametrlər**) göstərilmişdir. Bu müraciətə əsasən 100 qiyməti birinci **x** parametri, digər 100 qiyməti ikinci **y** parametri və s. əvəz edir. Müraciətdən əvvəl translyator **Tr** proseduranı tanımalıdır, yəni prosedura ona müraciətindən əvvəl təsvir olunmalıdır.

Digər iki üçbucağı çəkmək üçün proseduraya daxil olan bütün əmrləri təkrar yazmaq lazım deyil. Bunun üçün, proseduranın parametrlərini dəyişib ona iki dəfə də müraciət etmək lazımdır:

```
Tr (200, 100, COLOR(0, 255, 0)) ;
Tr (200, 160, COLOR(255, 0, 0)) ;
```

Qaydalar:

- Prosedura əsas proqram kimi tərtib olunmalıdır: başlıq və fiqur mötərizələrdə gövdə;
- Proseduranın adından qabaq **void** sözü yazılmalıdır. Bu o deməkdir ki, prosedura hesabat üçün yox, müəyyən əməliyyatların yerinə yetirilməsi üçün nəzərdə tutulub;
- Proseduranın adından sonra mötərizələrdə vergüllə ayrılmış onun **parametrləri** yazılmalıdır. Bu parametrlərdən onun işi asılıdır. Parametrlərə bəzən **arqumentlər** deyilir;
- Hər parametr üçün onun tipi göstərilməlidir (**int, float, char**);
- Parametrlərin adları kimi C dilində mümkün olan ixtiyari adlar ola bilər;
- Proseduranın başlığında yazılmış parametrlər **formal parametrlər** adlanır və bu parametrlərə yalnız proseduranın daxilindən müraciət etmək olar;
- Proseduraya müraciəti zamanı onun adı və mötərizələrdə **faktiki parametrlər** göstərilməlidir. Faktiki parametrlər formal parametrləri əvəz edirlər;
- Faktiki parametrlər ədədlər və ya istənilən cəbri ifadələr ola bilər. Əgər parametr ifadədirsə, onda əvvəlcə ifadənin qiyməti hesablanır, sonra isə alınmış qiymət parametr kimi proseduraya ötürülür;
- Birinci faktiki parametr birinci formal parametri, ikinci faktiki parametr – ikinci formal parametri və s. əvəz edir;
- Prosedura əsas proqramdan *əvvəl* elan olmalıdır. Bu ona görə lazımdır ki, translyator proseduranın çağırışına baxanda bilməlidir ki, proqramda belə prosedura var, onun neçə parametri var və onların tipləri necədir. Bu məlumatlar translyasiya zamanı xətaları aşkar etməyə kömək edir. Məsələn, səhvlər belə ola bilər:

Tr (100) ; **Too few arguments** (parametrlərin sayı azdır)

Tr (100, 100, 5, 5); **Too many arguments** (parametrlərin sayı çoxdur)

- Çox vaxt prosedurala bir dəfə müraciət olunur. Məqsəd - böyük mürəkkəb proqramı (və ya proseduranı) bir neçə müstəqil hissəyə ayırmaq. Bu ondan ötrüdür ki, böyük proqramlarda nə isə başa düşmək çox çətindir. Standarta əsasən hər proseduranın uzunluğu 50 sətirdən çox olmamalıdır;
- Proseduradan vaxtından tez çıxmaq üçün **return** operatorundan istifadə edilir. **return** operatoru icra olunanda prosedura öz işini bitirir;
- Prosedurada bir neçə **return** operatorundan istifadə etmək olar. Onların hər birisi proseduranın işini dayandırır.

8. FUNKSİYALAR

Funksiyaların proseduralardan fərqi

Alt proqramların digər növü – funksiyalardır. Proseduralar kimi funksiyalar proqramın təkrar olunan əməliyyatların yerinə yetirilməsi üçün nəzərdə tutulmuşdur. Fərq ondan ibarətdir ki, funksiya hər hansı nəticəni alaraq həmin nəticəni onu çağıran proqrama qaytarmalıdır.

Hər hansı nəticənin alınması ilə nəticələnən əlavə alt proqram **funksiya** adlanır.

Funksiyaların istifadəsini misalda göstərək.

Misal. Daxil edilmiş tam ədədin rəqəmlərinin cəmini hesablayan proqram tərtib etmək. Ədədin rəqəmlərinin cəmini hesablamaq üçün funksiyadan istifadə etmək.

Bildiyiniz kimi, ədədin axırıncı rəqəmini təyin etmək üçün onu 10-a bölərək qalığı təyin etmək lazımdır. Sonra axırıncı rəqəmi nəzərə almayaraq alınmış ədədi yenə də 10-a bölmək lazımdır və s. Alınmış rəqəmləri toplayaraq axtarılan cəmi alırıq.

```
#include <stdio.h>
#include <conio.h>
int SumDigits ( int N )     // funksiyanın başlığı
{                          //   funksiyanın əvvəli
int d, sum = 0;
while ( N !=0 )
    {
    d= N % 10;             //  funksiyanın gövdəsi
    sum = sum + d;
    N = N / 10;
    }
return sum;                //  funksiya sum qiymətini qaytarır
}                          //   funksiyanın sonu
main ()
{
int N, s;
printf ( “\nTam ededi daxil edin: “) ;
scanf ( “%d”, &N );
s = SumDigits (N);      //  funksiyaya müraciət
printf ( “%d ededinin reqemlerinin cemi beraberdir %d\n”, N, s );
getch ();
}
```

Qaydalar:

- Funksiya prosedura kimi tərtib edilir: funksiyanın başlığı və fiqur mötərizələr içində onun gövdəsi;

- Funksiya adından qabaq **nəticənin tipi** göstərilməlidir (**int, float, char**, və s.). Bu o deməkdir ki, o, göstərilmiş tip nəticə qaytaracaqdır.
- Funksiyanın adından sonra mötərizələr içində vergüllə ayrılmış onun parametrləri yazılır. Funksiyanın işi bu parametrlərdən asılıdır;
- Hər parametr üçün onun tipi (**int, float, char** və s.) göstərilməlidir;
- Parametrlərin adları kimi C dilində mümkün olan ixtiyari adlar ola bilər;
- Funksiyanın başlığında yazılmış parametrlər **formal parametrlər** adlanır. Bu o deməkdir ki, onlara, yalnız, funksiya daxilində müraciət etmək olar;
- Funksiyaya müraciəti zamanı onun adı və mötərizələrdə **faktiki parametrlər** göstərilməlidir. Faktiki parametrlər formal parametrləri əvəz edirlər;
- Faktiki parametrlər ədədlər və ya istənilən cəbri ifadələr ola bilər. Əgər parametr ifadədirsə, onda əvvəlcə ifadənin qiyməti hesablanır, sonra isə alınmış qiymət parametr kimi funksiyaya ötürülür;
- Birinci faktiki parametr birinci formal parametri, ikinci faktiki parametr – ikinci formal parametri və s. əvəz edir;
- Funksiyanın qiymətini təyin etmək üçün **return** operatorundan istifadə edilir. return operatorundan sonra qaytarılan qiymət (ədəd və ya cəbri ifadə) yazılmalıdır. Məsələn,

  ```
  return 34;
  return s;
  return a + 4*b – 5;
  ```

  return operatorundan sonra funksiya öz işini bitirir;
- Funksiyada bir neçə **return** operatorundan istifadə etmək olar.
- Əgər funksiyalar əsas proqramından sonra yazılılarsa, onda onları əsas proqramdan **əvvəl** elan etmək lazımdır. Funksiyanı elan etmək üçün axırda nöqtəli vergül qoyub onun başlığını yazmaq lazımdır;
- Funksiyanın elanında başlıqdan sonra mütləq nöqtəli vergül qoyulmalıdır, funksiyanın gövdəsi yazılan yerdə başlıqda **nöqtəli vergül qoyulmur!**

Məntiqi funksiyalar

Çox vaxt müəyyən məsələni həll edən və nəticə olaraq “Hə” və ya “Yox” cavabını verən funksiya tərtib etmək lazım olur. Bu funksiyalar **məntiqi funksiyalar** adlanır. Bildiyiniz kimi, C dilində 0 - “yalan” (false), 1 isə “doğru” (true) deməkdir.

> Cavab olaraq 1 (əgər cavab “hə”-dir) və ya 0 (əgər cavab “yox”-dur) qaytaran funksiya **məntiqi funksiya** adlanır.

Məntiqi funksiyaları, əsasən, aşağıdakı hallarda istifadə edirlər:

- hər hansı vəziyyəti araşdırıb, proqramın gedişinə təsir göstərən cavabı almaq;
- hər-hansı mürəkkəb əməliyyatları yerinə yetirib, səhvin olub olmamağını təyin etmək.

Ədəd sadədir, yoxsa yox?

Misal. **N** ədədi daxil edib onun sadə və ya mürəkkəb olmasını təyin etmək. Məsələnin həlli üçün funksiyadan istifadə etmək.

İndi isə funksiyanın gövdəsini əsas proqramdan sonra yerləşdirək. Funksiya haqqında translyatora xəbər vermək üçün onu qabaqcadan elan etmək lazımdır.

```
#include <stdlib.h>
#include <conio.h>
int Prime (int N );     // funksiyanın elanı
main ()
{
int N;
printf ( “\nTam ededi daxil edin: “ );
scanf ( “%d”, &N );
if ( Prime(N) )     //   funksiyaya müraciət
        printf ( “Eded %d – sadedir\n”, N );
else  printf ( “Eded %d – murekkebdir\n”, N );
getch ();
}
int Prime ( int N )   //   funksiyanın təsviri
{
for ( int i = 2; i*i < N; i ++ )
    if (N% i == 0 ) return 0;     //  bölən tapıldı –mürəkkəbdir!
return 1;     //   heç bir bölən tapılmadı – sadədir!
}
```

İki qiymət qaytaran funksiyalar

Tərifə əsasən, funksiya yalnız bir qiymət – nəticə qaytara bilər. Əgər iki və ya daha çox nəticə qaytarmaq lazımdırsa, onda xüsusi üsuldan istifadə edilir. Bu üsul **parametrlərin istinadla ötürülməsi** adlanır.

Misal. İki tam ədəddən ən böyük və ən kiçik ədədi təyin edən funksiyanı tərtib etmək.

Aşağıdakı funksiyada maraqlı üsuldan istifadə edilir: funksiya əsas proqrama məxsus olan dəyişənin qiymətini dəyişir. Nəticənin birini (iki ədədin kiçiyini) funksiya adi şəkildə, digər nəticəni isə - əsas proqramdan ötürülən dəyişənin qiymətini dəyişməklə qaytaracaq.

```
#include <stdlib.h>
#include <conio.h>
int MinMax (int a, int b, int &Max )
{
if ( a > b ) { Max = a; return b; }
else         { Max = b; return a; }
```

Nəticə - parametr

```
}
main ()
{
int N, M, min, max;
printf ( "\n Iki tam eded daxil edin: " );
scanf ( "%d%d", &N, &M );
min = MinMax ( N, M, max ) ;  // funksiyanın çağırışı
printf ( " Boyuk eded =  %d, Kicik eded = %d\n", max, min );
getch ();
}
```

Adətən, parametri proseduraya və ya funksiyaya ötürəndə, yaddaşda dəyişənin surəti (kopiyası) yaradılır, və funksiya bu surət ilə işləyir. Bu o deməkdir ki, funksiyada parametr üzərində aparılan dəyişiklər onun əsas proqramdakı qiymətinə təsir göstərməyəcək.

Əgər funksiyanın başlığında parametrinin adının əvvəlinə **&** işarəsi qoyularsa (bu işarə, həmçinin, dəyişənin ünvanının təyini üçün istifadə edilir), onda funksiya dəyişənin surəti ilə yox, onun yaddaşda yerləşən əsl qiyməti ilə işləyəcəkdir. Ona görə də, bizim proqramda funksiya **max** dəyişəninin qiymətini dəyişib ona iki ədədin ən böyüyünü yazacaqdır.

Baxmayaraq ki, prosedurlar əsas proqrama heç bir nəticə qaytarmırlar, bu üsulu prosedurlar üçün də istifadə etmək olardı.

Qaydalar:

- Əgər funksiya iki və daha çox qiymət qaytarmalıdırsa, onda nəticələrdən biri adi şəkildə, yəni **return** operatoru vasitəsi ilə, digər nəticələr isə istinadla ötürülmüş (dəyişdirilə bilən) parametrlər kimi qaytarılır;
- Belə ki funksiyalar parametrlərin *surətləri* ilə işləyirlər, adi parametrləri funksiya dəyişə bilməz (məsələn, əgər MinMax funksiyasında **a** və **b** parametrlərin qiymətləri dəyişərsə, əsas proqramda onlara uyğun N və M dəyişənlərin qiymətləri dəyişməyəcəkdir);
- İstənilən prosedura və ya funksiya nəticələri dəyişən parametr vasitəsilə qaytara bilər;
- Dəyişən parametrlər (istinadla ötürülən parametrlər) funksiyanın başlığında xüsusi şəkildə elan olunurlar: onların adlarının önündə **&** işarəsi qoyulur.
- Bu cürə funksiyaların çağırışı zamanı dəyişə bilən faktiki parametrinin əvəzinə dəyişənin adı yazılmalıdır. Burada **ədədlərdən və ya cəbri ifadələrdən istifadə etmək olmaz!** Bu zaman translyator xəbərdarlıq verəcəkdir.

9. PROQRAMIN STRUKTURU

Proqramın tərkib hissələri

C dilində yazılmış proqram bir neçə hissədən ibarətdir:

- *Başlıq faylların* qoşulması - **#include** sözü ilə başlayan sətirlər;
- *Sabitlərin (konstantaların) elanı:*

  const N = 20 ;
- *Qlobal dəyişənlər* – bu dəyişənlər əsas proqramdan kənar elan olunur. Proqramda bütün prosedurlar və funksiyalar qlobal dəyişənlərə müraciət edə bilərlər. Qlobal dəyişənləri hər prosedura və ya funksiyada təkrar elan etmək lazım deyil;
- *Funksiyaların və prosedurların elanı* – adətən, əsas proqramdan əvvəl olur. C dilinin standartına görə funksiyanın müraciəti zamanı o, artıq elan olunmuş və onun parametrlərin tipləri məlum olmalıdır;
- *Əsas proqram* - bütün alt proqramlardan əvvəl və ya sonra yerləşə bilər. Əsas proqramı funksiyaların arasına yerləşdirmək məsləhət görülmür.

Qlobal və lokal dəyişənlər

Qlobal dəyişənlər istənilən prosedura və ya funksiyadan əl çatandır. Ona görə də onları bütün alt proqramlardan əvvəl elan etmək lazımdır. Prosedurаlarda və funksiyalarda elan olunmuş digər dəyişənlər **lokal dəyişənlər** adlanır, çünki onlar, yalnız, elan olunduqları alt proqrama məlumdurlar. Lokal və qlobal dəyişənlər arasında olan fərqi aşağıdakı misalda görmək olar:

```
#include <stdio.h>
int var = 0;                    // qlobal dəyişənin elanı
void ProcNoChange ()
{
int var;                        // lokal dəyişən
var = 3;                  // lokal dəyişəninin qiymətinin dəyişdirilməsi
}
void ProcChange1 ()
{
var = 5;                  // qlobal dəyişəninin qiymətinin dəyişdirilməsi
}
void ProcChange2 ()
{
int var;                        //  lokal dəyişən
var = 4;                  //   lokal dəyişəninin qiymətinin dəyişdirilməsi
::var = ::var* 2 + var;   // qlobal dəyişəninin qiymətinin dəyişdirilməsi
}
main ()
{
ProcChange1 ();                 // var = 5;
```

```
ProcChange2 ();              // var=5*2+4=14;
ProcNoChange ();             // var dəyişmir
printf ( "%d", var );        // qlobal dəyişəninin çapı (14)
}
```

Qaydalar:

- alt proqramlarda qlobal dəyişənləri təkrar elan etmək lazım deyil;
- Əgər alt proqramda qlobal dəyişənlə eyni adla olan lokal dəyişən elan olunursa, onda alt proqramda lokal dəyişən istifadə edilir;
- Əgər lokal və qlobal dəyişənlərin adları üst-üstə düşürsə, onda qlobal dəyişənə müraciət etmək üçün onun adının qabağında 2 qoşa nöqtə qoyulmalıdır:

```
::var = ::var * 2 + var;
```

Mütəxəssislər qlobal dəyişənlərdən az (əlacsız halda) istifadə etməyə məsləhət görürlər, çünki qlobal dəyişənlər

- proqramın təhlilini və sınaqdan keçirilməsini çətinləşdirir;
- ciddi xətaların ehtimalını artırır;
- proqramın həcmini artır, çünki qlobal dəyişənlər verilənlər blokuna daxil olunmur, onlar yaddaşda daimi yer tuturlar.

Proqramın tərtibatı

Translyator üçün eyni sayılan proqramları müqayisə edək:

```
#include <stdio.h>
main ()
{
float x, y;
printf ( "\n2 ededi daxil edin: ");
scanf ( "%d%d", &x, &y );
printf ("Edelerin cemi = %d", x+y );
}
```

```
#include <stdio.h> main ()
{ float x, y; printf ("\n2 ededi
daxil edin: ");
scanf ( "%d%d", &x, &y );
printf ("Edelerin cemi = %d",
x+y ); }
```

Birinci proqram düzgün tərtib olunmuşdur.

> Proqramın tərkibində xətaları aşkar edib asanlıqla düzəltmək və, eyni zamanda, proqramdakı alqoritmi yaxşı başa düşmək üçün proqram düzgün (mükəmməl) tərtib olunmalıdır.

Funksiyaların və prosedurların tərtibatı

Funksiyaların və prosedurların tərtibatı zamanı aşağıdakı qaydalara riayət etmək məsləhətdir:

- Proqramın müxtəlif yerlərində yerləşən təkrar bloklar alt proqramlar kimi tərtib olunmalıdırlar;
- Funksiyaların və proseduraların adları informativ olmalıdır, yəni adından bilinməlidir bu proqram hansı əməliyyatları yerinə yetirir. Təəssüf ki, translyator azərbaycan dilində yazılmış sözləri başa düşmür, lakin latın əlifbasından istifadə edərək həmin sözləri yazmaq mümkündür. Məsələn, kvadratı çəkən proseduranı belə elan etmək olar:

```
void Square (int x, int y, int a );
```

və ya belə:

```
void Kvadrat (int x, int y, int a );
```

- Alt proqramının başlığının önündə bir neçə sətirli şərh yazılmalıdır. Şərhdə alt proqramının əsas məqsədi, parametrləri və nəticə haqqında məlumatlar öz əksini tapmalıdırlar.
- Alt proqramların uzunluğu 25-30 sətirdən çox olmamalıdır, çünki uzun proqramları başa düşmək çətindir. Əgər alt proqram uzun alınırsa, onda onu daha kiçik proseduralara və ya funksiyalara bölmək lazımdır.
- Alt proqramının bir hissəsini digərindən ayırmaq üçün boş sətirlərdən istifadə edilir. Daha iri blokları şərhdə bir sətir "-" işarəsi qoyub bir-birindən ayırmaq olar.

Düzgün tərtib olunmuş romb fiqurunu ekranda çəkən funksiyanı nəzərdən keçirək. Əgər romb ekrana yerləşirsə, onda funksiyanın nəticəsi 1, əks halda nəticə 0-dir.

```
//****************************************************************
// Romb  - verilmish movqede rombun chekilmesi
//      (x,y) – rombun merkezinin koordinatlari
//      a, b  -rombun eni ve hundurluyu
//      color, colorFill  - serhedin ve rombun rengi
//      Netice 1-dir, eger emeliyyat yerine yetirilib ve
//               0- dir, eger romb ekranda yerleshmir
//****************************************************************
int Romb (int x, int y, int a, int b, int color, int colorFill )
{
   if ( (x < a ) || (x > 640-a) || (y < b) || (y > 480-b) )
      return 0;
//----------------------------------------------------------------------
   setcolor ( color );
   line (x-a, y, x, y-b ); line (x-a, y, x, y+b);
   line (x+a, y, x, y-b ); line (x+a, y, x, y+b);

   setfillstyle (SOLID_FILL, colorFill );
   floodfill (x, y, color );

   return 1;
}
```

Başlıq

Səhvlərin emalı

İri blok

Boş sətir

Kənar boşluqlar

Proqramda struktur blokların (alt proqramlar, dövrlər, şərti operatorlar) seçilməsi üçün kənar boşluqlardan (ing. *tabs*) istifadə edilir. Boşluqların köməyi ilə proqramda çatışmayan və ya artıq qoyulmuş mötərizələri aşkar etmək, proqramın məntiqini başa düşüb və orada xətaları tapmaq asanlaşır. Kənar boşluqların qoyulması zamanı aşağıdakı qaydalara riayət etməyə məsləhətdir:

- kənar boşluğun miqdarı 2-4 simvoldur;
- Əlavə boşluqlarla aşağıdakıları seçmək olar:
    - **for, while, do–while** dövr operatorların gövdələrini;
    - **if** şərti operatorun və **else** blokunun gövdəsini;
    - **switch** operatorun gövdəsini;

Aşağıdakı proqramda kənar boşluqlardan istifadə edilib:

```
main ()
{
int a = 1, b = 4, i;
for (i=1; i<2; i++)
   {
 if ( a > b)
   {
   b = a + i;
   }
 else
   {
   a = b + i;
    }
 }
printf ( “%d %d\n”, a, b );
}
```

10. ANİMASİYA

Animasiya nədir?

Ekranda təsvirlərin canlandırılması **animasiya** (ing. animate) adlanır. Animasiya zamanı obyektlər hərəkət edir, fırlanır, toqquşur, rənglərini dəyişirlər və s.

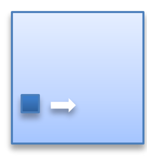
Proqramda animasiya effektlərindən istifadə etmək üçün iki məsələni həll etmək lazımdır:

- obyekt elə hərəkət etməlidir ki, titrəmələr olmasın;
- hərəkət zamanı idarəetmə klaviatura düymələri və ya mausla olmalıdır.

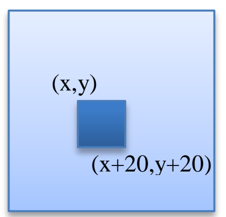
Proqramdan çıxış **Esc** düyməsi ilə olmalıdır.

Obyektin hərəkət etməsi

Ekranın sol tərəfindən sağ tərəfinə müəyyən obyekti (bizim misalda kvadratı) hərəkətə gətirən proqramı tərtib edək. Əgər obyekt ekrandan kənar çıxırsa və ya Esc düyməsi sıxılırsa, onda proqram öz işini bitirir.



İlkin təhlil



Ekranda hərəkət edən obyekti nəzərdən keçirək. Tutaq ki, bu tərəfləri 20 piksel olan kvadratdır. Hərəkət zamanı onun bütün nöqtələrinin koordinatları dəyişəcəkdir. Ekranın istənilən yerində kvadratı çəkmək üçün baza nöqtəsi kimi onun sol yuxarı küncü seçək və bu nöqtəni (x,y) kimi işarə edək. Qarşı küncün koordinatları (x+20, y+20) olacaqdır.

İndi isə hərəkət haqqında fikirləşmək lazımdır: hərəkət zamanı təsvir titrəməməlidir, və kompüterin sürətindən asılı olmayaraq, proqram bütün kompüterlərdə eyni nəticə göstərməlidir. Bunun üçün aşağıdakı alqoritmdən istifadə edirlər:

1. ekranda fiquru çəkirik;
2. bir balaca gecikmə veririk (10-20 ms);
3. fiquru silirik;
4. koordinatlarını dəyişirik;
5. birinci addıma keçirik.

Bu əməliyyatları "hərəkəti dayandır" (**Esc** düyməsinin sıxılması və ya fiqurun ekrandan kənar çıxması) əmrin alınmasına qədər təkrar etmək lazımdır.

Tutaq ki, kvadratın hərəkəti göy ekranda baş verir. Onda kvadratın silinməsi üçün ən sürətli və sadə üsul – onu göy rəngdə etməkdir. Ona görə də, proseduranın parametrləri **x, y** koordinatları və **color** (rəng) olmalıdır. Göy rəngdən istifadə zamanı fiqur ekrandan silinir.

```
void Draw ( int x, int y, int color )
{
```

```
setfillstyle (1, color );   // bircinsli color rəngi
bar (x, y, x+20, y+20 );    //  rəngli düzbucağın çəkilməsi
}
```

Alqoritmə daxil olan bütün əməliyyatları bir neçə dəfə yerinə yetirmək lazımdır, ona görə də dövrdən istifadə edək. Bundan əlavə, dövrün neçə dəfə təkrarlandığını bilmirik, ona görə də **while** dövr operatorundan istifadə edəcəyik.

Dövrdən çıxış şərti – fiqurun ekrandan kənar çıxması və ya **Esc** düyməsinin basılmasıdır. Tutaq ki, pəncərənin ölçüləri 400x400 piklseldır. Onda **x** koordinatı 0-dan 399-a kimi dəyişə bilər, yəni dövrün davamı üçün şərt belədir:

x + 20 < 400

x koordinatı bu şərtə uyğun gəlməyəndə kvadrat pəncərədən kənar çıxır və dövr dayanmalıdır.

Klaviatura ilə işləmə

Esc düyməsi ilə çıxmaq üçün müəyyən şərti yoxlamaq lazımdır. Həmin an obyekt hərəkətdə olur və ekranı **getch** funksiya vasitəsi ilə dayandırmaq olmaz. Bu zaman aşağıdakı alqoritmdən istifadə edilir:

1. Hər-hansı düymənin sıxılması yoxlanılır; Bu yoxlamanı **kbhit** funksiyası yerinə yetirir. Əgər heç bir düymə basılmayıbsa, onda funksiya 0 qaytarır ("yox" cavabı). Əgər hər hansı düymə sıxılıbsa, onda **kbhit** funksiyası sıfırdan fərqli qiymət qaytarır. Bu yoxlamanı şərti operator vasitəsi ilə həyata keçirmək olar:

   if (kbhit ()) { … }

2. Əgər hər hansı düymə sıxılıbsa, onda:

- **getch** funksiyası vasitəsi ilə sıxılmış düymənin kodunu təyin edirik. Düymənin kodu - simvollar cədvəlində onun nömrəsidir. Əgər simvola 1 bayt yer verilirsə, onda 256 müxtəlif simvollardan istifadə etmək olar, və onların kodları 0-dan 255-ə qədər dəyişəcəkdir.
- Əgər təyin edilmiş kod Esc düyməsinin koduna (27) bərabərdirsə, onda dövrdən çıxırıq.

Proqramı düymələrlə idarə etmək üçün onların kodlarını bilmək lazımdır. Onlardan bəziləri aşağıda göstərilmişdir:

Esc 27
Enter 13
Space 32

Yuxarıda verilmiş misalın proqramı aşağıdakı kimi olacaqdır:

```
#include <conio.h>
#include <graphics.h>
void Draw (int x, int y, int color )
        {setfillstyle ( 1, color );  //  bircinsli color rəngi
        bar (x, y, x+20, y+20 );  //   rəngli düzbucaq
        }
```

```
main ()
{
        int x, y ;                              //  kvadratın  koordinatları
        initwindow (400, 400);                  //  qrafika üçün pəncərənin açılması
        setfillstyle (1, COLOR( 0, 0, 255));    //  bircinsli göy rəngi
        bar ( 0, 0, 399, 399);                  //  pəncərənin rənglənməsi
        x = 0; y = 240;                         //  kvadratın başlanğıc koordinatları
        /*  animasiya  */
        closegraph ();                          //  qrafiki pəncərənin bağlanması
}
```

Kvadratı hərəkətə gətirən /* animasiya */ blokunu proqrama əlavə edək:

```
while ( x+20 < 400 )                          //      pəncərədən kənara çıxma şərtini yoxlanılması
        {
        if ( kbhit() )                              //  əgər hər hansı düymə sıxılıbsa ....
                if ( getch () == 27 ) break;        //  əgər bu düymə Esc-dirsə, onda çıxış
        Draw (x, y, COLOR (255, 255, 0));           //  sarı kvadratın çəkilməsi
        delay ( 20 );                               //  kvadratı görmək üçün gecikmə
        Draw (x, y, COLOR (0, 0, 255 ));            //  kvadratın silinməsi
        x++;                                        //  hərəkət etmə
        }
```

Fiqur ekran daxilində olduğuna qədər **while** dövr operatoru yerinə yetirilir. **Esc** düyməsinin sıxılması dövrdə emal olunur. Əvvəlcə **kbhit** funksiyası hər hansı düymənin basılmasını yoxlayır, sonra **getch()** funksiyası sıxılmış düymənin kodunu təyin edir. Əgər bu kod 27 bərabərdirsə (**Esc**-in kodu), onda **break** operatoru dövrü dayandırır.

Dövrün əsas hissəsində prosedura vasitəsi ilə fiquru çəkirik, sonra 20 ms gecikmə verəndən sonra (**delay** funksiyası), fiquru silirik. Ondan sonra **x** koordinatını dəyişib dövrün əvvəlinə qayıdırıq.

Qaydalar:

- Klaviatura düyməsinin basılmasını təyin etmək üçün **kbhit** funksiyasından istifadə edilir. Əgər heç bir düymə basılmayıbsa, funksiyanın qiyməti 0, əks halda sıfır olmayan bir qiymətə bərabərdir;
- Əgər klaviaturadan hər hansı düymə sıxılıbsa, onda onun kodunu **getch()** funksiyası vasitəsi ilə təyin etmək olar;
- İstənilən miqdarda gecikmə vermək üçün **delay** prosedurasından istifadə olunur. Bu proseduranın parametri milli saniyələrlə ifadə olunan gecikmədir. Əgər gecikmənin miqdarını azaltsaq, onda fiqur daha sürətli hərəkət edəcəkdir.

İstiqamət düymələri vasitəsi ilə idarəetmə

İstiqamət düymələrlə (→, ←, ↑, ↓) iş çox asandır. Hansı istiqamətli düymə sıxılırsa, həmin istiqamətdə obyekt hərəkət etməlidir.

Əgər dövrün hər addımında **x, y** koordinatların dəyişməsini **dx** və **dy** kimi işarə etsək, onda 4 istiqamətdə hərəkət etmək üçün şərtlər aşağıdakı kimi olmalıdır:

sola hərəkət: dx<0, dy=0
sağa hərəkət: dx>0, dy=0
yuxarıya doğru hərəkət: dx=0, dy<0
aşağıya doğru hərəkət: dx=0, dy>0

Bu o deməkdir ki, sıxılan düymədən asılı olaraq dörd variantdan birini seçmək lazımdır. Bunun üçün bir neçə **if** şərti operatorlardan istifadə etmək olar, lakin bir neçə variantlı seçimi təşkil etmək üçün xüsusi **switch** operatoru var.

Burada problem ondan ibarətdir ki, kursoru idarə edən düymələr adi düymələr deyil. Onlar funksional düymələr qrupuna aiddir və simvollar cədvəlində onların kodu yoxdur. Xüsusi düymələrdən biri sıxılanda sistem onu iki sıxılma kimi emal edir. Birinci sıxılmanın kodu həmişə sıfıra bərabərdir. İkinci sıxılmanın xüsusi və ya *scan kodu* (düymənin klaviaturada sıra nömrəsi) aşağıdakı cədvəldə göstərilmişdir:

→	77
←	75
↑	72
↓	80

Bu üsulun çatışmayan cəhəti ondan ibarətdir ki, obyekt kodları 75, 77, 72 və 80 olan (K, M, H və P hərfləri) düymələrin toxunuşuna da reaksiya verəcəkdir.

Sadə proqram

Fiquru yalnız istiqamət düymələri vasitəsi ilə idarə edən proqramı tərtib edək.

Əvvəlcə dövrdə fiquru çəkib, düymənin basılmasını gözləyirik və **getch** funksiyası vasitəsi ilə düymənin kodunu qəbul edirik. Sonra həmin yerdə fiquru silib, koddan asılı olaraq fiqurun koordinatlarını dəyişirik.

Burada sonsuz **while (1)** dövrdən istifadə olunmalıdır. Dövrdən yalnız **break** operatoru vasitəsi ilə çıxmaq olacaqdır.

```
while ( 1 )                              // sonsuz dövr
     {
     Draw (x, y, COLOR (255, 255, 0));   // kvadratın çəkilməsi
     code =  getch();                    // düymənin basilmasının gözlənilməsi
     if (code == 27 ) break;             // əgər Esc, onda çıxış
     Draw (x, y, COLOR (0,0, 255));      // kvadratın silinməsi
     switch ( code ) {                   // istiqamətin seçilməsi
          case 75: x --; break;          // sola
```

```
            case 77: x ++; break;          // sağa
            case 72: y --; break;          // yuxarı
            case 80: y ++;                 // aşağı
            }
    }
```

switch operatorunda koordinatlar 1 addımla dəyişirlər, qaldı ki istənilən addımdan istifadə etmək olardı. Hər variantın sonunda **break** operatoru yazılmalıdır ki, aşağıda yerləşən sətirlər yerinə yetirilməsin. Tam tipli **code** dəyişənini əsas proqramının əvvəlində elan etmək lazımdır.

Fasiləsiz hərəkət

İndi isə daha mürəkkəb məsələyə baxaq. Heç bir düymə sıxılmasa da obyekt seçilmiş istiqamətdə hərəkət edir, hər hansı düymə sıxıldıqda isə istiqamətini dəyişir. Burada istiqaməti təyin etmək üçün **dx** və **dy** dəyişənlərdən istifadə etmək lazımdır. Əvvəlcə düymənin basılmasını, sonra onun kodunu təyin edib **code** adlı dəyişənə yazırıq. **switch** operatoru vasitəsi ilə düymənin basılmasını emal edirik.

```
dx = 1; dy = 0;                              // əvvəlcə sağa hərəkət edir
while ( 1 )                                  // sonsuz dövr
    {
    if ( kbhit () ) {                        // əgər düymə sıxılıbsa
      code = getch ();                       // onun kodunu almaq
      if (code == 27 ) break;                // əgər Esc, onda çıxış
      switch ( code )   {                    // hərəkət istiqamətini dəyişmək
            case 75: dx = -1; dy = 0; break;
            case 77: dx  =1; dy = 0; break;
            case 72: dx = 0; dy = -1; break;
            case 80: dx = 0; dy = 1;
            }
        }
    Draw (x, y, COLOR (255, 255, 0));          // kvadratı çəkirik
    delay ( 10 );                              // gözləyirik
    Draw (x, y, COLOR (0, 0, 255) );           // silirik
    x += dx;                                   // kvadratı hərəkətə gətiririk
    y +=dy;
    }
```

11. TƏSADÜFİ VƏ PSEVDO TƏSADÜFİ ƏDƏDLƏR

Təsadüfi ədədlər nədir?

Təsəvvür edin ki, qar yağır. Tutaq ki, bu mənzərənin bir anını çəkirik. "Növbəti qar dənəsi hara düşəcək" suala dəqiq cavab vermək olarmı? Xeyr, çünki bu çoxlu sayda səbəblərdən asılıdır: hansı qar dənəciyi yerə daha yaxındır, külək necə olacaqdır və s. Demək olar ki, qar dənəciyi *təsadüfi yerə* düşəcəkdir, yəni onun yerini qabaqcadan söyləmək olmaz.

Kompüterdə təsadüfi proseslərin (qar dənəciklərin hərəkəti, molekulların Broun hərəkəti) modelləşdirilməsi üçün **təsadüfi ədədlərdən** istifadə edirlər.

Əvvəlki ədədləri bilərək növbəti ədədi təyin etməyə mümkün olmayan ədədlər ardıcıllığı **təsadüfi ədədlər** adlanır.

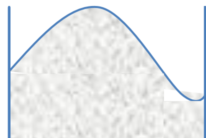
Kompüterdə təsadüfi ədədləri almağa çox çətindir. Bəzən, bunun üçün radio küylərin müxtəlif mənbələrindən istifadə edirlər. Lakin riyaziyyatçılar daha universal və rahat üsuldan - **psevdo təsadüfi ədədlərdən** istifadə edirlər.

Təsadüfi ədədlərə yaxın xassələrə malik olan ədədlər ardıcıllığı **psevdo təsadüfi ədədlər** adlanır. Burada növbəti ədəd müəyyən riyazi düstur əsasında əvvəlki ədədlər əsasında hesablanır.

Beləliklə, bu ardıcıllıq özünü təsadüfi ədədlər ardıcıllığı kimi aparır, baxmayaraq ki, düsturu bilərək, ardıcıllığın növbəti ədədini hesablamaq mümkündür.

Təsadüfi ədədlərin paylanması

Adətən, müəyyən intervalda dəyişən psevdo təsadüfi ədədlərdən istifadə edilir (daha sonra biz bu ədədləri təsadüfi ədədlər adlandıracayıq). Məsələn, fərz edin ki, qar bütün yerə yox, **OX** oxunun müəyyən **a, b** parçasına yağır. Özü də, bu ardıcıllığın müəyyən ümumi xassələrini bilmək vacibdir. Əgər küləksiz havada qar dənəciklərinə diqqət yetirsək, görmək olar ki, bütün yerlərdə qarın qalınlığı, demək olar ki, eynidir, küləkli havada isə - fərqlidir. Birinci halda deyirlər ki, təsadüfi ədədlərin **paylanması bərabərdir**, ikinci halda isə - **paylama qeyri-bərabərdir.**



Bərabər paylanma Qeyri-bərabər paylanma

Psevdo təsadüfi ədədlərin generatorlarının (düsturların) əksəriyyəti müəyyən interval üçün bərabər paylanma verirlər.

Belə ki, kompüterdə təsadüfi ədədlər düstur əsasında hasil olunurlar, onda hər hansı təsadüfi ardıcıllığı təkrar etmək üçün həmin başlanğıc qiyməti götürmək lazımdır.

Təsadüfi ədədlərlə işləyən funksiyalar

Təsadüfi ədədlərlə işləmək üçün C dilində bir neçə funksiya vardır. Onlar **stdlib.h** başlıq faylda təsvir olunublar. Bu o deməkdir ki, proqramın əvvəlində bu faylı qoşmaq lazımdır. Funksiyalar aşağıdakılardır:

n = rand ();	[0, **RAND_MAX)** intervalında tam təsadüfi ədədin alınması (RAND_MAX=32767).
srand (m);	təsadüfi ardıcıllığın başlanğıc qiymətini **m**-ə bərabər etmək.

Verilmiş intervalda təsadüfi ədədlərin alınması

Praktiki məsələlər üçün təsadüfi ədədləri verilmiş [a, b] intervalından almağa lazım gəlir. Əgər interval sıfırdan başlayırsa (a=0), onda qalığın tapılması əməliyyatından istifadə etmək olar: N-ə bölmə qalığı həmişə N-dən kiçikdir, yəni [0, N-1] intervalında yerləşir. Belə bir funksiya yazmaq olar:

```
int random ( int N )
{
  return rand() % N;     // [0, N-1] intervalında təsadüfi ədəd
}
```

Bu funksiya vasitəsi ilə [0, N-1] intervalında bərabər paylanmış təsadüfi ədədləri almaq mümkündür.

İndi isə, həmin funksiyanı [a,b] intervalı üçün istifadə edək. Aydındır ki, aşağıdakı düstur:

k = random (N) + a;

[a, a+N-1] intervalında yerləşən təsadüfi ədədləri hesablayır. Belə ki, bizə **[a,b]** intervalı lazımdır, onda **b=a+N-1** və **N=b-a+1**. Ona görə də,

> **[a,b]** intervalında bərabər paylanmış təsadüfi tam ədədlərin alınması üçün aşağıdakı düsturdan istifadə etmək lazımdır:
>
> k = random (b-a+1) + a;

Daha mürəkkəb məsələ - təsadüfi həqiqi ədədlərin alınmasıdır. Əgər **rand()** funksiyasını **RAND_MAX** ədədinə bölsək, onda nəticədə [0, 1) intervalında dəyişən təsadüfi həqiqi ədəd alacayıq:

x = (float) rand() / RAND_MAX;

Burada ədədlərin birini həqiqi tipə gətirmək lazımdır, çünki iki tam ədədin bir-birinə bölünməsi nəticəsində sıfır alınır.

[0, 1) intervalın uzunluğu 1-ə bərabərdir, bizdə isə intervalın uzunluğu **b-a** olmalıdır. Əgər ədədi **b-a**-ya vursaq və nəticəyə **a** əlavə etsək, onda lazım intervalı alacayıq.

> [a,b) intervalında bərabər paylanmış təsadüfi həqiqi ədədlərin əldə edilməsi üçün aşağıdakı düsturdan istifadə etmək lazımdır:
>
> **x = rand()*(b-a)/RAND_MAX + a;**

Bərabər paylanmış təsadüfi ədədlərdən qeyri-bərabər paylanmış ədədləri necə almaq olar? Riyazi statistikadan məlumdur ki, bir neçə bərabər paylanmış təsadüfi prosesin toplanması nəticəsində ədədlərin qeyri-bərabər (**normal və ya Qaus**) paylanması alınır.

Ekranda qarın yağması

Aşağıda göstərilmiş proqram **[0, 399]** intervalında **x** və **[0, 299]** intervalında **y** təsadüfi ədədləri generasiya edir və **(x,y)** koordinatlarına malik nöqtənin rəngini yoxlayır. Əgər nöqtənin rəngi qaradırsa, onda onun rəngi təsadüfi seçilir, əks halda qara olur. Rəngi təsadüfi seçmək üçün standart **COLOR** funksiyasından istifadə edilir. Funksiyanın parametrləri, yəni **R** (qırmızı), **G** (yaşıl), **B** (göy) [0,255] intervaldan təsadüfi seçilir.

```
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>
int random (int N) { return rand() % N; }     // funksiya
main ()
{
int x, y, R, G, B;
initwindow (500, 500);
while ( !kbhit () )   {                      //   hələ ki düymə sıxılmayıbsa
   x = random (400) ;                       //   təsadüfi koordinatlar
   y = random (300) ;
   R = random (256);                        //  təsadüfi rəng (R, G, B)
   G = random (256) ;
   B = random (256) ;
   if ( getpixel(x,y) != 0 )                // əgər nöqtə qara deyil
       putpixel (x, y, 0 );                 // onu qara edirik
   else                                     //  əks halda ...
       putpixel (x, y, COLOR(R, G, B) );   // təsadüfi rəng
    }
getch();
closegraph();
}
```

Qaydalar:

- Hər hansı düymənin basılmasını yoxlamaq üçün **kbhit** funksiyasından istifadə edilir. Əgər heç bir düymə sıxılmayıbsa, onda funksiya 0 qaytarır, əks halda funksiya sıfırdan fərqli qiymət qaytarır. Düymənin kodunu təyin etmək üçün **getch** funksiyasından istifadə edilir. Beləliklə, “hələ ki düymə sıxılmayıb” dövrü aşağıdakı kimi olmalıdır:

```
while ( ! kbhit() ) { … }
```

- cari nöqtənin rəngini təyin etmək üçün **getpixel** funksiyasından istifadə edilir.

12. MASSİVLƏR

Əsas anlayışlar

Massiv nədir?

Kompüterlərin əsas məqsədi **böyük həcmli verilənlərin emalıdır**. Çoxlu sayda verilənlərin yaddaşda yerləşdirilməsi zamanı müəyyən problemlərə rast gəlmək olur: yaddaşın hər bir xanasına ayrılıqda necə müraciət etmək olar? Hər xanaya ad verib sonra isə istifadə zamanı çaşmamaq – çətin işdir. Bu vəziyyətdən çıxmaq üçün adı xanaya yox, xanaların qrupuna verirlər. Həmin qrupda hər xana nömrələnib. Bu cürə təşkil olunmuş yaddaş sahəsi **massiv** adlanır.

> Eni tipli, yanaşı yerləşən və eyni ada malik olan yaddaş xanaların qrupu **massiv** adlanır. Qrupa daxil olan bütün xanalar unikal nömrəyə malikdir.

Massivlərlə işləyərkən üç məsələni həll etməyə bacarmaq lazımdır:

- massiv üçün lazımı ölçüdə yaddaşın ayrılması;
- verilənlərin lazımı xanaya yazılması;
- xanadan verilənlərin oxunması.

Massivin elanı

Massivi istifadə etmək üçün onu elan etmək, yəni ona yaddaşda yer ayırmaq lazımdır. Massivə daxil olan elementlərin tipi massivin tipini müəyyən edir. Massivlər müxtəlif tipli ola bilərlər: **int, float, char** və s. Massivi adi dəyişənlər kimi elan edirlər. Fərq ondan ibarətdir ki, massivin adından sonra kvadrat mötərizələrin içində onun ölçüsü yazılır.

```
int A[10], B[20];       // 10 və 20 tam ədədlərdən ibarət A və B massivləri
float C[12];            //  12 həqiqi ədəddən ibarət C massivi
```

Massivi elan edərkən fiqur mötərizələrin içində ədədləri sadalayaraq ona başlanğıc qiymətləri vermək olar.

```
int A[4] = { 2, 3, 12, 76 };
```

Əgər fiqur mötərizələrin içində massivin elementlərindən az ədəd yazılıbsa, onda qalan elementlərinin qiyməti sıfıra bərabər götürülür. Əgər siyahıda elementlərin sayından çox ədəd göstərilibsə, onda translyator səhv haqqında məlumat verəcəkdir. Məsələn,

```
int A[4] = { 2 };       //  massivin axırıncı üç elementi sıfıra bərabərdir
```

Proqramların universallığını artırmaq üçün massivin ölçüsünü konstanta (sabit ədəd) vasitəsi ilə təyin etmək məqsədə uyğundur. Bu halda proqramı başqa ölçülü massiv üçün yazmaqdan ötrü yalnız konstantanın qiymətini dəyişməyə kifayət edir.

```
cons int N = 20;      // konstanta
main ()
{
int A [N];           // massivin ölçüsü konstanta vasitəsi ilə verilib
…
}
```

Aşağıdakı cədvəldə düz və səhv elan olunmuş massivlər göstərilmişdir:

düz		səhv	
int A [20];	massivin ölçüsü birbaşa göstərilib	int A[ ];	massivin ölçüsü məlum deyil
const int N = 20; int A[N];	massivin ölçüsü – sabit ədəddir	int N = 20; int A [N];	massivin ölçüsü dəyişən ola bilməz

Massivin elementinə müraciət

Massivin hər elementinin sıra nömrəsi var. Massivin elementinə müraciət etmək üçün massivin adı yazılmalıdır, sonra isə kvadrat mötərizələr içində elementin sıra nömrəsi yazılmalıdır.

C dilində massivin elementləri **sıfırdan** başlayaraq nömrələnir. Ona görə də, əgər massivdə 10 element varsa, onda onlar aşağıdakılardır:

A[0], A[1], a[2], A[3], …. , A[9]

Elementin nömrəsi onun **indeksi** adlanır. **A** adlı massivə aşağıdakı kimi müraciət etmək olar:

```
x = ( A[3] + 5) * A[1];    // A[3] və A[1] elementləri əsasında x-i hesablamaq
A[0] = x + 6;              // yeni alınmış qiyməti A[0] yazmaq
```

C dilində massivin **sərhədlərinə nəzarət aparılmır**, ona görə də proqramda təyin olunmamış istənilən indeksli elementə müraciət etmək olar, məsələn, **A[345]** və ya **A[-12]**. Translyator səhv haqqında heç bir məlumat verməyəcəkdir, lakin proqram “asıla” və ya düzgün olmayan cavab verə bilər.

Massivlərin daxil edilməsi və xaric edilməsi

Verilənləri massivə necə daxil etmək olar? Bunun üçün çoxlu sayda üsul mövcuddur:

- massivin elementləri klaviaturadan daxil olunur;
- massiv təsadüfi elementlərlə doldurulur (təsadüfi proseslərin modelləşdirilməsi üçün);

- massivin elementləri fayldan oxunur;
- massivin elementləri hər hansı xarici qurğunun portundan daxil olunur (məsələn, skaner, modem və s.);
- massivin elementləri proqramda hesablanır.

Misal. Klaviaturadan massivin 10 elementini daxil edib, onları 2-ə vurmaq və alınmış massivi ekrana çıxartmaq.

Təəssüf ki, kompütere "**A** massivi daxil et" əmrini verə bilmərik. Massivin hər elementi ayrıca oxunmalıdır.

Massivi yaddaşa daxil etmək üçün, onun hər elementi **scanf** funksiyası vasitəsilə oxunmalıdır.

Sadə proqramlarda massivi klaviaturadan daxil edirlər. Burada elementlərin sayı az olur. Massivin daxil edilməsi uçun **for** dövr operatorundan istifadə edəcəyik. Daxil etməzdən əvvəl massivi elan etmək, yəni ona yaddaşda yer ayırmaq lazımdır.

Massivə yaddaşda nə qədər yer ayrılıbsa, o qədər də element daxil edilə bilər. Elementlərin indeksləri sıfırdan başlayır, ona görə də, əgər 10 elementli massiv elan olunubsa, onda onun axırıncı elementinin indeksi 9-a bərabərdir. Massivi daxil etmək üçün ekrana ümumi (bütün massiv üçün) və ayrı-ayrı elementlər üçün göstəriş çıxartmaq lazımdır.

Massivin elementlərini 2-ə vurmaq üçün yenidən dövrdən istifadə etmək lazımdır. Dövrün hər siklində 1 element emal olunur.

Massivi ekrana çıxartmaq üçün yenidən **for** dövründən istifadə olunmalıdır. Elementlər bir-bir ekrana çıxarılır. Əgər printf operatorun formatında aralıq (" ") qoyulsa, elementlər bir sətirdə, "\n" simvolu qoyulsa, onda massivin elementləri sütun şəkilində ekrana çıxarılacaqdır.

```
#include <stdio.h>
#include <conio.h>
const int N  = 10;            // massivin ölçüsü
main ()
{
int I, A[N];                          // massivin elanı
printf ( "A massivi daxil edin:\n" );     // daxiletmə göstərişi
for ( i = 0; i < N; i ++)              // bütün elementlər üzrə dövr
    {
    printf ( "A[%d] elementi daxil edin >", i );    //  A[i] elementi daxil etmək üçün göstəriş
    scanf ( "%d", &A[i] );                              // A[i] elementinin daxil edilməsi
    }
for ( i = 0; i < N; i ++)              // bütün elementlər üzrə dövr
    A[i] = A[i] * 2;                  // A[i] vurulur 2-ə
printf ( "\nNetice:\n");
for ( i = 0; i < N; i ++)              // bütün elementlər üzrə dövr
   printf ( "%d  ", A[i] ) ;            // A[i] elementi çap olunur
```

```
getch();
}
```

Massivin təsadüfi ədədlərlə doldurulması

Bu üsuldan təsadüfi proseslərin, məsələn, *Broun hərəkətinin* modelləşdirilməsi üçün istifadə edilir. Tutaq ki, massivi **[a, b]** parçasında bərabər paylanmış təsadüfi ədədlərlə doldurmaq lazımdır. Belə ki, tam və həqiqi ədədlərin generasiyası fərqlidir, hər iki variantı nəzərdən keçirək. Proqramın əvvəlində aşağıdakı sətir yazılmalıdır:

```
cons int N  = 10;
```

Təsadüfi ədədləri generasiya edən **rand** funksiyası **stdlib.h** başlıq faylında yerləşir, ona görə də bu faylı proqrama qoşmaq lazımdır. [0, N-1] parçasında bərabər paylanmış ədədlərin generasiyası üçün **random** funksiyasından istifadə edək:

```
int random (int N) { return rand() % N; }
```

[a, b] parçasında bərabər paylanmış təsadüfi tam ədədlərin alınması üçün **random** funksiyasına aşağıdakı kimi müraciət olmalıdır:

```
k  = random (b – a +1 ) + a;
```

Təsadüfi həqiqi ədədlərin generasiyası üçün başqa üsuldan istifadə olunur:

```
x = rand() * (b – a) / RAND_MAX +a;
```

Burada **RAND_MAX** konstantası **rand** funksiyası tərəfindən hasil olunana maksimal təsadüfi ədəddir.

Aşağıda göstərilmiş proqramda **A** massivi [-5, 10] intervalından təsadüfi *tam* ədədlərlə, **X** massivi isə - təsadüfi *həqiqi* ədədlərlə doldurulur.

```
#include  <stblib.h>
const int N  = 10;
main ()
{
int i, A[N], a = -5, b = 10;
float X[N];
for ( i  = 0; i < N; i ++)
     A[ i ]  = random (b – a + 1) +a;

for ( i  = 0; i < N; i ++)
    X [ i ] = (float) rand() * (b – a) % RAND_MAX + a;
...........
}
```

rand funksiyasının qabağında **(float)** sözü yazılmalıdır, çünki:

- **rand** funksiyasının qiymətini **(b-a)**-a vuranda, ola bilsin alınmış qiymət elə böyük olsun ki, **int** tipli dəyişənə yerləşməsin;
- **C** dilində iki tam ədədlərin bölünməsi nəticəsində qalıq nəzərə alınmır, yəni nəticə düz olmayacaqdır.

Misal. Massivi [-10, 15] intervalında dəyişən təsadüfi tam ədədlərlə tamamlamaq. Bütün elementləri 2-ə vurub, əvvəlki və alınmış massivi ekrana çıxartmaq.

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int random (int N) { return rand() % N; }

const int N = 10;

main()
{
int i, A[N];
for (i = 0; i < N; i ++ )     //  massiv təsadüfi ədədlərlə doldurulur
   A[i] = random(26)  - 10;

printf ( " Evvelki massiv: \n");         // əvvəlki massivin xaric edilməsi
for (i = 0; i < N; i ++ )
  printf ( " %d  ", A[i] );
for (i = 0; i < N; i ++ )
   A[i] =A[i] * 2 ;                      //   bütün elementlər 2-ə vurulur
printf ( "\n Netice: \n" );
for (i = 0; i < N; i ++ )               //   hesablanmış massivin xaric edilməsi
     printf ( " %d ", A[i] );
getch();
}
```

Mətn fayllarla işləmə

Əgər klaviaturadan çoxlu sayda verilənlər daxil olunursa, onda onları hər dəfə təkrar daxil etməsi yorucu bir iş olar. Bunun üçün diskdə bir fayl yaradıb lazımı verilənləri ora yazırlar, və proqram ona lazım olan məlumatları məhz bu fayldan oxuyur.

Fayllar **mətn** (bu tipli fayllara hərifləri, rəqəmləri, mötərizələri və digər simvolları) və **binar** (bu tipli fayllarda simvollar cədvəlinin istənilən simvolları saxlanıla bilər) tipli olurlar. Biz yalnız mətn fayllara baxacayıq.

Proqramdan fayllara necə müraciət etmək olar

Fayllarla iş sendviç prinsipinə əsaslanır:

"Faylı açmaq" onunla işə başlamaq, onu aktiv etmək və digər proqramların bu fayla müraciətini blokadaya salmaq deməkdir. Faylı bağlayanda o blokadadan çıxarılır, digər proqramlar artıq onunla iş görə bilirlər və sizin bütün dəyişiklikləriniz diskdə yadda saxlanılır.

Faylla işləmək üçün **faylın göstəricisi** adlanan xüsusi dəyişəndən istifadə edilir. Bu açıq fayl haqqında bütün məlumatları özündə saxlayan yaddaş blokunun ünvanıdır. Fayl göstəricisini aşağıdakı kimi elan etmək lazımdır:

```
FILE *fp ;
```

Faylı açmaq üçün **fopen** funksiyasına müraciət edilir. Bu funksiya faylı açıb onun ünvanını **fp** dəyişənə yazır. Bundan sonra fayla müraciət etmək üçün onun adına yox, **fp** göstəricisinə müraciət etmək lazımdır.

```
fp = fopen ( "qq.dat", "r" );
```

Burada cari qovluqda yerləşən **qq.dat** faylı oxumaq üçün (fopen funksiyasının ikinci parametri "r" **read** rejimi deməkdir) açılır. Əgər lazım gələrsə, onda faylın marşrutu tam göstərilməlidir. Məsələn,

```
fp = fopen ( "c:\\data\\qq.dat", "r" );
```

Marşrutu göstərərkən "/" (sləş) simvolu həmişə cüt yazılmalıdır, çünki tək sləş simvolu – müəyyən kombinasiyalarda istifadə olunan xüsusi simvoldur, məsələn, **\n**.

"**r**" (fayldan oxuma) rejimindən başqa digər rejimlərdən də istifadə oluna bilər:

"r"	Faylı oxumaq üçün açmaq.
"w"	Yeni fayla yazma. Əgər diskdə həmin adda fayl mövcuddursa, onda əvvəlcə o silinəcəkdir.
"a"	Faylın sonuna əlavə. Əgər diskdə bu adda fayl varsa, onda yeni məlumatlar faylın sonuna əlavə olunurlar. Əgər fayl mövcud deyilsə, onda o, yaranacaqdır.
"r+"	Yazıb/oxumaq üçün mövcud faylı açmaq.
"w+"	Yazıb/oxumaq üçün yeni fayl yaratmaq (əgər həmin adda fayl varsa, onda o, yenisi ilə əvəz olunur).

Bəzən proqram faylı aça bilmir. Əgər fayl oxumaq üçün açılırsa, onda bu vəziyyət aşağıdakı hallarda ola bilər:

- faylın adı düz verilməyib və ya bu fayl diskdə yoxdur;
- Fayl digər proqram tərəfindən istifadə olunur və blokadaya salınıb;

Əgər fayl yazmaq üçün açılırsa, onda bu əməliyyat aşağıdakı hallarda uğursuzluqla nəticələnə bilər:

- diskdə yer yoxdur;
- fayl yazmaq üçün müdafiə olunur (**protected**);
- faylın adı düz yazılmayıb (məsələn, adında iki nöqtə, sual işarəsi və s. simvollar var).

Əgər faylı açmaq mümkün deyilsə, onda **fopen** funksiyası **NULL** kimi ifadə olunan xüsusi sıfır qiyməti (boş göstərici) qaytarır. Ona görə də, faylın düzgün açılmasına nəzarət etmək lazımdır. Əgər fayl açılmırsa, onda səhv haqqında məlumat verib, proqramdan çıxmaq lazımdır.

```
if ( fp == NULL )
  {
   printf ( “Fayl movcud deyil “);
   return 1;                    //  proqramdan çıxış. Səhvin kodu 1-dir
  }
```

Əgər fayl açılıbsa, onda bu fayldan verilənləri oxumaq olar. Bundan ötrü **fscanf** funksiyasından istifadə olunur. Bu funksiya **scanf** funksiyasına oxşayır. Fərq ondan ibarətdir ki, bu funksiya verilənləri klaviaturadan yox, fayldan oxuyur və onun birinci parametri fayl göstəricisidir.

```
n = fscanf ( fp,  “%d”,  &A[ i ] ) ;
```

Nəticə olaraq, **fscanf** funksiyası oxuduğu ədədlərin sayını qaytarır. Əgər bir ədəd oxunursa, onda **n** dəyişənin qiyməti 1 ola bilər. Əgər oxuma prosesi səhvlə nəticələnirsə (verilənlər qurtarır və ya səhvdir, yəni fayla ədədlərin əvəzinə sözlər yazılıb), onda n-in qiyməti **0** olur. Fayla verilənləri düzgün yazmaq üçün onlar boşluqlarla və ya yeni sətrə keçmə simvolu ilə (**Enter** simvolu) bir-birindən ayırmalıdırlar.

Əgər fayl yazmaq üçün açılıbsa, onda **fprintf** funksiyası vasitəsi ilə bu fayla verilənləri yazmaq olar. Bu funksiya **printf** funksiyasına oxşardır.

Faylla işi bitirdikdən sonra onu **fclose** funksiyası vasitəsi ilə bağlamaq lazımdır.

```
fclose ( fp ) ;
```

Bundan sonra **fp** göstəricini digər fayl üçün istifadə etmək olar.

Müəyyən ölçülü massivlər

Misal. **input.dat** faylından 10 tam tipli ədədlərdən ibarət massivi daxil edib, hər elementi 2-a vurub, alınmış massivi **output.dat** faylına yazmaq.

Bu məsələni həll etmək üçün **fopen, fscanf, fprintf** və **fclose** funksiyalardan istifadə etmək lazımdır. Proqramda iki vəziyyəti (səhvi) nəzərə almaq lazımdır:

- fayl yoxdur (açılmır);
- faylda verilənlərin sayı azdır və ya verilənlər səhv yazılıb.

```
#include <stdio.h>

const int N = 10;

main ()
{
int i, A[N];
FILE *fp;              //  fayl göstəricisi
fp = fopen ( "input.dat", "r" );     // oxumaq üçün faylın açılması
if ( fp == NULL )                      // səhvin emalı
    { printf ( "Fayl mövcud deyil!" );
     return 1;                       // proqramın dayandırılması. Səhvin kodu 1-dir
     }
for ( i = 0; i < N; i ++ )
  if ( fscanf (fp, "%d", &A[i] ) == 0 )   // verilənlər oxumadığı halda
      { printf ( "Faylda verilenlerin sayi azdir" );  // səhv haqqında məlumat
        break;
      }
fclose (fp );                       // faylın bağlanması
for ( i = 0; i < N; i ++ )
   A[i] = A[i] * 2;
fp = fopen ( "output.dat", "w" );                 //  verilənləri yazmaq üçün faylın açılması
for ( i = 0; i < N; i ++ )
     fprintf (fp, "%d\n", A[i] );            //  sütun şəkilində massivin fayla yazılması
fclose ( fp );
}
```

Yuxarıdakı proqramlardan fərqli olaraq bu proqramın nəticələri ekrana çıxmır, onlar **output.dat** faylına yazılır.

Qeyri müəyyən ölçülü massivlər

Misal. **input.dat** faylında iki sütun şəkilində **(x,y)** cütlükləri yazılıb. Hər cütlük üçün **(x+y)** cəmləri hesablayıb nəticəni **output.dat** faylına yazmaq.

Məsələnin mürəkkəbliyi ondan ibarətdir ki, **input.dat** faylına yazılan cütlüklərin sayı qabaqcadan bilinmir. Hər iki **x** və **y** massivinə yaddaşda yer ayırmaq olmayacaqdır, çünki fayla yazılan ədədlərin sayı bilinmir.

Lakin, bu məsələni başqa cürə həll etmək olar. Hər cütlüyün cəmini hesablamaq üçün yalnız iki ədəd lazımdır, qalan ədədləri yaddaşda saxlamasaq da olar. Cəmi

hesablayandan sonra, onu yaddaşda saxlamaq lazım deyil, onu birbaşa çıxış faylına yazmaq lazımdır. Məsələnin həlli üçün aşağıdakı **alqoritmdən** istifadə olunacaqdır:

1. iki faylı açmaq (biri – verilənləri oxumaq, digəri isə verilənləri yazmaq üçün istifadə olunur);
2. iki ədədi **x** və **y** dəyişənlərə oxuyuruq. Əgər oxumaq olmursa (verilənlər yoxdur və ya düz deyil), onda işi bitirmək;
3. **x** və **y** toplayıb, nəticəni çıxış faylına yazmaq;
4. 2-ci addıma keçmək.

Oxuma əməliyyatının müvəffəqiyyətlə başa çatmasını yoxlamaq üçün **fscanf** funksiyasının qaytardığı qiyməti qeyd edəcəyik (bu funksiya oxunan ədədlərin sayını nəticə kimi qaytarır). Hər dəfə iki ədəd, yəni **x** və **y** oxuyacayıq. Əgər hər şey normaldırsa, onda **fscanf** funksiyasının qiyməti 2-ə bərabər olacaqdır. Əgər funksiyanın qiyməti 2-dən azdırsa, onda verilənlər qurtarıb və ya səhvdir.

Qeyd edək ki, eyni zamanda iki açıq fayl ilə işləmək lazımdır, ona görə də iki fayl göstəricisindən istifadə olunmalıdır. Onları **fin** və **fout** kimi işarə edək. Proqramı qısaltmaq məqsədilə faylın açılması zamanı rast gələ biləcəyimiz səhvlər emal olunmur.

```
#include <stdio.h>
main ()
{
int n, x,y, sum;
FILE *fin, *fout;              //
fin = fopen ( “input.dat”, “r” );   //
fout = fopen (“output.dat”, “w” );   //
while ( 1 ) {
     n = fscanf ( fin, “%d%d”, &x, &y );
     if ( n < 2 ) break;              //
     sum = x + y;
     fprintf ( fout, “%d\n”, sum );
    }
fclose ( fout );
fclose (fin );
}
```

Proqramda sonsuz **while** dövrdən istifadə olunur. Əgər faylda verilənlər qurtarırsa, onda dövr öz işini bitirir.

Binar fayllarla işləmə

Mətn fayllardan fərqli olaraq binar fayllarda məlumatlar maşın dilində yazılır. Binar fayla baxarkən heç nə başa düşmək olmur. Binar faylların üstünlüyü ondan ibarətdir ki, binar faylından massivi bütöv blok kimi oxumaq olar. Bir əmr vasitəsi ilə bütöv massivi və ya onun bir hissəsini yazmaq olar.

Binar faylların açılması zamanı **“r”, “w”** və **“a”** rejimlərin əvəzinə, uyğun olaraq, **“rb”, “wb”** və **“ab”** rejimlərdən istifadə olunur. Əlavə **“b”** hərfi onu göstərir ki, fayl binardır (ing. *binary*). Baxdığımız məsələni binar fayl üçün həll edək:

Misal. **input.dat** binar faylından 10 tam ədədi oxuyub, hər elementi 2-ə vurub, **output.dat** binar faylına yazmaq.

```
#include <stdio.h>
const int N = 10;
main ()
{
int i, n, A[N];
FILE *fp ;           // fayl göstəricisi
fp = fopen ( “input.dat”, “rb” );  //  oxumaq üçün binar  faylın açılması

n = fread ( A, sizeof(int), N, fp );     // bütöv massivin oxunması
if ( n < N ) {
        printf ( “Faylda ededlerin sayi azdir!” );
        break;
        }
fclose ( fp );               //  faylın bağlanması
for ( i = 0; i <N; i ++ )
      A[i] = A[i] * 2 ;
fp = fopen ( “output.dat”, “wb” );    //  yazmaq üçün binar faylın açılması
fwrite ( A, sizeof(int), N, fp );      // bütöv massivin yazılması
fclose ( fp );                          // faylın bağlanması
}
```

Binar fayldan verilənləri oxumaq üçün **fread** funksiyasından istifadə olunur. Bu funksiyanın 4 parametri var:

- **yaddaş ünvanı**, yeni oxunmuş verilənləri (bizim misalda **A** massivinin birinci elementinin ünvanı. **&A[0]** və ya sadəcə, A kimi işarə edilir) hara yazmaq;
- **bir elementin ölçüsü**. Yaxşı olardır ki, kompüter ölçünü özü təyin etsin. Bizim misalda **sizeof(int)** – tam ədədin ölçüsüdür. Baxmayaraq ki, DevC++-da tam ədəd 4 bayt yer tutur, digər proqramlaşdırma paketlərində bu rəqəm fərqli ola bilər. Ona görə də standart **sizeof** funksiyasından istifadə etmək daha məqsədə uyğundur, çünki onda proqram **daşına bilən** (ing. **compatible**) olur;
- **elementlərin sayı (N)**;
- **fayl göstəricisi (fp)**.

Nəticə olaraq, **fread** funksiyası oxunmuş massiv elementlərinin ümumi sayını qaytarır. Bu qiyməti səhvlərin emalı üçün istifadə etmək olar. Əgər funksiyanın qaytardığı qiymət elementlərin sayından kiçikdirsə, onda faylda verilənlər çatışmır.

Massivi binar fayla yazmaq üçün **fwrite** funksiyasından istifadə olunur. Bu funksiyanın parametrləri **fread** funksiyasının parametrləri ilə üst-üstə düşür. Nəticə olaraq, bu funksiya fayla yazılmış elementlərin sayını qaytarır.

Bu üsulun əsas üstünlüyü ondan ibarətdir ki, massiv bir vahid blok şəkilində həm oxunur, həm də yazılır. Bu isə proqramın sürətini artırır.

Massivdə sadə axtarış

Məsələlərin əksəriyyətində verilmiş massivin bütün elementlərini bir-bir yoxlayaraq tələb olunan elementləri tapmaq lazımdır. Bu məsələlər aşağıdakılardır:

- verilmiş elementin axtarışı;
- verilmiş şərtə uyğun elementlərin axtarışı;
- seçilmiş elementlərdən yeni massivin yaradılması;
- minimal (maksimal) elementin seçilməsi.

Bütün bu məsələlər dövr operatoru vasitəsi ilə həll olunur. Dövrdə **0**-dan **N-1**-ci elementinə kimi seçim aparılır. Bu cürə axtarış **xətti axtarış** adlanır, çünki bütün elementlər ardıcıl olaraq yoxlanılır.

Verilmiş elementin axtarışı

Misal. Verilmiş massivdə qiyməti x-a bərabər olan elementi tapmaq. Əgər belə element varsa, onun nömrəsini təyin etmək.

Əgər massivin elementləri haqqında heç bir əlavə informasiya yoxdursa, onda xətti axtarışdan istifadə olunmalıdır, yəni elementlər bir-bir yoxlanılmalıdır və əgər hər hansı element verilmiş qiymətlə üst-üstə düşürsə, onda axtarış dayandırılır və nəticə ekranda əks olunur. Bu alqoritmi aşağıdakı proqram reallaşdırır:

```
#include <stdio.h>
const int N =10;
main ()
{
int i, X, A[N];
int success = 0;              //  bayraq dəyişəni.
// burada A massivi və X daxil olunmalıdır
for (i = 0 ; i <N; i ++)
    if ( A[i] == X )          //  əgər element tapılıbsa, onda
       { success = 1;         // bayrağı 1-ə qoymaq
       break;                 //  dövrdən çıxmaq
       }
if ( success )
    printf ( "A[%d] = %d", i, A[i] );
else
    printf ( "Element %d  tapilmadi.", X );
}
```

Əgər X-a bərabər element tapılmırsa, onda bu vəziyyəti qeyd etmək üçün xüsusi **success** dəyişəndən istifadə etmək lazımdır. Əgər element tapılıbsa, onda onun qiyməti 1 olur, əks halda 0 qalır. Bu cürə dəyişənlər **bayraq** adlanır. Bayrağı qaldırmaq olar (=1) və aşağı endirmək olar (=0).

Xətti axtarış zamanı ən pis halda N müqayisə əməliyyatı olacaqdır. Aydındır ki, axtarışı sürətləndirmək üçün əvvəlcə hər hansı üsulla verilənləri nizamlamaq lazımdır. Bu halda axtarış daha səmərəli olur.

Müəyyən şərtə uyğun elementlərin axtarışı

Misal. Massivdə müsbət elementlərin sayını təyin edib, nəticəni ekrana çıxartmaq.

Məsələnin həlli üçün **sayğac** adlanan xüsusi dəyişəndən istifadə edəcəyik. Növbəti müsbət element tapıldığı anda sayğacın qiyməti bir vahid artırılacaqdır.

```
#include <stdio.h>
const int N=10;
main ()
{
int i, A[N], count = 0;       // count – müsbət elementlərin sayğacıdır
//  Burada massiv daxil olunmalıdır
for (i = 0; i < N; i++ )        //  bütün elementlərə görə dövr
    if ( A[i] > 0 ) {            //  əgər element müsbətdirsə,
    count ++;                   //   sayğac 1 artırılır
    printf ( "%d ", A[i] );     //  elementi çapa veririk
    }
printf ( "\nMassivde %d musbet element var", count );
}
```

Verilmiş şərt əsasında massivin yaradılması

Misal. **A** massivinin bütün müsbət elementləri əsasında yeni **B** massivi yaratmaq. **B** massivi ekrana çıxartmaq.

Tutaq ki, **A[N]** massivi verilib. Bu massivdən bütün müsbət elementləri seçərək yeni massivə yazmaq lazımdır. Sonra alınmış massivlə işləmək lazımdır.

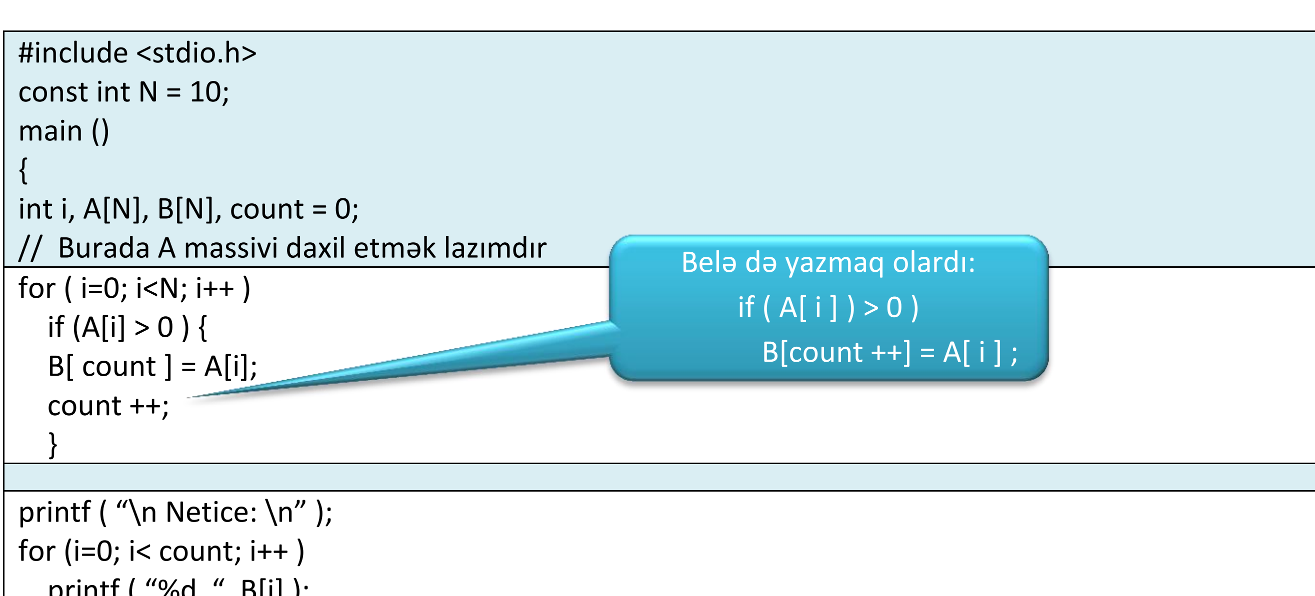
Əvvəlcə **B** massivinin yaddaşda tutduğu yer haqqında düşünmək lazımdır. **B** massivinə nə qədər yer ayırmalıdır? Ən pis halda **A** massivin bütün elementləri müsbət ola bilər, onda onlar hamısı **B** massivinə yazılacaqdır, ona görə də **B** massivin ölçüsü **A** massivin ölçüsü kimi olmalıdır.

Belə üsuldan da istifadə etmək olar: bütün **A** massivi yoxlanılır və, əgər növbəti element üçün **A[i]>0**, onda onun qiyməti **B[i]** massivinə köçürülür.

Lakin bu cürə **B** massivindən istifadə etmək çətindir, çünki elementlər ardıcıl yerləşmir.

Daha mükəmməl üsuldan istifadə etmək lazımdır. **count** - sayğac tipli dəyişəni elan edirik. Bu dəyişən müsbət elementlərin sayını özündə saxlayacaqdır. Əvvəlcə onun qiyməti sıfıra bərabərdir. Əgər növbəti müsbət element tapılırsa, onda onu B[count] elementinə yerləşdiririk və sayğacı artırırıq. Beləliklə, bütün elementlər **B** massivinin əvvəlində ardıcıl yerləşir.

A	1	-1	3	-2	5	→	B	1	3	5	?	?



```
#include <stdio.h>
const int N = 10;
main ()
{
int i, A[N], B[N], count = 0;
//  Burada A massivi daxil etmək lazımdır
for ( i=0; i<N; i++ )
   if (A[i] > 0 ) {
   B[ count ] = A[i];
   count ++;
   }

printf ( "\n Netice: \n" );
for (i=0; i< count; i++ )
   printf ( "%d  ", B[i] );
}
```


Axırıncı dövrdə sayğac **0**-dan **count-1**-ə kimi dəyişir, ona görə də ekrana **B** massivinin real istifadə olunan elementləri çıxarılır.

Minimal element

Misal. **A** massivinin minimal elementini təyin edib ekrana çıxartmaq.

Məsələnin həlli üçün əlavə dəyişən götürülməlidir. Bu dəyişən təyin olunmuş minimal elementi özündə saxlayacaqdır. Əvvəlcə bu dəyişənə A massivinin birinci elementi yazılır (**A[0]**). Sonra növbəti element götürülür və minimal elementlə müqayisə olunur. Əgər müqayisə olunan element minimal elementdən kiçikdirsə, onda bu element minimal elementin yerinə yazılır, və bu müqayisə bütün elementlər üçün davam edir. Axırıncı müqayisədən sonra əlavə dəyişənin içində massivin minimal elementi olacaqdır.

Qeyd edək ki, müqayisə 0-ci yox, 1 indeksli elementdən başlayır, çünki başlanğıc elementə ayrıca baxılıb.

```
#include <stdio.h>
const int N = 10;
main ()
```

```
{
int i, A[N], min;
// Burada A massivi daxil edilməlidir
min = A[0];              // tutaq ki, A[0]  - minimal elementdir
for ( i=1; i<N; i++ )    // bütün elementlərə görə dövr
if ( A[i] < min )        // əgər A[i] minimal elementdən kiçikdirsə, ...
    min = A[i];          //  onda A[i] olur minimal element
printf ( "\nMinimal element %d", min );
}
```

Maksimal elementi təyin etmək üçün proqramda yalnız yoxlama şərti əksi ilə əvəz olunmalıdır (A[i]>min). Əlavə dəyişənin adını **max** etməyə daha məqsədə uyğundur.

Məsələni bir az da çətinləşdirək.

Misal. **A** massivinin minimal elementini və onun nömrəsini təyin edib ekrana çıxartmaq.

Məsələnin həlli üçün minimal elementin nömrəsini yadda saxlamaq üçün əlavə dəyişən götürülməlidir. Minimal element təyin olunandan sonra onun qiymətini bir dəyişənə, nömrəsini isə digər dəyişənə yazmaq lazımdır.

Bu məsələni bir əlavə dəyişənlə də həll etmək olar. Əlavə dəyişəndə yalnız minimal elementin nömrəsini saxlasaq kifayət edər, çünki elementin nömrəsi bilinirsə, onun qiyməyini asanlıqla təyin etmək mümkündür. İndi proqramda **nMin** adlı dəyişəndə elementin qiymətini yox, nömrəsini saxlayacayıq.

```
#include <stdio.h>
const int N = 10;
main ()
{
int i, A[N], nMin;          // nMin – minimal elementin nömrəsi
// Burada A massivi daxil etmək lazımdır
nMin = 0;                   // tutaq ki, A[0] - minimaldır
for ( i=1; i<N; i++ )       //  bütün elementlərə görə dövr
if ( A[i] < A[nMin] )       // əgər A[i] < A[nMin], onda ...
   nMin = i;                //    indeksi yadda saxlayırıq
printf ("\n Minimal element A[%d]=%d", nMin, A[nMin] );
}
```

Massiv elementlərinin yerdəyişməsi

Yerdəyişmə

Tutaq ki, stolun üzərində iki qab var. Birinin içində süd, digərində isə qatıq var. Əgər bu qabların içini dəyişmək lazım gələrsə, onda bizə üçüncü qab lazım olacaqdır. Əvvəlcə südü üçüncü qaba, qatığı birinci qaba tökürük və nəhayət, üçüncü qabdan südü ikinci qaba qaytarırıq.

Proqramlaşdırmada da belədir. Yaddaşın iki xanasının tərkibinin yerdəyişməsi üçün əlavə üçüncü dəyişəndən istifadə etmək lazımdır. Tutaq ki, **a** və **b** xanalara müəyyən qiymətlər yazılıb. Aşağıdakı əməliyyatlar vasitəsi ilə onların qiymətləri dəyişdiriləcəkdir.

```
a = 4; b = 6;
c = a;      //  a=4; b=6; c=4;
a = b;      //  a=6; b=6; c=4;
b = c;      //   a=6; b=4; c=4;
```

Inversiya

İnversiya (ing. inverse – tərs) yerdəyişmənin bir növüdür. Burada birinci element axırıncı, ikinci element axırıncıdan qabaqki element olur və s.

1	2	3	4	5

→

5	4	3	2	1

Bu məsələnin həllində bəzi məqamlar var. Tutaq ki, massivin ölçüsü **N**-dir. Onda **A[0]** elementi **A[N-1]**, **A[1]** elementi **A[N-2]** və s. elementləri əvəz etməlidir. Qeyd edək ki, yeri dəyişilən elementlərin indekslərin cəmi **N-1**-ə bərabərdir. Ona görə də, **A[i]** və **A[N-1-i]** elementlərin yerdəyişməsi üçün dövr 0-dan N-1-ə kimi olmalıdır. Lakin, proqram işləyib qurtarandan sonra məlum olur ki, massiv dəyişməyib. Bu onunla əlaqədardır ki, əvvəlcə massivin birinci yarısında yerləşən elementlər ikinci yarısının elementləri ilə yerlərini dəyişirlər, lakin sonra indeks ikinci yarıya keçəndən sonra (i > N/2) həmin elementlər öz yerlərinə qaytarılır. Məsələnin düzgün həlli ondan ibarətdir ki, dövrün indeksi massivin ortasına qədər dəyişməlidir.

```
#include <stdio.h>
const int N = 10;
main ()
{
int i, A[N], c;
//  Burada A massivi daxil olmalıdır
for (i=0; i<N/2; i ++)       // dövrün indeksi ortaya qədər dəyişir
   {
    c = A[i];                // yerdəyişmə
    A[i] = A[N-1-i];
    A[N-1-i]  = c;
    }
printf ( "\n Netice: \n" );
for ( i=0; i<N; i++ )
   printf ( "%d  ", A[i] );
}
```

Dövrlü sürüşdürmə

Dövrlü sağa sürüşdürmə zamanı birinci element ikincinin yerinə, ikinci element üçüncünün yerinə və s., axırıncı element birincinin yerinə sürüşdürülür.

1	2	3	4	5

→

5	1	2	3	4

Dövrlü sürüşdürmə üçün əlavə dəyişən lazımdır. Bu dəyişən yerdəyişmə zamanı axırıncı elementin qiymətini özündə saxlayacaqdır. Qeyd edək ki, dövr massivin axırıncı elementindən başlamalıdır, əks halda massivin bütün elementləri birinci elementin qiymətini alacaqlar. Birinci elementin qiyməti ayrıca hesablanır. O, əlavə dəyişəndən götürülür.

```
#include <stdio.h>
const int N = 10;
main ()
{
int i, A[N], c;
//  Burada A massivi daxil edilməlidir
c = A[N-1];     //  axırıncı elementi yadda saxla
for ( i=N-1; i>0; i -- )    //  indeksi azaldan dövr
   A[i] = A[i-1];
A[0] = c;                 //  birinci elementin qiyməti
printf ( "\n Netice: \n" );
for ( i=0; i<N; i++ )
    printf ( "%d  ", A[i] );
}
```

Massivlərin çeşidlənməsi

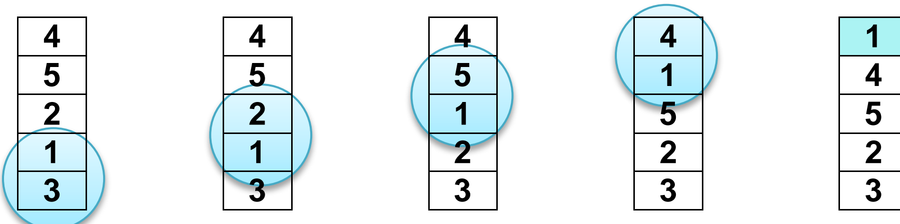
Siyahıda verilmiş elementlərin verilmiş ardıcıllıqda yerləşdirilməsi **çeşidləmə** (sorting) adlanır.

Çeşidləmənin bir neçə növü var (əlifbaya görə, tarixlərə görə və s.). Bütün bunlar bir-birindən elementlərin müqayisə prosedurаları ilə fərqlənir. Ən sadə çeşidləmə üsulunu - massiv elementlərinin artmaya görə çeşidlənməsini nəzərdən keçirək.

Hava qabarcığı üsulu

Bu üsulun adı məlum fiziki hadisədən götürülmüşdür: suda hava qabarcıqları yuxarı qalxır. Bu üsulda əvvəlcə massivin ən kiçik ("yüngül") elementi "yuxarı" (massivin əvvəlinə) qalxır, sonra növbəti element və s.

Əvvəlcə axırıncı elementi axırıncıdan əvvəlki elementlə müqayisə edirik. Əgər onlar düzgün dayanmayıblarsa, onda onların yerlərini dəyişirik. Sonra növbəti cütlüyü götürüb həmin ardıcıllığı təkrar edirik. Axırıncı cütlüyü, yəni A[0] və A[1] elementlərini emal etdikdən sonra minimal element A[0]-ın yerində dayanacaqdır. Növbəti addımlarda bu elementi nəzərə almırıq.



Növbəti addımda **A[1]** elementi öz yerinə qoyulmalıdır. Həmin alqoritmdən istifadə edirik, lakin **A[0]** elementinə artıq baxmırıq (o, öz yerində dayanıb).

```
#include <stdio.h>
const int N = 10;
main ()
{
int i, j, A[N], c;
// Burada A massivi daxil edilməlidir
for ( i=0; i<N-1; i++ )           // müqayisələrin sayı N-1 olmalıdır
   for ( j=N-2; j >=i; j -- )       // massivin axırından əvvələ gedirik
      if ( A[j] > A[j+1] )          // əgər şərt ödənmirsə ...
         {
          c = A[j]; A[j] = A[j+1]; A[j+1] = c;   // A[j] və A[j+1] elementlər öz yerlərini dəyişir
         }
printf ( "\n Cheshidlenmish massiv: \n" );
for ( i=0; i<N; i++ )
    printf ( "%d  ", A[i] );
}
```

Hava qabarcığı üsulu böyük massivlər üçün çox ləng işləyir. Massivin ölçüləri 10 dəfə artdıqda proqramın yerinə yetirilmə müddəti 100 dəfə artır (N^2 dəfə). Təəssüf ki, bu ən sadə çeşidləmə üsullarına aiddir.

Minimal elementin seçilməsi üsulu

Hava qabarcığı üsulun çatışmayan cəhətlərdən biri yanaşı elementlərin tez-tez yerdəyişməsidir. Bu problemi həll etmək üçün *minimal elementin seçilməsi* üsulundan istifadə etmək lazımdır. Bu üsul aşağıdakından ibarətdir: massivdə minimal elementi təyin edib onu birinci yerə qoyuruq. Sonra qalan elementlər içindən minimal elementi tapıb, onu növbəti yerə qoyuruq və s.

Hava qabarcığı üsuldan fərqli olaraq burada yerdəyişmələrin sayı kifayət qədər azdır (ən pis halda **N-1**). Proqramın icrası müddəti azalır.

```
#include <stdio.h>
const int N = 10;
main ()
{
int i, j, nMin, A[N], c;
// Burada A massivi daxil edilməlidir
```

```
for ( i=0; i<N-1; i++)
   {
    nMin = i;                    //  A[i]-dən başlayaraq minimal elementi təyin edirik
    for ( j = i+1; j<N; j++ )
       if ( A[j] < A[nMin] )
          nMin = j;
    if ( nMin !=i )              // əgər minimal element öz yerində dayanmayıbsa, ...
       {
       c = A[i]; A[i] = A[nMin]; A[nMin] = c;   //  onu lazımı yerə qoyuruq
       }
     }
printf ( "\n Cheshidlenmish massiv: \n" );
for ( i=0; i<N-1; i++ )
  printf ( "%d  ", A[i] );
}
```

Massivdə binar axtarışı

Tutaq ki, **A** massivin elementləri artmaya görə nizamlanıb və **L**, **R** nömrəli elementlər arasında **x**-a bərabər olan elementi tapmaq lazımdır. Bunun üçün belə bir alqoritmdən istifadə edirlər: **L** və **R** arasında yerləşən orta elementi təyin edirik. Onun nömrəsi **m=(L+R)/2**-dir (burada bölmə tam nəticə verir). Alınmış A[m] elementi verilmiş x-lə müqayisə edirik. Əgər x=A[m] olarsa, onda proses dayandırılır. Əgər x<A[m] olarsa, onda axtarış A[L] və A[m] arasında aparılır. Əgər x>A[m] olarsa, onda axtarış A[m] və A[R] arasında aparılır.

```
#include <stdio.h>
const int N =10;
main ()
{
int L = 0, R = N-1, m, A[N],
  flag = 0;                    // bayraq dəyişəni. Tapıldı (1), tapılmadı – (0)
// Burada A massivi daxil etmək lazımdır
// Burada A massivi artmaya görə çeşidlənməlidir
printf ( "Axtarilan elementi daxil edin\n" );
scanf ( "%d", &x );
while ( L <= R )
   {  m = (L + R )/2;             // intervalın mərkəzi
   if ( A[m] == x )            // element tapıldı
      { flag = 1;            // bayraq =1
       break;                 // dövr dayandırılır
      }
    if  ( x < A[m] )    R = m – 1;    // axtarış sərhədlərini daraldırıq
    else              L  = M + 1;
    }
```

```
if ( flag )
    printf ( " Tapildi: A[%d] = %d", m, A[m] );
else printf ( "Bele element yoxdur!" );
}
```

flag adlı dəyişən (bayraq) elementin tapılmasını bildirir. Əgər **x**-a bərabər element tapılırsa, onda bayrağa 1 qiyməti verib dövrdən çıxırıq. Tapılmış elementin nömrəsi **m** dəyişənində qalır.

Əgər massiv kiçikdirsə, onda binar axtarışın sürəti xətti axtarışın sürətindən o qədər də fərqlənmir. Tutaq ki, massivin ölçüsü 1000000 elementdən ibarətdir və bizə lazım element massivdə axırıncıdır (xətti axtarış üçün ən pis variant).

Massivin ölçüsü, N	**Müqayisələrin sayı**	
	Xətti axtarış	**Binar axtarışı**
10	≤ 10	≤ 4
1000	≤ 1000	≤ 10
1000000	≤ 1000000	≤ 20
N	≤ N	$\le \log_2 N$

Beləliklə, massivin elementlərin sayı artdıqca, binar axtarışın səmərəliliyi artır. Binar axtarışın çatışmayan cəhəti ondan ibarətdir ki, massivin qabaqcadan nizamlanmış olması gərəkdir. Binar axtarışı böyük ölçülü massivlər üçün istifadə edilir.

Proseduralarda və funksiyalarda massivlərin istifadəsi

Massivləri sadə dəyişənlər kimi proseduralara və funksiyalara parametr kimi ötürmək mümkündür. Massivin elementlərinin cəmini hesablayan funksiyanı nəzərdən keçirək. Funksiya elə tərtib olunmalıdır ki, ona istənilən ölçüdə massivi ötürmək olsun. Bunun üçün funksiyada massivin ölçüsü müəyyən olunmalıdır. **C** dilində funksiyalar massivin ölçüsünü müstəqil təyin edə bilmirlər, ona görə massivin ölçüsü parametrlərdən biri olmalıdır.

```
int Sum ( int A[ ], int N )     //  A[ ]  - parametr_massiv, N – massivin ölçüsü
{
int i, sum;
sum = 0;
for ( i=0; i<N; i++ )
  sum += A[i];
return sum;
}
```

Qeyd edək ki, funksiyanın başlığında massivi **A[N]** parametri kimi təyin etmək olmaz. Massivin ölçüsü (**N**) ayrıca parametr kimi verilməlidir, massiv isə **A[ ]** kimi təyin olunmalıdır. Digər tərəfdən, bu yazılış yalnız funksiyaların başlığında mümkündür, yəni massiv üçün yaddaşda yeni yer ayrılmır. Lokal və ya qlobal massivlərin ölçülərini isə mütləq göstərmək lazımdır.

Funksiyaya müraciət etmək üçün onun faktiki parametrlərində mötərizələr yazılmadan massivin adı və onun ölçüsü göstərilməlidir.

```
main ()
{
 int A[20], B[30], s;
// Burada A və B massivləri daxil olunmalıdır
s = sum (A, 20 );      //  A massivinin elementlərinin  cəmini hesablayırıq
printf ( “A massivin cemi  = %d, B massivin cemi = %d”, s, Sum(B, 30) );    // B massivinin
                                                     // elementlərinin cəmini çap operatorunda hesablayırıq
}
```

13. SİMVOLLAR SƏTİRLƏRİ

Simvol sətri nədir?

Simvol sətri simvollar ardıcıllığıdır. Hər simvola 1 bayt yer ayıran simvol sətirlərini nəzərdən keçirək. Bu halda 2^8=256 simvoldan istifadə etmək lazımdır. Hər simvolun 0-dan 255-ə qədər dəyişən kodu var. Bu kodlar ASCII cədvəlində yerləşir.

Digər dəyişənlər kimi sətir yaddaşa yazılır, və sətrə hansı verilənlərin yazılması kompüter üçün fərq etmir. Kompüter üçün sətir bayt ardıcıllığıdır. Onda bəs sətrin sonunu necə təyin etmək olar? Bunun üçün 2 həll yolu vardır:

1. Sətrin uzunluğunu ayrıca yaddaş xanasında saxlamaq;
2. sətrin sonunu göstərən xüsusi simvol seçmək, özü də bu simvoldan sətrin ortasında istifadə oluna bilməz.

C dilində ikinci yanaşmadan istifadə olunur.

Sıfır (0) kodlu simvolla bitən simvollar ardıcıllığı **simvol sətri** adlanır.

Sıfır kodlu simvolun təsviri yoxdur, proqramda onu '\0' kimi yazırlar.

Sifir kodlu simvol ('\0') və 0 rəqəmi ('0') müxtəlif simvollardır.

Sətirlərin elanı və başlanğıc qiymətlərin verilməsi

Sətir simvollar massividir, ona görə də sətirləri massiv kimi elan etmək lazımdır:

```
char s[80];
```

Lakin massivlərdən fərqli olaraq sətir '\0' simvolla bitməlidir.

Əgər simvollar massivi sətir kimi istifadə olunacaqsa, onda onun elanı üçün 1 bayt artıq yer verilməlidir.

Yaddaş ayrılarkən qlobal dəyişənlər sıfırlarla tamamlanır, lokal dəyişənlər isə "yaddaş zibilini" özündə saxlayırlar. Sətrin başlanğıc qiymətini elan zamanı vermək olar.

```
char s[80] =  "Salam dostlar!" ;
```

Kompilyator dırnaqda yazılmış simvolları hesablayıb, onların sayının üzərinə 1 gəlib yaddaşda o sayda yer ayırıb sətri və sıfır-simvolunu ora yazacaqdır. Analoji olaraq göstərici üçün yer ayırmaq olar:

```
char *s = "Salam dostlar!" ;
```

Nəticə - eynidir, sadəcə olaraq, indi **s** – göstəricidir (yaddaş ünvanını özündə saxlayan dəyişən) və onunla adi göstərici kimi işləmək olar (mənimsətmək, dəyişmək və s.). Əgər proqramın icrası zamanı sətir dəyişməyəcəkdirsə, onda onu sabit kimi elan etmək olar:

```
const char SALAM[ ] = "Salam dostlar!" ;
```

Standart daxiletmə və xaricetmə

Sətirlərin daxil və xaric edilməsi üçün **scanf** və **printf** funksiyalarında xüsusi “**%s**” formatından istifadə olunur:

```
#include <stdio.h>
main ( )
{
  char Name[50];
  printf ( “what is your name? “);
  scanf ( “%s”, Name );
  printf ( “Hello, %s!”, Name );
}
```

Qeyd edək ki, **scanf** funksiyasına sətrin adını ötürmək lazımdır (**&** işarəsiz), çünki massivin adı, eyni zamanda, onun birinci elementinin ünvanıdır.

Lakin **scanf** funksiyasının bir çatışmayan cəhəti var: birinci boşluğa rast gələndə daxiletmə dayanır. Məsələn, əgər əvvəlki misalda “**Anar Samadov**” daxil etsək, onda ekranda “**Hello, Anar!**” görəcəyik (sətrin yarısı oxunmur). Əgər sətrin bütövlüklə daxil olunması tələb edilirsə (boşluqlar daxil olmaqla), onda **scanf** funksiyasının əvəzinə **gets** (ing. *get string*) funksiyasından istifadə edilməlidir.

```
gets ( s );
```

Sətrin ekrana çıxarılması üçün **printf**-dan əlavə **puts** funksiyasından istifadə etmək olar. Bu funksiya sətri ekrana çıxardandan sonra kursoru növbəti sətrin əvvəlinə gətirir. Aşağıdakı misalda **Name** sətri ekranın yeni sətrindən çap olunacaqdır.

```
#include <stdio.h>
main ()
{
   char Name[50] = “Anar!”;
   puts ( “Hello, ”);
   puts ( Name );
}
```

Misal. Simvollar sətrini daxil edib, həmin sətirdə ‘**A**’ hərfini ‘**C**’ hərfi ilə əvəz etmək.

Simvollar sətrinə simvollar massivi kimi baxaq. Massivin bütün elementlərini yoxlayaraq, həmin massivdə ‘**A**’ hərfini ‘**C**’ hərfi ilə əvəz etməliyik. Dövr ‘\0’ simvola rast gələnə kimi işləməlidir. Sətrin uzunluğu qabaqcadan bilinmədiyi üçün **while** dövr operatorundan istifadə olunmalıdır.

```
#include <stdio.h>
main ( )
{
  char s[80];
  int i;
```

```
  printf ( "\n Setri daxil edin \n" );
  gets ( s );
  i =0;                    //  birinci s[0] simvolundan başlamaq
  while ( s[i] !='\0' )    //  hələ ki sətrin sonu deyil
   {
    if ( s[i] == 'A' )     // əgər növbəti simvol "A" deyilsə, ...
      s[i] = 'C';          // onu "C" dəyişirik
    i ++;                  // növbəti simvola keçid alırıq
    }
  puts ( "Netice: \n" );
  puts ( s );
}
```

Qeyd edək ki, tək simvol apostroflarla, simvollar sətri isə dırnaq içində yazılır.

Sətirləri **printf** funksiyası vasitəsi ilə xaric edən zaman, formatlamadan istifadə edilir. **'%'** işarəsindən sonra sətrin xaric edilməsi üçün sahənin ölçüsü göstərilir. Bu işarədən qabaq minus işarəsini ('-') yazmaq olar, yəni sətri sol tərəfdən yazmaq.

Misal	Nəticə	Şərh
printf (" [%s]", "Hello");	[Hello]	Pozisiyaların minimal sayı
printf ("[%6s]", "Hello");	[Hello]	6 pozisiya, sağdan düzənnəmə
printf ("[%-6s]", "Hello");	[Hello]	6 pozisiya, soldan düzənnəmə
printf ("[%2s]", "Hello");	[Hello]	Sətir ona ayrılan 2 pozisiya yerləşmədiyinə görə sahə artırılır

Fayllarla işləmə

Real şəraitdə emal olunan sətirlərin sayı çox olur və bu məlumatlar, çox vaxt, fayllarda yerləşir. Adətən, faylda yerləşən sətirlərin sayı qabaqcadan məlum deyil. Əgər bir sətrin emalı digər sətirlərdən asılı deyilsə, onda ölçüsü məlum olmayan massivlərin emalında istifadə olunan üsullardan burada da istifadə etmək olar. Fayldan sətirləri bir-bir oxuyub, onları emal edib, çıxış faylına yazmaq olar.

Fayllarla işləmənin bir neçə xüsusiyyətləri var. Birincisi, sətrin oxunması üçün **fscanf** funksiyasından istifadə etmək olar. Lakin, bu funksiya bir sözü oxuyub birinci boşluqda dayanacaqdır.

Əgər fayldan məlumatlar sözbə-söz oxunmalıdırsa, onda **fscanf** funksiyasından istifadə etmək olar.

```
 #include <stdio.h>
main ( )
{
   char s[80];
   FILE *fp;
   fp = fopen ( "input.dat", "r" );
```

```
   fscanf ( fp, "%s", s );
   printf ( "Setrin birinci sozu - %s", s );
   fclose ( fp );
}
```

Əgər sətir boşluqlarla bir yerdə bütövlüklə oxunmalıdırsa, onda **fgets** funksiyasından istifadə olunmalıdır. Bu funksiyanın 3 parametri var:

- **simvol sətrinin adı**;
- **sətrin maksimal uzunluğu**. Əgər oxunan sətrin uzunluğu göstərilən parametrdən çoxdursa, onda sətrin bir hissəsi oxunur, qalan hissəsi isə **fgets** funksiyasının növbəti çağırışlarına saxlanılır.
- **fayl göstəricisi**.

Əgər **fgets** funksiyası fayldan sətri oxuya bilmirsə və ya faylda sətirlər qurtarıbsa, onda nəticə olaraq o, **NULL** qiymətini qaytarır. Funksiyanın bu xassəsini səhvlərin emalında və daxiletmənin dayandırılmasında istifadə etmək olar.

```
#include <stdio.h>
main ( )
{
   char s[80];
   FILE *fp;
   fp = fopen ( "input.dat", "r" );
   if ( NULL == fgets ( s, 80, fp ) )
      printf ( "Setir oxunmur!" );
   else
      printf ( "Faylin birinci setiri - %s", s );
   fclose ( fp );
}
```

fgets funksiyası fayldan sətir oxuyur. O, öz işini aşağıdakı hallarda dayandıra bilər:

- növbəti sətrə keçmə - '\n' simvoluna rast gələndə;
- **fgets** funksiyanın parametrində göstərilən simvolların hamısını oxuyub qurtaranda. Bizim misalda sıfır-simvolu ilə birlikdə 79 simvol.

Sətrin sonunda '\0' simvolu qoyulacaqdır. Əgər oxunan simvolların içində növbəti sətrə keçmə - '\n' simvolu vardırsa, o, **s** sətrində saxlanılacaqdır.

Misal. **input.dat** faylın hər sətrində '**A**' hərifləri '**C**' hərfləri ilə əvəz edib nəticəni **output.dat** faylına yazmaq.

Qeyd edək ki, fayl istənilən uzunluqda ola bilər. Lakin sətirlər ardıcıl oxunur və növbəti sətrin emalı üçün əvvəlki və sonrakı sətirlər haqqında məlumatlar lazım deyil.

Fayldan məlumatları oxumaq üçün **while** dövr operatorundan istifadə olunmalıdır. Sətirlər qurtarandan sonra fgets funksiyası **NULL** qiymətini alır və dövr öz işini bitirir.

```
#include <stdio.h>
main ()
{
```

```
  char s[80];
 int i;
 FILE *fin, *fout;
 fin  =fopen ( “input.dat”, “r” );
 fout =   fopen ( “output.dat” , “w” );
 while ( NULL != fgets (s, 80, fin ) )     // sətri oxuyuruq
 {
  i = 0;                                    //  s[0] –dan başlayırıq
  while ( s[i] !=’\0’ )                    // hələ ki sətrin sonu deyil,
    {
     if  ( s[i] == ‘A’ ) s[i] = ‘C’;           // simvolu dəyişirik
     i ++;                                  // növbəti simvola keçirik
     }
   fprintf ( fout, “%s”, s );             // sətri fayla yazırıq
   }
 fclose ( fin );
 fclose ( fout );
}
```

Qeyd edək ki, **fprintf** funksiyasında **‘\n’** simvoluna ehtiyac yoxdur, çünki **fgets** funksiyasından istifadə zamanı o, hər oxuduğu sətrin sonuna **‘\n’** simvolunu əlavə edir. Ona görə də sətirlər çıxış faylına olduğu kimi çıxarılacaqdır.

Sətirlərlə işləyən funksiyalar

Sətirlərlə işləmək üçün C dilində çoxlu sayda funksiyalar var. Bu funksiyaları tətbiq etmək üçün proqrama **string.h** başlıq faylı əlavə olunmalıdır.

```
#include <string.h>
```

Sətrin uzunluğu – strlen

strlen funksiyası (ing. ***str**ing* ***len**gth*) ‘\0’ simvolu nəzərə almayaraq sətirdəki simvolların sayını təyin edir.

```
#include <stdio.h>
#include <string.h>
main ()
{
  int len;
  char s[ ] = “Azerbaijan” ;
  len = strlen (s);
  printf ( “%s setrin uzunlugu beraberdir %d”, s, len );
}
```

Bu misalda **strlen** funksiyası "Azerbaijan" sətrinin uzunluğunu təyin edəcəkdir. Cavab 10 olacaqdır. İndi isə daha mürəkkəb məsələyə baxaq:

Misal. **input.dat** faylına 2 sətirli mətn yazılıb. Sətirlərin uzunluğunu təyin edib, onları alt-alta **output.dat** faylına yazmaq.

```
#include <stdio.h>
#include <string.h>
main ()
{
  char s[80];
  FILE *fin, *fout;
  fin = fopen ( "input.dat", "r" );
  fout = fopen ( "output.dat", "w" );
  while ( NULL != fgets (s, 80, fin) )      //  s sətrini oxuyuruq
       {
        fprintf (fout, "%d\n", strlen(s) );  // uzunluğu təyin edib fayla yazırıq
        }
  fclose( fin );
  fclose ( fout );
}
```

Baxmayaraq ki, proqram düz yazılıb, çıxış faylına yazılmış ədədlər real qiymətlərdən (axırıncı sətrin uzunluğundan başqa)1 vahid artıq olacaqlar. Növbəti misalda bu məsələyə aydınlıq gətirəcəyik.

Sətirlərin müqayisəsi – strcmp

İki sətrin müqayisəsi üçün **strcmp** funksiyasından (ing. ***str**ing **comp**arison*) istifadə edilir. Əgər sətirlər eynidirsə, onda funksiyanın qiyməti sıfıra bərabərdir. Əgər sətirlər fərqlidirsə, onda funksiya sıfırdan fərqli qiymət qaytarır. Müqayisə sətirlərdə olan simvolların kodlarına görə aparılır, ona görə də strcmp funksiyası kiçik və baş hərifləri fərqləndirir (onların kodları fərqlidir).

```
#include <stdio.h>
#include <string.h>
main ()
{
char s1[ ] = "Tarix";
       s2[ ] = "Fizika";
if ( 0 == strcmp(s1, s2) )
      printf (" %s ve %s setirleri eynidir", s1, s2 );
else
      printf (" %s ve %s setirleri ferqlidir", s1, s2 );
}
```

Əgər sətirlər fərqlidirsə, onda funksiya nəticə olaraq sətirlərin “fərqini”, yəni *birinci fərqli simvolların kodlarının fərqini* göndərir. Bu nəticələri sətirlərin çeşidlənməsi zamanı istifadə etmək olar. Əgər “fərq” mənfidirsə, bu o deməkdir ki, birinci sətir əlifba sırasına görə ikincidən sonra gəlir. Aşağıdakı cədvəldə bir neçə misal göstərilmişdir (‘**A’** hərifin kodu **65**, ‘**B**’-nin kodu – **66**, ‘**C**’-nin kodu – **67-**dir).

s1	**s2**	**strcmp(s1, s2)**
AA	AA	0
AB	AAB	‘B’ –‘A’ = 66 – 65 =1
AB	CAA	‘A’ –‘C’ = 65 – 67 = -2
AA	AAA	‘\0’ – ‘A’ = -65

Misal. İki sətri daxil edib, onları əlifba sırasına düzüb ekrana çıxartmaq lazımdır.

```
#include <stdio.h>
#include <string.h>
main ()
{
  char s1[80], s2[80];
  printf ( “Birinci setri daxil edin “ );
  gets ( s1 );
  printf ( “Ikinci setri daxil edin “ );
  gets ( s2 );
  if (strcmp(s1, s2) <= 0 )
     printf ( “%s\n%s”, s1, s2 );
  else
     printf ( “%s\n%s”, s2, s1 );
}
```

Bəzən iki sətrin müqayisəsi zamanı bütün sətri yox, birinci bir-neçə simvolu müqayisə etmək lazım gəlir. Bunun üçün **strnsmp** funksiyasından istifadə edirlər. Bu funksiyanın 3 parametri var: müqayisə olunan sətirlərin adları və müqayisə olunan simvolların sayı. **strncmp** funksiyasının iş prinsipi **strcmp** funksiyasının iş prinsipi ilə eynidir. Əgər müqayisə olunan simvollar eynidirsə, onda nəticə sıfır olur.

```
#include <stdio.h>
#include <string.h>
main ()
  { char s1[80], s2[80];
   printf ( “Birinci setri daxil edin: ” );
   gets ( s1 );
   printf ( “Ikinci setri daxil edin: “ );
   gets ( s2 );
   if ( 0 == strncmp (s1, s2, 2 ) )
     printf ( “ %s ve %s setrlerin birinci 2 simvolu eynidir”, s1, s2 );
   else
     printf (“ %s ve %s setrlerin birinci 2 simvolu ferqlidir”, s1, s2 );
}
```

strcmp funksiyasının tətbiq sahələrindən biri parolların yoxlanılmasıdır. Parolu soruşan və parol düz daxil olunubsa müəyyən əməliyyatları yerinə yetirən proqramı tərtib edək.

Misal. Sətirdə rəqəmlərin sayını hesablayan proqramı tərtib etmək lazımdır. Proqram yalnız "password" parolu daxil olduğu halda işləməlidir.

```
#include <stdio.h>
#include <string.h>
main ()
{
   char pass[ ] = "password",     // düz parol
         s[80];                          // əlavə sətir
   int i, count = 0;
   printf ( "Parolu daxil edin: ") ;
   gets ( s );
   if ( strcmp( pass, s) !=0 )
      {
      printf ( "Parol sehvdir!" );
      return 1;                          // səhv olduğu halda proqramdan çıxış
      }
    printf ( "Setri daxil edin: " );
    gets (s);
   i = 0;
   while ( s[i] !='\0' )
     { if (s[i] >='0'  && s[i] <= '9' )
         count ++;
      }
   printf ( "\nSetirde %d reqem var", count );
}
```

Bu proqramda o faktdan istifadə edilib ki, rəqəmlər simvollar cədvəlində ardıcıl yerləşir, yəni **'0'**-dan **'9'**-kimi. Qeyd edək ki, **'\0'** simvolun kodu **0**-dır, **'0'** simvolun kodu (sıfır rəqəmi) **48**-dir. **count** dəyişəni sayğac rolunu oynayır.

Sətirlərin kopyalanması - strcpy

Çox vaxt sətrə yeni qiymət yazmaq və ya digər sətirdən məlumatları köçürmək lazım gəlir. Köçürülmə funksiyalardan istifadəsi zamanı ciddi səhvlərə rast gəlmək olur. Əgər sətrə yazılan məlumatlar sətrin ölçüsündən böyükdürsə, onda məlumatlar massivin sərhədlərindən kənar yazılacaqdır.

Kopyalanmada iki sətir iştirak edir. Biri "*mənbə*" (məlumatlar bu sətirdən götürülür), digəri isə "*qəbuledici*" (məlumatlar bu sətrə yazılır) sətir adlanır.

Kopyalanma zamanı əmin olmaq lazımdır ki, qəbuledici-sətrin uzunluğu kifayət qədər olsun.

İki sətrin köçürülməsi üçün **strcpy** funksiyasından istifadə edilir. Funksiyanın 2 parametri var: qəbuledici sətir və mənbə.

```
char s1[50], s2[10];
gets (s1);
strcpy ( s2, s1 );        // s2 (qəbuledici sətir) ← s1 (mənbə)
puts ( s2 );
```

Proqramın bu fraqmenti çox “təhlükəlidir”, çünki **s1** sətrə *49* simvol yazmaq olar (axırıncı simvol ‘\0’-dır). Əgər **s1**-ə **9**-dan artıq simvol daxil olunarsa (**s2**-nin uzunluğu **9** simvoldur+‘\0’), onda proqram səhvlə nəticələnəcəkdir.

Belə ki, funksiyaya sətrin əvvəlinin ünvanı ötürülür, onda köçürməni istənilən simvoldan başlamaq olar. Məsələn, aşağıdakı cümlədə s2 sətri s1 sətrinə 6-ci simvoldan başlayaraq köçürülür. s1-də birinci 5 simvol dəyişilməz olaraq qalır.

```
strcpy ( s1 + 5, s2 );
```

Hər iki sətirdə simvolların sayını nəzərə almaq lazımdır. Əgər s1 sətrində 5 simvoldan azdırsa, onda köçürülmə baş verməyəcəkdir.

İkinci sətirdən birinciyə müəyyən sayda simvolların köçürülməsi üçün **strncpy** funksiyasından istifadə edilir. Bu funksiyanın üçüncü parametri var – köçürülən simvolların sayı. Qeyd edək ki, **strcpy**-dan fərqli olaraq bu funksiya **sonuncu ‘\0’ simvolu köçürmür**. Əgər hər hansı sətri bir neçə hissədən yığmaq lazım gələrsə, onda **strncpy** funksiyasından istifadə edilməlidir.

```
#include <stdio.h>
#include <string.h>
main ()
{
   char s1[ ] = “Salam dostlar!”, s2[10];
   strncpy( s2, s1, 5 );   //  s1-dən 5 simvolu s2 köçürmək
   puts ( s2 );            //  səhvdir! Sətrin sonu, yəni ‘\0’ simvolu yoxdur!
   s2[5] = ‘\0’;           //  sətrin sonunu göstərən simvolu əlavə edirk
   puts ( s2 );            //  sətrin çapı
}
```

Sətirlərin birləşdirilməsi - strcat

strcat funksiyası (ing. *string concatenation*) ikinci sətri birinci sətrin sonuna əlavə edir (yekunlaşdırıcı ‘\0’ simvolu avtomatik yazılır). Nəzərə almaq lazımdır ki, s2 sətrini özünə birləşdirmək üçün s1 sətri kifayət qədər uzun olmalıdır. strcat funksiyası yekunlaşdırıcı ‘\0’ simvolu alınmış sətrə avtomatik yazır.

```
#include <stdio.h>
#include <string.h>
main ()
{
    char s1[80] = “Baku ”,
         s2 [ ] = “Higher “,
         s3 [ ] = “Oil School”;
```

```
   strcat (s1, s2);          //  s1-in sonuna s2-i əlavə etmək
   puts (s1);                //  “Baku Higher ”
   strcat (s1, s3 );         // s1-in sonuna s3-ü əlavə etmək
   puts (s1);                // “Baku Higher Oil School”
}
```

Qeyd edək ki, əgər **s1** sətri **s1[ ]** kimi elan olunsaydı, onda əlavə olunan sətir yaddaşın ayrılmış hissəsinə yerləşməyəcəkdir və nəticə səhv olacaq idi.

Misal. Klaviaturadan faylın adını daxil edib onun genişlənməsini **“.exe**”-ya dəyişmək lazımdır.

Məsələnin həll alqoritmi aşağıdakı kimi olacaqdır:

1. Faylın adında nöqtəni ‘.’ və ya sətrin sonunu göstərən simvolu ‘\0’ aşkar etmək;
2. Əgər nöqtə aşkar olunubsa, onda həmin mövqedən başlayaraq **strcpy** funksiyasından istifadə edərək **“.exe**” genişlənməsini kopyalamaq;
3. Əgər sətrin sonu aşkar olunubsa (nöqtə yoxdur), onda **strcat** funksiyasından istifadə edərək sətrin sonuna **“.exe**” genişlənməsini əlavə etmək.

```
#include <stdio.h>
#include <string.h>
main ()
{
   char s[80];
   int n;               // ’.’ simvolun nömrəsi
   printf ( “Faylin adini daxil edin: “ );
   gets (s);
   n = 0;
   while ( (s[n] !=’.’ )  && (s[n] !=’\0’) )    // birinci nöqtəyə və ya sətrin sonununa kimi
      n++ ;
   if  (s[n] == ‘.’ )                  // əgər nöqtə aşkar olunubsa, onda …
         strcpy (s+n, “.exe” );     // genişlənməni dəyişirik,
  else
         strcat (s, “.exe” );   //    əks halda əlavə edirik
   puts (s);
}
```

Misal. Klaviaturadan soyadı və adı bir sətrə daxil edib (məsələn, “**Aliyev Ali**”), ekranda dəyişdirilmiş sətri “**Salam, Ali Aliyev!**” əks etdirmək.

Məsələnin həll alqoritmini aşağıdakı kimi yazmaq olar:

1. Sətri (s) klaviaturadan daxil etmək:

s	A	l	i	y	e	v		A	l	i	\0								

2. Soyadın uzunluğunu təyin edib onu **n**-ə mənimsətmək. Onda **n** nömrəli simvol - soyad və ad arasında olan boşluqdur.

							n												
s	A	l	i	y	e	v		A	l	i	\0								

3. **strcpy** funksiyasından istifadə edərək yeni sətrə cümlənin birinci hissəsini ("Salam, ") yazaq:

a	S	a	l	a	m	,		\0										

4. **strcat** funksiyasından istifadə edərək yeni sətrin sonuna ikinci sözü, yəni adı əlavə edək. Bunun üçün a sətrin axırına **s** sətrin **n+1**-ci simvolundan başlayaraq bütün simvolları əlavə etmək lazımdır.

a	S	a	l	a	m	,		A	l	i	\0							

5. **strcat** funksiyasından istifadə edərək cümlənin sonuna boşluq əlavə edək.

a	S	a	l	a	m	,		A	l	i		\0						

6. Alınmış cümlənin uzunluğunu təyin edib onu **len** adı ilə yadda saxlayaq. **len** nömrəli simvol **'\0'** simvoludur.

												len						
a	S	a	l	a	m	,		A	l	i		\0						

7. **strncpy** funksiyasından istifadə edərək alınmış cümlənin sonuna birinci sözü, yəni soyadı əlavə edək. Bunun üçün **a** sətrin sonuna **s** sətrin birinci **n** simvolu əlavə edək. Qeyd edək ki, indi **a** sətrin sonunda sətrin sonunu göstərən **'\0'** simvolu yoxdur.

																	len+n	
a	S	a	l	a	m	,		A	l	i		A	l	i	y	e	v	?	?

8. **strcpy** funksiyasından istifadə edərək **a** sətrin sonuna **'!'** və **'\0'** simvolları əlavə edək. Belə ki, **len** uzunluqlu sətrə **n** uzunluqlu soyad əlavə olunursa, simvolların köçürülməsi üçün **s+len+n** ünvanından istifadə olunmalıdır.

a	S	a	l	a	m	,		A	l	i		A	l	i	y	e	v	!	\0

Proqramda C dilinin massivlərlə işləmək üçün çox vacib xüsusiyyətindən istifadə edilmişdir.

Əgər massivin adı **s**-dırsa, onda **s+i** yazısı **s[i]** elementin ünvanı **(&s[i])** deməkdir.

Belə ki, **strcpy** və **strcat** funksiyalara köçürülən verilənlərin ünvanlarını ötürmək lazımdır, **&a[len]** yerinə **a+len** yazılışından istifadə etmək olar.

```
#include <stdio.h>
#include <string.h>
main ()
{
```

```
  char s[80], a[80]  =”Salam, “;
  int n, len;
  printf ( “Soyadi ve adi daxil edin “ );
  gets (s);
  n = 0;
  while ( (s[n] != ‘ ‘)                  //  birinci boşluğu
          && (s[n] !=’\0’) )          //  və ya sətrin sonunu axtarırıq
     n ++;
  if ( s[n] !=’ ‘ ) {                 // əgər boşluq yoxdursa, …
       printf ( “Setr duz deyil” );
       return 1;                     // səhvə görə proqramdan çıxış.
       }
  strcat (a, s+n+1);            // adı əlavə etmək
  strcat (a, “ “);                  // boşluğu əlavə etmək
  len = strlen (a);               // sətrin uzunluğunu müəyyən etmək
  strncpy (a+len, s, n);        // soyadı əlavə etmək
  strcpy (a+len+n, “!” );        //  “!” işarəsini əlavə etmək
  puts (a);
}
```

Sətirlərdə axtarış

Sətirlərdə axtarış haqqında danışanda, adətən iki məsələyə baxılır:

1. sətrinin əvvəlindən və ya sonundan başlayaraq verilmiş simvolu aşkar etmək;
2. Verilmiş alt sətri sətirdə aşkar etmək.

Birinci məsələnin həlli üçün **strchr** (sətrin əvvəlindən axtarış) və ya strrchr (sətrin sonundan axtarış) funksiyalarından istifadə edilir. İkinci məsələnin həlli üçün **strstr** funksiyasından istifadə edilməlidir.

Nəticə olaraq, bütün bu funksiyalar aşkar edilmiş simvolunun (və ya alt sətrinin birinci simvolunun) göstəricisini qaytarırlar. Bu o deməkdir ki, nəticə simvol tipli dəyişənin göstəricisinə yazılmalıdır. **Göstərici** – digər dəyişənin *ünvanını* özündə saxlayan yaddaş xanasıdır.

strchr, strrchr, strstr funksiyaların 2 parametri var: birinci parametr - sətirdir (axtarışı harada etmək lazımdır), ikinci parametr - axtardığımız simvol (**strchr, strrchr** funksiyalar üçün) və ya alt sətirdir (**strstr** funksiyası üçün). Sətrin əvvəlindən başlayaraq simvolun nömrəsini hesablamaq üçün göstəricidən massivin başlanğıc elementin ünvanını çıxmaq lazımdır. Əgər axtarış nəticə vermirsə, onda bütün funksiyalar **NULL** qaytarırlar.

```
#include <stdio.h>
#include <string.h>
main ()
{
char s1[ ] = “Baku is the capital of Azerbaijan”,
     s2[ ]  =”Baku Higher Oil School”, *p;
   p = strchr (s1, ‘a’ );
   if ( p !=NULL )
```

```
    {  printf ( " Birinci a herifi: pozisiya %d", p-s1);
    p = strrchr (s1, 'a' );
    printf ( "\nAxirinci a herfi: pozisiya %d", p-s1);
    }
 p = strstr (s2, "School");
 if ( p != NULL )
     printf ( "\n%s setrinde School sozu ashkar edildi!", s2);
 else
     printf ( "\n%d setrinde School sozu yoxdur!", s2 );
}
```

Qeyd. **fgets** funksiyası vasitəsi ilə fayldan sətirləri oxuyanda hər sətrin sonunda əlavə '\n' (yeni sətrə keçmə) simvolu qoyulur. Çox vaxt bu simvol lazım deyil və onu silib onun yerinə sətrin sonunu göstərən simvolu qoymaq lazım gəlir. Bunu belə etmək olar:

```
char s[80], *p;
....
p = strrchr (s, '\n');    //  '\n' simvolunu axtarırıq
if ( p != NULL)           //  əgər aşkar olunubsa, ...
   *p = '\0';             // onu '\0' simvolla əvəz edirik
```

İndi isə daha mürəkkəb məsələni həll edək.

Misal. Klaviaturadan bir cümlə və bir söz daxil olunur. Daxil edilmiş sözün verilmiş cümlədə neçə dəfə təkrar olunduğunu təyin etmək lazımdır.

strstr funksiyası sözün yalnız birinci girməsini müəyyən edə bilər, ona görə də bu məsələni bir addımla həll etmək olmaz. Aşağıdakı alqoritmdən istifadə etmək lazımdır.

Əgər sətirdə ilk dəfə verilmiş söz aşkar edilirsə, onun ünvanı **p** adlı göstəriciyə yazılır. Sözün növbəti axtarışını sətrin əvvəlindən yox, **p+sözün_uzunluğu** ünvanından aparmaq lazımdır. Bu əməliyyatı dövrdə təkrar edərək **strstr** funksiyası vasitəsi ilə sətrin qalan hissəsində də axtarışı aparmaq olar. Belə ki, axtarış sahəsinin başlanğıc ünvanı həmişə dəyişir, bu ünvanı ayrı simvol tipli göstərici-dəyişənində saxlamaq məsləhətdir. Proqram aşağıdakı kimi yazıla bilər:

```
#include <stdio.h>
#include <string.h>
main ()
{  int len, count;
   char s[80], word[20],
        *p,                  //  aşkar edilmiş sözün göstəricisi
        *start;              // axtarış sahəsinin başlanğıc ünvanı
puts ( "Cumleni daxil edin:" );
gets ( s );
puts ( "Axtarish uchun sozu daxil edin: ");
gets (word);
len = strlen (word);          // axtarılan sözün uzunluğu
count = 0;                    // aşkar edilmiş sözlərin sayı
start  =s;                    // axtarış cümlənin əvvəlindən başlayır
```

```
while ( 1 ) {
    p  = strstr (start, word );     // cümlədə yenə də söz var, yoxsa yox
    if ( p == NULL ) break;       // əgər yoxsa, onda çıxış
    count ++;                    // sayğac artırılır
    start = p + len;             // axtarış ünvanı sürüşdürülür
    }
 printf ( "Bu cumlede %s sozune %d defe rast gelmek olar", word, count );
}
```

Sətirlərin formatlanması

Çox vaxt sətrin ekrana çıxarılmasından əvvəl ona müəyyən məlumatları əlavə etməyə lazım gəlir. Məsələn, səhv haqqında məlumat standart funksiya vasitəsi ilə ekrana çıxarılır və bu məlumata ədədləri daxil edib yeniləşmiş məlumatı ekranda təsvir etmək lazımdır. Digər misal – qrafiki rejimdə mətnin ekrana çıxarılması. Bunun üçün **printf** funksiyası yaramır.

Yuxarıda göstərdiyimiz misallar üçün **sprintf** funksiyasından istifadə edilməlidir. Bu funksiya **printf** funksiyasının dəstəklədiyi formatlardan istifadə edir, lakin nəticəni ekrana və ya fayla yox, *simvol sətrinə* yazır (sətir üçün qabaqcadan yer ayrılmalıdır). Qrafiki rejimdə **(x,y)** koordinatlarının qiymətlərinin ekrana çıxarılması aşağıdakı kimi olmalıdır:

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
main ()
{
  char s[80];                          // əlavə sətir
  int x, y;
  initwindow (300,400);      // qrafika üçün pəncərənin açılması
  x = 1;
  y = 5;
  sprintf ( s, "X=%d, Y=%d", x, y);       //  s sətrinə xaricetmə
  outtextxy (100, 100, s);               // s sətrinin ekrana çıxarılması
  getch();
  closegraph ();
}
```

Sətirdən daxiletmə

Bəzən fayllardan məlumatları oxuyarkən tərs məsələyə rast gəlmək olur: verilənləri özündə saxlayan simvol sətri var. Bu verilənləri yaddaşın müəyyən xanalarına daxil etmək lazımdır.

Bunun üçün **sscanf** funksiyasından istifadə olunur. Verilmiş formata əsasən bu funksiya verilənləri klaviaturadan (bunun üçün **scanf** funksiyasından istifadə olunur) və ya fayldan (bunun üçün **fscanf** funksiyasından istifadə olunur) yox, simvol sətrindən oxuyur.

Aşağıda göstərilən misalda fayldan **'#'** işarəsi ilə başlayan sətir müəyyən edilir, bu sətirdən **x** və **y** qiymətləri oxunur.

Məsələnin çətinliyi ondan ibarətdir ki, faylda sətrin sayca neçənci olduğu bilinmir. Əgər sscanf funksiyasından istifadə edilməzsə, onda əvvəlcə faylda sətrin nömrəsini müəyyən etmək lazımdır, sonra isə faylın əvvəlindən başlayaraq lazımlı sayda sətirləri keçərək **fscanf** funksiya vasitəsi ilə verilənləri oxumaq.

```
#include <stdio.h>
main ()
{
   char s[80];         //  əlavə sətir
   int x, y;
   FILE *fp;
   fp = fopen ( "input.dat", "r" );
   while ( fgets (s, 80, fp) )
        if ( s[0] == '#' ) {              // əgər sətir '#' simvolu ilə başlayırsa, ...
             sscanf (s+1, "%d%d", &x, &y );   // verilənləri oxuyuruq
             break;                           //  dövrdən çıxış
             }
  fclose ( fp );
  printf ( "x = %d, y = %d", x, y );
}
```

Funksiya və proseduralarda sətirlər

Standart və istifadəçi tərəfindən yazılmış funksiyalar sətirləri parametr kimi qəbul edə bilərlər. Lakin burada bəzi məqamlar var.

Bildiyiniz kimi, əgər funksiyanın parametri **int a** kimi elan olunursa, onda funksiyada **a**-nın dəyişməsi onun əsas proqramda olan qiymətə heç bir təsir göstərmir (funksiyalar ədədi dəyişənlərin surətləri ilə işləyirlər). Funksiya parametrinin dəyişməsi üçün onu **int &a** kimi elan etmək lazımdır. Bu qayda bütün sadə tiplərə aiddir.

Funksiyaya və ya proseduraya sətri ötürəndə, **sətrin başlanğıc ünvanı** ötürülür, sətrin surəti yaradılmır. Ona görə də, yadda saxlamaq lazımdır ki,

> Funksiya və ya prosedurada sətir-parametri dəyişdikdə əsas proqramda ona uyğun sətir də dəyişəcəkdir.

Funksiyalarda sətirləri necə elan etmək olar?

1. massiv kimi:

char s [ ]

2. göstərici kimi (onsuzda funksiyaya sətrin ünvanı göndərilir):

char *s

Bu iki üsulun fərqi var: birinci halda **s** – simvollar massivinin adıdır, ona görə də onu dəyişmək olmaz. İkinci halda **s** – simvol göstəricisidir və onu sürüşdürmək olar.

C dilinin bütün standart funksiyaları sətirləri göstərici kimi elan edirlər. Aşağıda göstərilmiş funksiyalarda iki fərqli üsuldan istifadə edərək eyni əməliyyat yerinə yetirilir: **s2** sətri **s1**-ə köçürülür.

```
void copy1 (char s1 [ ], char s2 [ ] )
{
  int i = 0;
  while ( s2[i] )
     { s1[i] = s2[i];
       i++;
     }
   s1[i] = '\0';
}
```

```
void copy2 (char *s1, char *s2 )
{
 while ( *s1++ = *s2++ );
}
```

1. *s2-nin qiymətini s1-ə yazmaq
2. s1 və s2-i artırmaq
3. hələ ki, *s2 sıfır deyil bu əməliyyatları yerinə yetirmək

İkinci funksiya birincidən yığcamdır. Proqram aydın olmasa da, göstəricilərin istifadə edilməsi əlavə dəyişənə ehtiyac saxlamadı. **while** dövründə mənimsətmə operatorundan istifadə edilir. **s2** ünvandan simvol götürülür və **s1** ünvanına yazılır. Bu əməliyyatdan sonra növbəti simvollara baxılır **(*s1++** və ***s2++**). Dövrün şərti nə vaxt yoxlanılır? Şərt **++** (inkrement) işarələrin yerindən asılıdır. Mizim misallarda inkrement işarəsi dəyişənlərdən **sonra** yerləşir, ona görə də inkrement yoxlamadan **sonra** baş verir. Yoxlama isə belə olur: növbəti simvol köçürüləndən sonra yoxlanılır. Əgər o, **'\0'**-dırsa, onda dövr öz işini bitirir, əks halda **s1** və **s2** göstəricisi artırılır və dövr davam edir. Qeyd edək ki, dövrdən çıxandan sonra göstəricilər yenə də artırılır və bu göstəricilər yaddaşın növbəti xanalarına göstəriş verəcəklər.

14. MATRİSLƏR (İKİ ÖLÇÜLÜ MASSİVLƏR)

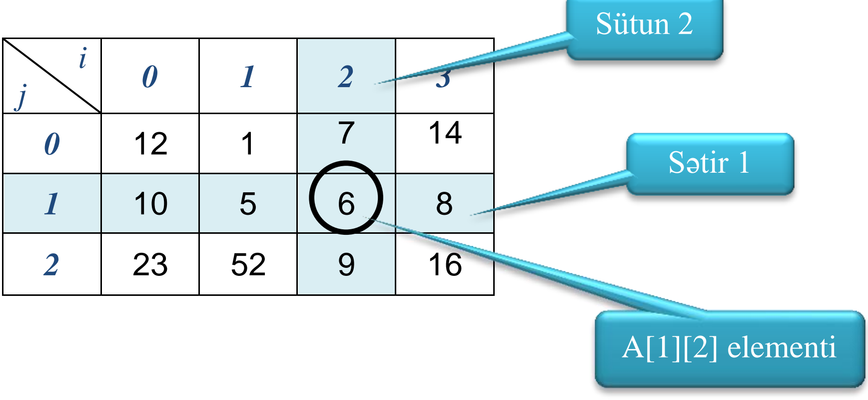
Matrisa nədir?

Çox vaxt xətti struktura görə massivlərdən istifadə etmək narahatdır. Məsələn, tutaq ki, bir neçə il ərzində yağıntılar haqqında məlumatları emal etmək lazımdır, özü də hər ay üçün yağıntıların miqdarı məlumdur. Əgər məlumatları əl ilə emal etsək, onda ən rahat üsul cədvəldir. Cədvəldə üfüqi xətt üzrə illəri, şaquli xətt üzrə isə ayları qeyd etmək olar. Əslində, proqramda birölçülü massivdən də istifadə etmək olardı, lakin bu halda hər dəfə indekslər sürüşdürməlidirlər.

Cədvəldə yerləşən məlumatların emalı üçün bütün müasir proqramlaşdırma dillərində iki *ölçülü massivlərdən* və ya *matrisalardan* istifadə edilir.

Düzbucaqlı elementlər (məsələn, ədədlər və ya simvollar) cədvəlinə **matrisa** deyilir. Matrisa iki ölçülü massivdir. Bu massivin iki indeksı var.

Matrisa sətirlərdən və sütunlardan ibarətdir. Hər elementin iki indeksi var: elementin yerləşdiyi sətir və sütun.



j \ i	0	1	2	3
0	12	1	7	14
1	10	5	6	8
2	23	52	9	16

C dilində hər indeks kvadrat mötərizələr içində ayrıca yazılır. Matrisanın hər sütununa və hər sətrinə adi birölçülü massiv kimi baxmaq olar. Ona görə də, matrisa – *massivlərin massividir*. Matrisalara iki məhdudiyyət qoyulur:

1. Matrisanın bütün elementləri **eyni tipə** malik olmalıdırlar;
2. Matrisanın bütün sətirləri **eyni uzunluğa** malik olmalıdırlar.

Matrisanın birinci indeksi **sətirdir**, ikinci indeksi isə **sütundur**. Sətirlərin və sütunların sayı bərabər olan matrisalar **kvadrat matrisalar** adlanır. Kvadrat matrisalarda **əsas diaqonalı** seçmək olar. Əsas diaqonalda yerləşən elementlərin sətrinin nömrəsi sütunun nömrəsinə bərabərdir, yəni $N \times N$ ölçülü matrisa üçün:

A[0][0], A[1][1], A[2][2],…, A[N-1][N-1]

Matrisaların elanı

Matrisaları adi massivlər kimi elan edirlər. Massivlərdən fərqli olaraq matrisaların iki indeksi var. Elan zamanı kvadrat mötərizələr içində matrisanın sətirlərin sayı və sütunların sayı göstərilməlidir. Məsələn, aşağıdakı operator yaddaşda 200 elementli (20 sətir, 10 sütun) tam ədədlərin matrisası üçün yer ayıracaqdır:

```
int BM[20][10];
```

Əgər matrisa qlobaldırsa (bütün prosedura və funksiyalardan əvvəl elan olunursa), onda o, əvvəldən sıfırlarla tamamlanır. Lokal matrisalar (prosedura və funksiyalarda elan olunmuş) məlum olmayan qiymətləri özündə saxlayırlar.

Elementlərin başlanğıc qiymətləri

Elan zamanı matrisanın bütün və ya bəzi elementlərinin qiymətlərini vermək olar. Məsələn,

```
float X[2][3]  = { {1., 2., 3.}, {4., 5., 6. } };
```

Misaldan göründüyü kimi hər sətrin elementləri fiqurlu mötərizələr içində yazılır. Əgər bütün elementlərin qiymətləri verilməsə, onda qalanları sıfırlarla tamamlanacaqdır:

```
float X[2][3] = { {1., 3.}, {6.} };
```

Burada X[0][2], X[1][1], X[1][2] elementləri sıfır qiyməti alacaqlar.

Matrisaların yaddaşda yerləşməsi

Bütün müasir proqramlaşdırma dillərində (Fortran-dan başqa) matrisanın elementləri yaddaşda *sətir-sətir* yerləşirlər, yəni axırıncı indeks tez dəyişəndir. Yuxarıda elan olunmuş **X** matrisası yaddaşda belə yerləşir:

X[0][0]	X[0][1]	X[0][2]	X[1][0]	X[1][1]	X[1][2]

Standart daxiletmə və xaricetmə

Birölçülü massivlərdəki kimi matrisaları klaviaturadan, fayldan oxumaq və təsadüfi ədədlər vasitəsi ilə doldurmaq olar. Ümumi prinsip ondan ibarətdir ki, matrisanın hər elementi ayrıca oxunmalıdır. Təsəvvür edin ki, matrisa eyni uzunluqlu sətirlərin massividir. Bir sətri daxil etmək üçün bir dövr lazımdır, bir-neçə sətri daxil etmək üçün 2 dövr lazım olacaqdır. Ona görə, matrisalarla işləmək üçün *ikiqat (bir-birinin içində yerləşdirilmiş*) dövrlərdən istifadə olunur.

Klaviaturadan daxiletmə

```
#include <stdio.h>
```

```
const int M = 5;     //   sətirlərin sayı
const int N = 4;    //    sütunların sayı
main ()
{
  int i, j, A[M][N];
  for ( i=0; i<M; i++ )               // sətirlərə görə dövr
    for ( j=0; j<N; j++ )             //  sütunlara görə dövr
      {
       printf ( "A[%d][%d] = ", i, j );      // daxiletmə üçün göstəriş
       scanf ( "%d", &A[i][j] );             //  A[i][j] elementin daxil edilməsi
      }
//
}
```

Qeyd edək ki, dövrlərin ardıcıllığını dəyişdirsək, onda matrisanın elementləri yaddaşa sütun-sütun oxunacaqlar.

Təsadüfi ədədlərlə tamamlama

Birölçülü massivlərlə analoji olaraq ikiqat dövrdə həyata keçirilir. Aşağıdakı misalda tam tipli matrisa **[a, b]** intervalında dəyişən təsadüfi ədədlərlə tamamlanır (həqiqi ədədlər üçün düstur dəyişir). Bu funksiya isə **[0, N-1]** intervalından təsadüfi tam ədədi generasiya edir. Onu proqrama əlavə etmək lazımdır.

```
int random ( int N ) { return rand()%N; }
```

M sətirli və **N** sütunlu tam tipli matrisanı elementlərlə tamamlamaq üçün **random** funksiyasından istifadə etmək lazımdır.

```
for ( i=0; i<M; i++ )
 for ( j=0; j<N; j++ )
   A[i][j] = random (b-a+1) + a;
```

Ekrana xaricetmə

Matrisanı ekranda təsvir etdikdə onun elementlərini sətirlərlə yerləşdirmək məqsədə uyğundur, yəni matrisanın bir sətrini çapa verib növbəti sətrə keçmək lazımdır və s. Nəzərə almaq lazımdır ki, matrisanın ekranda səliqəli yerləşdirilməsi üçün hər elementə eyni ölçüdə yer ayrılmalıdır. Bunu etmək üçün format sətrində faiz işarəsindən sonra hər simvola ayrılan pozisiyaların sayını göstərmək lazımdır.

```
printf ( "A matrisasi\n" );
for ( i=0; i<M; i++ )  {            // sətirlərə görə dövr
  for ( j=0; j<N; j++ )             // dövrdə sətrin çapı
    printf ( "%4d", A[i][j] );       // hər ədədə 4 simvol yer ayrılır
  printf ( "\n" );                   // növbəti sətrə keçmə
}
```

Fayllarla işləmə

Mətn faylları

Mətn fayllarla iş zamanı matrisa elementləri fayldan ardıcıl oxunmalıdır. Faylın oxunmasında yaranan səhvləri (elementlərin azlığı və ya faylın boş olması) proqramda emal etmək lazımdır.

```
#include <stdio.h>
const int M = 5;        // sətirlərin sayı
const int N = 4;         // sütunların sayı
main ()
{
   int i, j, A[M][N];
   FILE *fp;
   fp  = fopen( "input.dat", "r" );
   for ( i=0; i<M; i++ )              // sətirlərə görə dövr
     for ( j=0; j<N; j++ )            // sütunlara görə dövr
        if ( 0 == fscanf (fp, "%d", &A[i][j] ) )  // A[i][j] elementinin daxil edilməsi
          {
            puts ( "Verilenler chatishmir!" );
            fclose ( fp );                          // səhvə görə faylın bağlanması
            return 1;                               // proqramdan çıxış
            }
  fclose ( fp );                          // normal vəziyyətdə faylın bağlanması
// matrisa ilə iş
}
```

Matrisanı fayla yazmaq üçün analoji əməliyyatlardan istifadə olunur. Əvvəlcə **fopen** funksiyası vasitəsi ilə faylı açmaq lazımdır (burada yazma rejimindən istifadə olunmalıdır **"w"**). Sonra isə, ikiqat dövrdə **fprintf** funksiyasından istifadə edərək matrisanı fayla ötürmək və axırda **fclose-**la faylı bağlamaq lazımdır.

Binar fayllar

Əgər verilənlər başqa proqram vasitəsi ilə fayla yazılırsa, onda bu verilənləri binar faylı vasitəsi ilə oxumağa rahat olacaqdır. Binar faylların əsas üstünlüyü ondan ibarətdir ki, onlardan oxunma və onlara yazma sürətlə baş verir, çünki bütöv massiv birdəfəlik oxunur və ya yazılır. Bu zaman **fread** və **fwrite** funksiyalarda bir elementin uzunluğunu və elementlərin ümumi sayını, yəni $M \times N$ göstərmək lazımdır.

Aşağıdakı proqramda matrisa binar fayldan oxunur, onun üzərində müəyyən əməliyyatlar aparılır və sonra o, binar fayla yazılır.

```
#include <stdio.h>
const int M = 5;      //  sətirlərin sayı
const int N = 4;      //  sütunların sayı
```

```
main ()
{
   int total, A[M][N];
   FILE *fp;

   fp = fopen ("input.dat", "rb" );
   total = fread (A, sizeof (int), M*N, fp );        // matrisanın oxunması
   fclose(fp);
   if ( total !=M*N )      // əgər bütün elementlər oxunmayıbsa,
     {
      printf ( "Verilenlerin sayi azdir");
      return 1;
      }
  //
  fp = fopen( "output.dat", "wb" );            // matrisanın fayla yazılması
  if ( M*N  != fwrite(A, sizeof(int), M*N, fp) )
     printf ( "Faylin yazilmasinda sehv" );
  fclose ( fp);
}
```

Səhvlərin emalı üçün o faktdan istifadə olunur ki, **fread** və **fwrite** funksiyaları, nəticə olaraq, oxunmuş və ya yazılmış elementlərin sayını göndərirlər və əgər o say elementlərin ümumi sayına bərabər deyilsə, onda ekrana səhv haqqında məlumat çıxarılır.

Matrisalarla işləmək üçün alqoritmlər

Matrisanın minimal elementinin təyini

Birölçülü massivlərdən fərqli olaraq matrisanın bütün elementlərini nəzərdən keçirmək üçün ikiqat dövrdən istifadə olunmalıdır. Matrisanın minimal elementini təyin etmək üçün əvvəlcə fərz edirik ki, bu element **A[0][0]**-dır. Sonra bütün elementləri yoxlayaraq bu elementdən kiçiyini təyin edirik. Birölçülü massivdəki kimi elementi yox, onun indekslərini yadda saxlamaq məqsədə uyğundur. İndeksləri biləndən sonra elementin qiymətini asanlıqla təyin etmək olar.

```
#include <stdio.h>
const int M = 5;       //  sətirlərin sayı
const int N = 4;       //   sütunların sayı
main ()
{
float A[M][N], i, j, row, col;
// matrisanın daxil edilməsi
row = col = 0;                    //  fərz edirik ki, A[0][0] elementi minimaldır
for ( i=0; i<M; i++ )             //  bütün sətirlərin yoxlanılması
  for ( j=0; j<N; j++ )           // bütün sütunların yoxlanılması
```

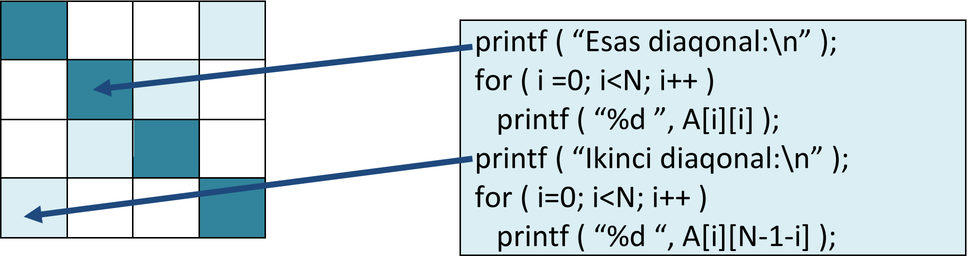
```
        if ( A[i][j] < A[row][col] )
          { row = i;                    //  yeni indekslərin yadda saxlanılması
            col = j;
          }
  printf ( “Matrisanin minimal elementi A[%d][%d] = %d”, row, col, A[row][col] );
}
```

Ayrı-ayrı elementlərlə iş

$N \times N$ ölçülü kvadrat matrisanı nəzərdən keçirək. Matrisanın əsas və ona perpendikulyar diaqonalda yerləşən elementləri ekranda əks etdirək. Əsas diaqonalda yerləşən elementlərin indeksləri eyni olur, yəni ***A[i][i], i=0,1, …, N-1***. İkinci diaqonalda yerləşən elementlərin indeksləri aşağıdakı kimi dəyişir:

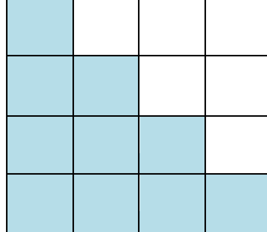
A[0][N-1], A[1][N-2], A[2][N-3], …, A[N-1][0]

Gördüyünüz kimi, sətir və sütun indekslərinin cəmi **N-1**-ə bərabərdir, yəni ikinci diaqonalda **A[i][N-1-i]** elementlər dayanır.



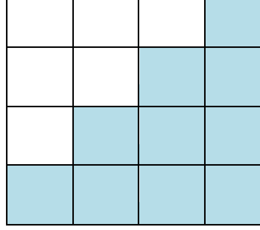
```
printf ( “Esas diaqonal:\n” );
for ( i =0; i<N; i++ )
  printf ( “%d ”, A[i][i] );
printf ( “Ikinci diaqonal:\n” );
for ( i=0; i<N; i++ )
  printf ( “%d “, A[i][N-1-i] );
```

İndi isə daha mürəkkəb məsələyə baxaq: əsas diaqonaldan aşağı yerləşən elementlərin qiymətlərini sıfırlaşdıraq. Birinci sətirdə bu **A[0][0]**, ikinci sətirdə **A[1][0], A[1][1]**, üçüncü sətrə **A[2][0], A[2][1], A[2][2]** və s. Qeyd edək ki, **i** nömrəli sətirdə sütunların indeksləri **0**-dan **i**-ə kimi dəyişir.



```
for ( i =0; i<N; i++ )
  for ( j=0; j<=i; j++ )
     A[i][j]  = 0 ;
```

İndi isə başqa məsələnin həllinə baxaq. İkinci diaqonaldan aşağı yerləşən elementləri (diaqonal daxil olmaqla) sıfırlaşdırmaq lazımdır. Birinci sətirdə **A[0][N-1]** elementi, ikinci sətirdə **A[1][N-2], A[1][N-1]** elementləri, üçüncü sətirdə **A[2][N-3], A[2][N-2], A[2][N-1]** elementləri və s. sıfırlaşdırmaq lazımdır. **i** sətrində yerləşən elementlərin sütun indeksləri **N-1-i**-dən **N-1**-ə kimi dəyişir.



```
for ( i =0; i<N; i++ )
  for ( j=N-1-i; j<=N-1; j++ )
     A[i][j]  = 0 ;
```

Sətirlərin və sütunların yerdəyişməsi

Tutaq ki, **i1** və **i2** nömrəli sətirlərin yerlərini dəyişmək lazımdır. Bu o deməkdir ki, müvəqqəti **temp** dəyişənindən istifadə etməklə hər **j** sütununda **A[i1][j]** və **A[i2][j]** elementlər öz yerlərini dəyişməlidirlər.

```
for ( j=0; j<N; j++ )
  {
  temp = A[i1][j];
  A[i1][j] = A[i2][j];
  A[i2][j] = temp;
  }
```

İkiölçülü matrisanın birölçülü massivə çevrilməsi

Bəzən ölçüləri **M** və **N** olan **A** matrisasını ölçüsü **M*N** olan birölçülü **B** massivinə çevirmək lazım gəlir. Aydındır ki, sətirlərlə oxuyanda əvvəlcə birinci sətrin **A[0][j]** elementləri **B[j]**, sonra ikinci sətrin **A[1][j]** elementləri **B[N+j]** elementlərinə köçürüləcəkdir və s. Beləliklə, **i** nömrəli sətrin elementləri **B[i*N+j]** elementlərinə köçürüləcəkdir.

```
for ( i=0; i<M; i++)
  for (j=0; j<N; j++)
    B[i*N+j] = A[i][j];
```

Əgər yanaşı yerləşən matrisa elementləri ilə müəyyən əməliyyatları yerinə yetirmək lazım gələrsə, onda birölçülü massivlər üçün yaratdığımız prosedurları matrisa üçün də istifadə etmək olar. Qeyd edək ki, matrisanın hər hansı sətrinin adı o sətirdə yerləşən birinci elementin adı ilə üst-üstə düşür. Məsələn, A[0] 0-ci sətrinin əvvəlinə göstəricidir. Yaddaşda matrisa elementləri sətir-sətir yerləşirlər. Məsələn, elementlərin cəmini hesablayan funksiya matrisa üçün aşağıdakı kimi istifadə oluna bilər:

```
s0 = Sum(A[0], N);              // sıfır indeksli sətrin cəmi
s14 = Sum(A[1], 4*N );           // 1-dən 4-ə qədər sətirlərin cəmi
sAll = Sum(A[0], M*N);           // bütün sətirlərin cəmi
```

15. SİMVOL SƏTİRLƏRİNİN MASSİVLƏRİ

Əgər sətir – simvollar massividir, onda sətirlərin massivi – massivlərin massividir və ya ikiölçülü simvol massividir (matrisa). Matrisalarda olduğu kimi, sətirlərin uzunluğu eyni olmalıdır. Ədədi matrisalardan fərqli olaraq simvol matrisada sətrə simvollar yığımı kimi yox, bütöv simvol sətri kimi baxılır.

İkiölçülü simvol massivlərinin elanı və inisializasiyası

Aşağıdakı fraqmentdə 4 sətirdən ibarət simvol massivinin elanı göstərilmişdir. Hər sətirdə **'\0'** simvolu daxil olmaqla 80 simvol ola bilər.

```
char s[4][80];
```

Bu elanı başqa cürə etmək olardı. Əvvəlcə verilənlərin yeni tipi - **str80** (80 simvollu sətir) təyin olunur. Sonra isə lazımlı ölçüdə massiv elan olunur.

```
typedef str80[80];    // verilənlərin yeni tipi – 80 simvollu sətir
.....
str80 s[4];           // 4 sətirli massivin elanı
```

typedef direktivi yeni tipin birinci istifadəsindən qabaq yazılmalıdır. Adətən, onu qlobal dəyişənlərlə bir yerdə bütün prosedura və funksiyalardan əvvəl yazırlar. Belə elandan sonra hər sətri ayrıca istifadə etmək olar. Bu cürə təsvir olunmuş ikiölçülü massivin istənilən simvoluna müraciət etmək olar. Məsələn, **s[2][15]** - 3-cü sətirdə yerləşən 16-cı simvoldur.

Bütün və ya bir-neçə sətrə başlanğıc qiymət vermək olar. Əgər başlanğıc qiymətlərin sayı massivdə olan sətirlərin sayından azdırsa, onda qalan sətirlər sıfırlarla tamamlanır.

```
char s[4][80] = { "Fizika", "Kimya" };
```

Daxiletmə və xaricetmə

Hər sətir üçün daxiletmə və xaricetmə ayrıca həyata keçirilir, yəni sətirlər massivinin daxil və xaric edilməsi üçün dövrdən istifadə etmək lazımdır. Aşağıdakı fraqmentdə 20-dən çox olmayan sətirləri daxil etmək olar (dövrdən çıxış - boş sətirdir). **count** adlı dəyişən daxil edilmiş sətirlərin sayını özündə saxlayacaqdır.

```
chat s[20][80];
int i, count = 0;
printf ( "Metni daxil edin (chixmaq uchun  - Enter)\n" );
for ( i=0; i<20; i++ )          // 20 sətri oxumaq üçün
  {  gets ( s[count] );           // sətri klaviaturadan oxuyuruq
  if ( s[count][0] =='\0' )   // əgər boş sətir daxil olunubsa,
       break;                  //  dövrdən çıxırıq
   count ++;
   }
```

Sətirlər massivinin ekrana çıxarılması və ya mətn faylına yazılması və oxunması analoji olaraq dövrdə aparılır.

Çeşidləmə

Simvol tipli massivlərinin sətirlərin çeşidlənməsi əlifba sırası ilə aparılır. Çeşidləmə zamanı iki problemə rast gəlmək olar:

1. Verilmiş iki sətirdən kiçiyini, yəni öndə dayandığını təyin etmək lazımdır;
2. Ədədi massivlərin çeşidlənməsində elementlərin yerdəyişməsi baş verirdi. Simvol tipli massivlərdə bu əməliyyat böyük həcmli verilənlərin köçürülməsi ilə bağlı olduğundan faydalı deyil.

Sətirlərin müqayisəsi üçün **strcmp** funksiyasından istifadə etmək olar. Bu funksiya nəticə olaraq iki sətrin fərqini, yəni birinci fərqli simvolların kodlarının fərqini qaytarır. Əgər sətirlər bərabərdirsə, onda nəticə sıfırdır. Əgər nəticə mənfidirsə, onda birinci sətir ikincidən kiçikdir və ondan əvvəl dayanmalıdır. Əgər nəticə müsbətdirsə, onda birinci sətir ikincidən sonra dayanmalıdır. Qeyd edək ki, müqayisə kodlar cədvəli əsasında aparılır və bu cədvəldə baş həriflərin kodları kiçik hər iflərin kodlarından kiçikdir.

İkinci problemi həll etmək üçün *göstəricilərdən* ibarət massivdən istifadə etmək lazımdır. Proqramda əvvəlcə sətir tipli göstəricilər massivi elan olunmalıdır, özü də **i** nömrəli göstərici massivin **i** nömrəli sətrinə göstərir. Hava qabarcığı üsulu vasitəsi ilə sətirlərin çeşidlənməsi aşağıdakı kimi olacaqdır:

```
void SortStrins ( char *s[ ], int n )     // *s[ ] – göstəricilər massivi, n –sətirlərin sayı
{  char *p;
  int i, j;
  for ( i=0; i<n-1; i++ )
    for ( j=n-1; j>i; j-- )
      if ( strcmp( s[j-1], s[j] ) > 0 )
        { p = s[j]; s[j] = s[j-1];  s[j-1]  =p;   }  // göstəricilərin yerdəyişməsi
}
```

Bu funksiyadan istifadə edərək sətriləri əlifba sırasına düzüb ekrana çıxardan proqram aşağıda göstərilmişdir:

```
main()
{
   char s[20][80];              // 20 sətirli simvol massivi
   char *ps[20];                // sətir tipli 20 göstəricidən ibarət massiv
   int i, count;
   // burada sətirləri daxil edib
   // onların sayını count dəyişəninə yazmaq lazımdır
  for ( i=0; i<count; i++ )    // göstəriciləri qoymaq
      ps[i]=s[i];
   SortStrings( ps, count);            // göstəricilərin çeşidlənməsi
   for ( i=0; i<count; i++ )
     puts ( ps[i] );                   // göstəricilər vasitəsi ilə xaricetmə
}
```

16. YADDAŞIN İDARƏEDİLMƏSİ

Göstəricilər

Belə bir məsələyə baxaq: fayla tam ədədlər yazılıb. Onları çeşidləyib başqa fayla yazmaq lazımdır. Problem ondan ibarətdir ki, fayldakı ədədlərin sayı qabaqcadan məlum deyil. Bu məsələni həll etmək üçün iki variant var:

1. Əgər qabaqcadan bilinir ki, ədədlərin sayı 1000-dən çox deyilsə, onda ehtiyat üçün artıqlaması ilə yaddaş ayırmaq;
2. Yaddaşı dinamik ayırmaq. Əvvəlcə massivdə olan ədədlərin sayını təyin etmək, sonra isə lazımı yaddaşı ayırmaq.

İkinci variant daha yaxşıdır. Dinamik yaddaşla işləmək üçün göstəricilərlə daha ətraflı tanış olaq.

> Digər dəyişənin və ya yaddaş sahəsinin ünvanını özündə saxlayan dəyişən **göstərici** adlanır.

Göstəricilər C dilinin əsas anlayışlardan biridir. Bu cürə dəyişənlərə yaddaşın istənilən sahələrin ünvanlarını, ən çox da dinamik massivin başlanğıc elementinin ünvanını yazmaq olar. Göstəricilərlə hansı əməliyyatları yerinə yetirmək olar?

Elan etmək	char *pC; int *pI; float *pF;	Göstəricilər dəyişənlər bölməsində elan olunur, onların adlarının qabağında ***** işarəsi qoyulur. Göstərici hansı tip dəyişən üçün elan olunmuşdur, ona da göstərir. Məsələn, **pC** istənilən simvola, **pİ** – tam ədədə, **pF** – həqiqi ədədə göstərə bilər.
Ünvanı mənimsətmək	int i, *pI; ... pI = &i;	Bu yazı "**pİ** göstəriciyə **i** dəyişənin ünvanını yazmaq" deməkdir. **&i** yazısı **i** dəyişənin ünvanı deməkdir.
Bu ünvandan qiyməti almaq	float f, *pF; ... f = *pF;	Bu yazı "**f** dəyişənə **pF** göstəricisi göstərən həqiqi ədədi yazmaq" deməkdir. ***pF** yazısı **pF** göstərən xananın içi (tərkibi) deməkdir.

Sürüşdürmək	int *pI;
	...
	pI ++;
	pI += 4;
	--pI;
	pI -= 4;
		Bu əməliyyatlar nəticəsində göstəricinin qiyməti dəyişir: **pİ++** yazısı göstəricini TAM ƏDƏDIN ÖLÇÜSÜ qədər, yəni **4** bayta sürüşdürür. **pİ +=4** və ya **pİ=pİ+4** yazısı göstəricini 4 tam ədədə, yəni 16 bayta sürüşdürür.

Sıfırlaşdırmaq	char *pC; ... pC = NULL;	Əgər göstərici sıfır ünvana - **NULL** bərabərdirsə, onda bu göstərici etibarsızdır. Bu göstəriciyə heç nə yazmaq olmaz (proqram səhvə çıxa bilər).
Ekrana çıxartmaq	int i, *pI; ... pI = &i; printf ("i-nin ünvanı = %p", pI);	Göstəriçilərin çapı üçün **%p** formatından istifadə edirlər.

Qaydalar:

- Digər dəyişənin ünvanını özündə saxlayan dəyişənə göstərici deyilir;
- Göstəricinin elanı zamanı onun tipini göstərmək və adının qabağına * işarəsi qoymaq vacibdir;
- dəyişənin adının qabağında & işarəsi onun ünvanı deməkdir;
- Proqramın işçi sahəsində (elan sahəsində yox) göstəricinin qabağında qoyulmuş * işarəsi göstərici göstərən xananın qiymətidir;
- Heç bir dəyişənlə bağlı olmayan göstəriciyə qiymət yazmaq olmaz. Bunun nəticəsində naməlum xanalarda məlumatlar silinir və bu proqramın nasazlığına gətirib çıxardır;
- Etibarsız göstəricini qeyd eləmək üçün **NULL** sabitindən istifadə olunur;
- Göstəricinin qiymətini n elədikdə, o, verilmiş tipin növbəti **n** saylı ədədinə sürüşdürülür, məsələn, tam tipli göstəricilər üçün **n*sizeof(int)** bayta sürüşdürülür;
- göstəriciləri **%p** formatı ilə çap edirlər.

scanf və **fscanf** funksiyaları parametr kimi dəyişənlərin ünvanlarından istifadə edirlər:

```
scanf ( “%d”, &i );
```

Yaddaşın dinamik ayrılması

Proqramın yazılması zamanı ölçüsü bilinməyən massivlər **dinamik massivlər** adlanır. Aşağıdakı proqramda istifadə olunan üsul C++ dilinə aiddir. Bu proqram dinamik massivdən istifadə edir. Massivin ölçüsü və onun elementləri klaviaturadan daxil olunur, massiv çeşidlənir və ekrana çıxarılır.

```
#include <stdio.h>
main ()
{
int N;              // massivin ölçüsü (qabaqcadan məlum deyil)
int *A;             // yaddaşı ayırmaq üçün göstərici
printf ( “Massivin olchusu =” );   // massivin ölçüsünün daxil edilməsi
scanf ( “%d”, &N );
A = new int [N];            // yaddaşın ayrılması
if ( A == NULL )  {         // əgər yaddaş ayrılmamışdırsa,
   printf ( “Yaddashi ayrilmadi” );
   return 1;                  // səhvə görə çıxış
   }
for ( i=0; i<N; i++ ) {     //  adi massivlərdəki kimi
    printf ( “\nA[%d] = “, i+1 );
    scanf ( “%d”, &A[i] );
    }
// burada massivin çeşidlənməsi və xaric edilməsi
delete A;     // yaddaşın boşaldılması
}
```

Beləliklə, proqramın icrası zamanı tam tipli massivə yer ayırmaq üçün əvvəlcə onun ölçüsü daxil olmalıdır (tutaq ki, bu **N**-dir). Yaddaşın ayrılması üçün **new** operatorundan istifadə edilir. Bu operator nəticə olaraq yeni ayrılmış yaddaş blokunun ünvanını qaytarır. Bu ünvanın qiymətini xüsusi dəyişəndə, yəni göstəricidə yadda saxlanılmalıdır.

Qaydalar:

- əgər proqram yazılan zaman massivin ölçüsü məlum deyilsə, onda dinamik massivlərdən istifadə olunur;
- dinamik massivlərlə işləmək üçün uyğun tipli göstərici elan olunmalıdır (bu göstəricidə massivin birinci elementinin ünvanı saxlanacaqdır);

```
int *A;
```

- massivin ölçüsünü biləndən sonra mötərizədə massivin ölçüsünü göstərməklə **C++** dilinin **new** operatorundan istifadə etmək lazımdır.

```
A = new int [N];
```

- mənfi və ya sıfır qiymətli **N** üçün **new** operatorundan istifadə etmək **olmaz**;
- yaddaşı ayırandan sonra bu əməliyyatın müvəffəqiyyətlə həyata keçməsini yoxlamaq lazımdır. Əgər yaddaş ayrılmayıbsa, onda göstəricinin qiyməti **NULL** olacaqdır və proqram səhvlə nəticələnə bilər;
- A adlı göstəricinin göstərdiyi dinamik massivlə iş adi massivlə görünən işə oxşayır.
- Massivi istifadə edəndən sonra ona ayrılmış yaddaşı boşaltmaq lazımdır:

```
delete A;
```

- yaddaşı boşaldandan sonra göstəricinin qiyməti dəyişmir, lakin yaddaş boş olduğundan ondan istifadə etmək olmaz;
- nəzərə alaraq ki, göstəricinin üzərinə ədəd gəldikdə o, ədədin tipindən asılı olaraq uyğun xanaya sürüşdürülür, onda aşağıdakı yazılar eynidir və **i** nömrəli massiv elementinin ünvanını hesablayırlar.

```
&A[i] ⇔ A+i
```

Burdan görünür ki, **A** massivin **&A[0]** birinci elementinin ünvanı ilə üst-üstə düşür, yəni massivin adını onun birinci elementinin ünvanı kimi istifadə etmək olar.

Yaddaşın ayrılmasında yaranan səhvlər

C dilində proqramlarda ən çətin təyin olunan səhvlər dinamik massivlərin istifadəsi ilə bağlıdır. Aşağıdakı cədvəldə səhvlər və onları aradan götürmək üçün üsullar göstərilmişdir:

Səhv	Səbəb və aradan götürmə üsulu
Başqasının yaddaş sahəsinə yazma	Yaddaş düz ayrılmayıb, lakin massiv istifadə olunur. **Nəticə**: göstəricini həmişə **NULL** qiymətinə görə yoxlamaq lazımdır.
Göstəricinin təkrar	Massiv silinib, lakin yenə də təkrar silinir.

silinməsi	**Nəticə**: əgər massiv silinirsə, göstəricini sıfırlaşdırmaq lazımdır. Onda səhv daha tez aşkar olunur.
Massiv sərhədlərinin aşması	Massivə mənfi indeksli və ya massivin sərhədlərini aşan element yazılır.
Yaddaşın aşıb daşması	İstifadə olunmayan yaddaş özü-özünə boşalmır. Əgər yaddaş funksiyada ayrılırsa və bu funksiya bir neçə dəfə çağırılırsa, onda yaddaş tez dolacaqdır. **Nəticə**: yaddaşı ara bir boşaltmaq lazımdır.

Matrisa üçün yaddaşın ayrılması

Birölçülü tam tipli massivə yaddaşda yer ayırmaq üçün tam tipli göstəricidən istifadə olunmalıdır. Matrisa üçün tam tipli massiv göstəricini təyin etmək lazımdır. Onun elanı aşağıda göstərilmişdir:

```
int **A;
```

Lakin, ən yaxşı üsul verilənlərin yeni tipini – tam tipli göstəricini təyin etmək. Yeni tiplər bütün proseduralardan və funksiyalardan kənar (qlobal dəyişənlər təyin olduğu yerdə) **typedef** direktivi vasitəsi ilə təyin olunurlar.

```
typedef int *pInt;
```

Bu sətirdən məlum olur ki, **pİnt** tam ədədə və ya tam tipli massivə göstərən göstəricidir. Təəssüf ki, iki ölçülü massiv üçün göstərici vasitəsi ilə yer ayırmaq olmur. Kompilyator səhv haqqında məlumat verir.

```
int M = 5, N = 7;
pInt *A;
A = new int[M] [N];    // sətir səhvdir
```

Bu onunla bağlıdır ki, **A[i][j]** yazını oxumaq üçün kompilyator bir sətrin uzunluğunu bilməlidir. Bu məsələnin həlli üçün istifadə olunan 3 üsul aşağıda göstərilmişdir.

Ölçüsü məlum olan sətir

Əgər matrisa sətrinin ölçüsü məlumdursa, lakin sətirlərin sayı məlum deyilsə, onda “matrisa sətri” adlanan yeni tip təyin etmək olar. Sətirlərin sayı məlum olandan sonra **new** operatoru vasitəsi ilə bu cürə verilənlərin massivini təyin etmək olar.

```
typedef int row10[10];    // yeni tip: 10 elementli massiv
main ()
{
int N;
row10 *A;                 // massiv tipli göstərici (matrisa)
printf ( “Setrlerin sayini daxil edin:”);
scanf ( “%d”, &N );
```

```
A = new row10[N];                // N sətirli yer ayırmaq
A[0][1] = 25;                   // matrisadan adi istifadə
printf ( "%d", A[2][3] );
delete A;                       // yaddaşın boşaldılması
}
```

Ölçüsü məlum olmayan sətirlər

Tutaq ki, matrisanın ölçüləri, yəni **M** və **N** qabaqcadan məlum deyil və proqramın gedişi zamanı təyin olunurlar. Onda matrisa üçün yaddaşda yer ayırmaq üçün aşağıdakı üsuldan istifadə etmək olar. Belə ki matrisaya sətirlərin massivi kimi baxmaq olar, onda **M** sayda göstərici elan edək və hər göstərici üçün **N** uzunluqlu birölçülü massivin yerini ayıraq. Göstəricilər dinamik massiv şəkilində təsvir olunmalıdırlar. Matrisanın ölçülərini təyin etdikdən sonra dinamik göstəricilər massivinə yer ayırırıq, sonra isə hər göstəriciyə bir sətirlik yer veririk.

```
typedef int *pInt;    // yeni verilənlər tipi: tam tipli göstərici
main ()
{
  int M, N, I;
  pInt *A;            //  göstəriciyə göstərici
   //
A = new pInt[M];            // göstəricilər massivinə yer ayırmaq
for  ( i=0; i<M; i++)        // bütün göstəricilərə görə dövr
   A[i] = new int[N];       // i sətrinə yer ayırırıq
// Adi şəkildə A matrisa ilə iş
for ( i=0; i<M; i++)        // bütün sətirlər üçün yaddaşı boşaldırıq
  delete A[i];
delete A;                   // göstəricilər massivinə ayrılmış yeri boşaldırıq
}
```

Yuxarıda göstərilmiş misalda hər sətir üçün yaddaş ayrılır. Bunu başqa cürə etmək olar. Əvvəlcə bütöv matrisa üçün yer ayırıb onun ünvanını **A[0]**-a yazmaq. Sonra göstəriciləri elə düzmək lazımdır ki, **A[1]** - **N+1**-ci elementə, **A[2]** – **2N+1**-ci elementə və s. göstərərdi. Beləliklə, yaddaşda yalnız iki blok olur: göstəricilər massivi və matrisanın özü.

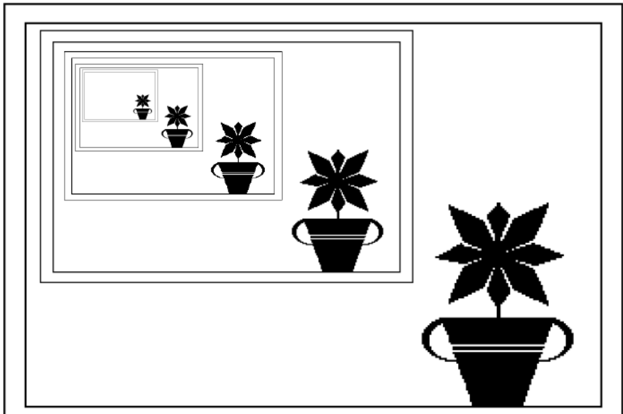
```
typedef int *pInt;
main ()
{   int M, N, I;
  pInt *A;             // göstəricilər göstəricisi
   // M və N-nin daxil edilməsi
A = new pInt[M];       // göstəricilər massivinə yer ayırmaq
A[0] = new int [M*N];  // matrisa üçün yaddaşın ayrılması
for (i=1; i<M; i++ )     // göstəricilərin qoyulması
   A[i] = A[i-1] + N;
// matrisa ilə iş
delete A[0];            // matrisaya ayrılmış yaddaşın boşaldılması
delete A;         // göstəricilərə ayrılmış yaddaşın boşaldılması
}
```

17. REKURSİYA

Rekursiya nədir?

Rekursiv obyektlər

Rekursiyanın bir nümunəsi – öz təsvirini özündə saxlayan şəkildir.



Bir obyektin özü özündən təyin edilməsi **rekursiya** adlanır.

Riyaziyyatda rekursiya vasitəsi ilə çoxlu sayda sonsuz çoxluqları, məsələn, natural ədədlərin çoxluğunu təyin edirlər. Həqiqətən, bunu belə etmək olardı:

Natural ədəd:
- 1 – natural ədəddir.
- Natural ədədin ardınca gələn ədəd – natural ədəddir.

Faktorial anlayışını da (**n!=1·2·3·...·(n-1) ·n**)rekursiya vasitəsi ilə vermək olar:

Faktorial:
- 0! = 1 (qəbul olunub).
- n! = n*(n-1)!

Rekursiv prosedura və funksiyalar

Müasir proqramlaşdırma dillərində prosedura və funksiyalar özü özlərini çağıra bilərlər.

Özü özünü çağıran prosedura və funksiyalar rekursiv adlanırlar.

Məsələn, faktorialı hesablayan funksiyanı aşağıdakı kimi yazmaq olardı:

```
int Factorial ( int n )
  {
   if ( n<=0 ) return 1;              //  1 qaytarmaq
   else   return n*Factorial(n-1);      // rekursiv çağırış
   }
```

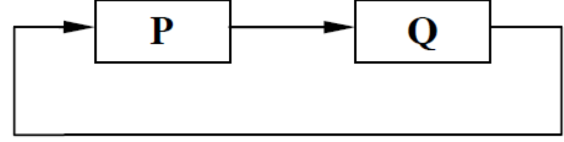
Diqqət yetirsək, görmək olar ki, əgər n>0-dan, onda **Factorial** funksiyası özü özünü çağırır. Bu məsələnin həlli üçün funksiyadan yox, rekursiv proseduradan da istifadə etmək olardı. Rekursiv prosedura nəticəni yalnız istinad parametri vasitəsi ilə qaytara bilər (bu parametrin qabağında **&** işarəsi dayanmalıdır). Prosedurаnın rekursiv çağırışı zamanı bu parametr öz qiymətini dəyişəcəkdir.

```
void Factorial ( int n, int &fact )
 {
  if ( n == 0 ) fact = 1;     // rekursiya bitdi
  else
    { Factorial( n-1, fact);   // rekursiv çağırış, (n-1)! hesablanır
      fact *=n;                 // n! =n*(n-1)!
     }
  }
```

Funksiyadan fərqli olaraq prosedura istinadla göndərilən bir neçə parametrin qiymətini dəyişə bilər.

Dolayısı rekursiya

Bəzən daha mürəkkəb konstruksiyadan istifadə edilir. Burada prosedura özünü bilavasitə yox, digər prosedura və ya funksiya vasitəsi özünü çağırır. Bu konstruksiya aşağıdakı sxemlə təsvir olunur:



Bu cürə rekursiya **dolayısı rekursiya** adlanır.

Sonsuz rekursiya

Rekursiv prosedura və funksiyalardan istifadəsi zamanı çox böyük təhlükə yaranır: rekursiv çağırışlar sonsuz davam edə bilər. Ona görə də, belə funksiyalarda hər addımda yoxlanılan şərti nəzərə almaq lazımdır. Əgər şərt yerinə yetirilmirsə, onda funksiyanın çağırışları dayanır.

Faktorialı hesablayan funksiya üçün belə şərt **n<=0** – dır. Sübut edək ki, yuxarıda göstərilmiş prosedurada rekursiya sonlu olacaqdır.

1. **n** sıfıra bərabər olduğu halda rekursiv çağırışlar dayandırılır.
2. Hər yeni rekursiv çağırışda **n**-nin qiyməti **1** vahid azalır (bunu prosedurаnın çağırışından görmək olur: **Factorial (n-1, ...)**).
3. Ona görə də, əgər əvvəl **n>0** idisə, onda n-in qiyməti tədricən azalaraq, **0**-ra çatacaq və rekursiya bitəcəkdir.

Rekursiv prosedura və ya funksiya rekursiyanı bitirən şərti özündə saxlamalıdır.

Rekursiyadan nə vaxt istifadə etmək lazım deyil

Hər yeni rekursiv çağırış zamanı kompüter aşağıdakı əməliyyatları yerinə yetirir:

1. Hər addımda hesablamaların vəziyyəti qeyd olunur.
2. Stekdə (yaddaşın xüsusi sahəsi) lokal dəyişənlərin **yeni** yığımı yaranır.

Belə ki, funksiyanın hər çağırışında yeni yaddaş ayrılır və ona vaxt sərf olunur, rekursiyadan istifadə zamanı aşağıdakıları nəzərə almaq lazımdır:

Rekursiyanın dərinliyi (çağırışların sayı) kifayət qədər az olmalıdır.

Dərinliyi çox olan rekursiyadan istifadə edən proqram daha gec yerinə yetiriləcəkdir və bu isə **stekin dolmasına** gətirib çıxara bilər. Ona görə də,

əgər rekursiyadan istifadə etməməklə məsələni asanlıqla həll etmək olursa, onda bu halda rekursiyadan istifadə etmək məqsədə uyğun deyil.

Məsələn, faktorialın hesablanması məsələsi **for** dövrü vasitəsi ilə asanlıqla həll olunur (bu cürə həll etmə **iterativ** və ya **dövrlü** adlanır):

```
int Factorial ( int n)
  {
  int i, fact = 1;
  for ( i=2; i<=n; i++ )
    fact *=I;
   return fact;
   }
```

Yuxarıdakı funksiya rekursiv funksiyadan daha sürətli işləyir.

Sübut olunub ki, istənilən rekursiv proqram rekursiyadan istifadə etmədən yazıla bilər, lakin bu realizasiya çox mürəkkəb ola bilər.

Misal. Fibonaçi ədədlərini (f_i) hesablayan funksiyanı tərtib etmək lazımdır. Ədədlər aşağıdakı kimi hesablanır:

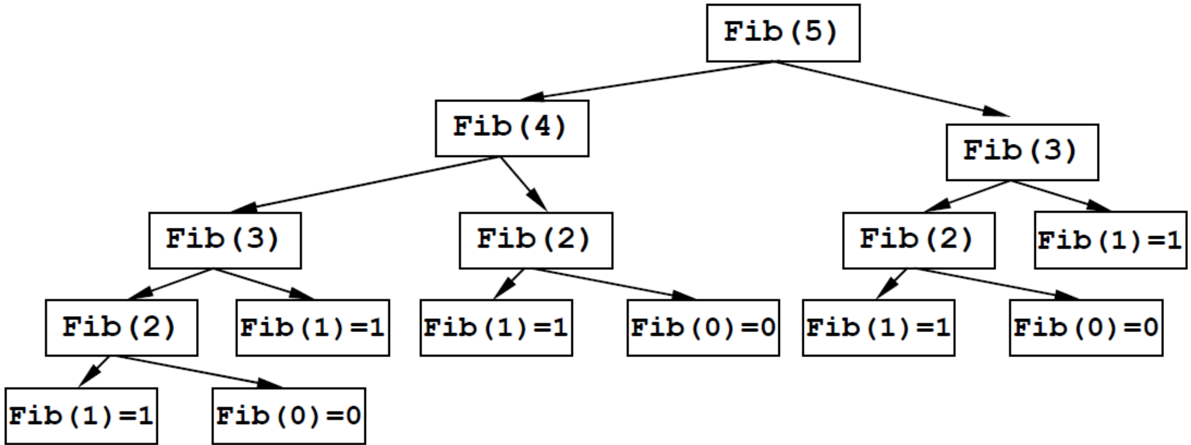
1. $f_0 = 0, f_1 = 1.$
2. $f_i = f_{i-1} + f_{i-2},\ bütün\ i > 1.$

Rekursiyadan istifadə edərək aşağıdakı funksiyanı yazmaq olar:

```
int Fib ( int n )
  {
   if ( n == 0 ) return 0;
   if ( n == 1 ) return 1;
   return Fib(n-1) + Fib(n-2);
   }
```

Qeyd edək ki, hər rekursiya çağırışı **n>1** olduğu halda funksiyanın 2 çağırışına gətirib çıxardır. Bəzi ifadələr, təkrar olaraq bir neçə dəfə hesablanır. Ona görə də, böyük **n**-lər

üçün bu alqoritm faydalı deyil. **Fib(5)**-in hesablama sxemi ağac şəkilində aşağıda göstərilmişdir.



Qeyd edək ki, növbəti Fibonaçi ədədinin qiyməti əvvəlki iki ədəddən asılıdır. Bu qiymətləri **f1** və **f2** dəyişənlərdə saxlayaq. Əvvəlcən **f1=1** və **f2=0** qəbul etsək, sonrakı Fibonaçi ədədini x-a yazaq. Növbəti addımda **f2**-nin qiyməti artıq lazım deyil və onun yerinə **f1**, **x**-in qiymətini isə **f1**-ə köçürürük.

```
int Fib2 (int n )
{
  int i, f1 = 1, f2 = 0, x;
  for ( i=2; i<=n; i ++)  {
     x = f1 + f2;            //  növbəti ədəd
     f2 = f1; f1 = x;        //  qiymətlərin sürüşdürülməsi
     }
  return x;
}
```

Bu cürə yazılmış funksiya böyük **n**-lər üçün (**>20**) rekursiv funksiyadan xeyli sürətli işləyir. Nəticə:

əgər rekursiyadan istifadə etmədən daha yaxşı nəticə almaq mümkündürsə, onda rekursiyadan istifadə etmək gərək deyil.

Rekursiyalı axtarış

Misal. Təqdim edilmiş cümlədə verilmiş sözün neçə dəfə təkrar olunduğunu təyin etmək lazımdır.

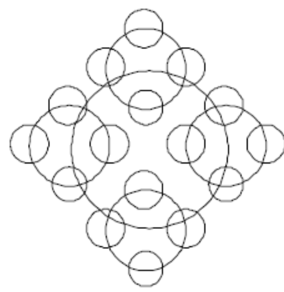
Rekursiv proseduradan istifadə etməklə, alqoritmi aşağıdakı kimi təsvir etmək olar:

1. **strstr** funksiyası vasitəsi ilə sözün birinci daxil olmasını axtarırıq. Əgər tapılmasa, onda stop;
2. sözlərin sayı = 1 + cümlənin qalan hissəsində sözlərin sayı.

```
int HowMany (char *s, char *word)
{
  char *p = strstr(s, word);           // sözün birinci daxil olmasını axtarırıq
  if ( !p ) return 0;                  // əgər söz tapılmırsa, onda nəticə - 0
  return 1+ HowMany( p+strlen(word), word);  // söz bir dəfə tapılıbsa, onda axtarışı davam edirik
 }
```

Funksiya qısa və aydındır, lakin sürətə görə çox da səmərəli deyil.

Rekursiyalı fiqurlar



Şəkillərin və əyrilərin əksəriyyəti rekursiv çəkilir. Sadə misala baxaq. Şəkilin mərkəzində böyük çevrə yerləşir. Onun üzərində 4 kiçik diametrli çevrənin mərkəzləri yerləşir, onların üzərində isə daha dörd çevrənin mərkəzi yerləşir və s. - cəmi **N** müxtəlif diametr (**N** rekursiya səviyyəsi).

```
void RecCircle ( float x, float y, float R, int n)
{
  float k = 0.5;
  circle (x, y, R );           //  çevrəni çəkirik
  if ( n == 1 ) return;        // əgər hamısı çəkilibsə, onda çıxış
  RecCircle(x+R, y, k*R, n-1);        // dörd rekursiv çağırış
  RecCircle(x-R, y, k*R, n-1);
  RecCircle(x, y+R, k*R, n-1);
  RecCircle(x, y-R, k*R, n-1);
}
```

Bu cürə fiqurun çəkilməsi üçün prosedurаnın 4 parametri var: əsas çevrənin mərkəzinin **(x,y)** koordinatları, onun radiusu – **R** və çəkilən səviyyələrin sayı – **n**. Hər növbəti səviyyədə radius **k** dəfə azalır (bizim misalda **k=0.5**), səviyyələrin sayı **1** azalır və növbəti səviyyə üçün çevrələrin koordinatları yenidən hesablanır.

Sübut edək ki, rekursiya sonlu olacaqdır. Proseduradan göründüyü kimi, **n=1** olduqda rekursiya bitir. Yeni rekursiya çağırışında səviyyələrin sayı 1 azalır, ona görə də, əgər əvvəldən n>0, onda o, 1 qiymətinə çatacaq və rekursiya bitəcəkdir. Mənfi ədədlərdən qorunmaq üçün (əgər kimsə səviyyələrin sayını mənfi göstəribsə, onda rekursiya sonsuz olacaqdır) şərti belə düzəltmək lazımdır:

```
if ( n<= 1 ) return;
```

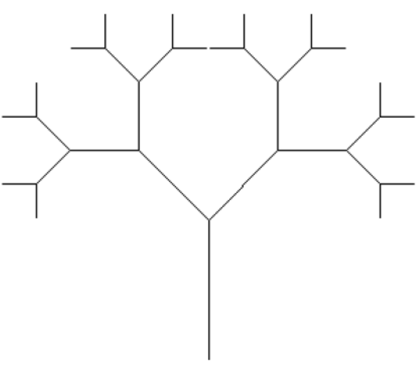
Əsas proqram çox qısa olur. Burada qrafiki pəncərəni açıb, 1 dəfə **RecCircle** prosedurasını çağırmaq lazımdır.

```
#include <conio.h>
#include <graphics.h>
//  bura proseduranı daxil etmək lazımdır
```

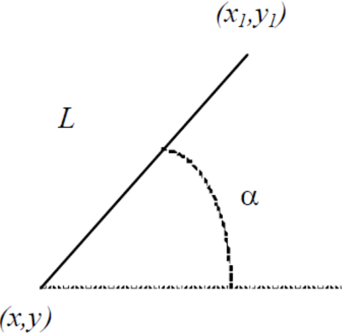
```
main ()
{
  initwindow(600, 500);
  RecCircle (300, 250, 100, 3 );   // 3 səviyyə
  getch ();
  closegraph();
 }
```

Pifaqor ağacı

Pifaqor ağacı belə çəkilir: ağacın "gövdəsi" – uzunluğu **1** olan parçadır. Gövdənin yuxarı hissəsində iki budaq var. Budaqların arasında olan bucaq **90** dərəcədir, onların uzunluğu **k*1**-ə bərabərdir (burada **k<1**). Bu cürə verilmiş səviyyələrin sayı qədər təkrar olunur. Şəkildə **k=0.7** və səviyyələrin sayı **5** olan Pifaqor ağacı göstərilmişdir.



Proseduranı yazmaq üçün həndəsə və triqonometriya qanunlarından istifadə etmək lazım olacaqdır. Əgər (x,y) – budağın başlanğıc koordinatlarıdırsa və α onun əyilmə bucağıdırsa, onda budağın ucunun (x_1, y_1) koordinatları aşağıdakı kimi hesablanır:



$$x_1 = x + L * \cos(\alpha)$$
$$y_1 = y - L * \sin(\alpha)$$

y oxunun aşağı istiqamətləndirilməsinə görə **y_1**-in düsturunda çıxma işarəsi yazılır. **(x_1, y_1)** nöqtəsindən çıxan növbəti səviyyənin sağ budağı **$\alpha+\pi/4$** bucağa, sol budağı isə **$\alpha-\pi/4$** bucağa əyilir. **sin** və **cos** funksiyaların istifadəsi üçün proqrama əlavə **math.h** başlıq faylını qoşmaq lazımdır. Proseduranın 5 parametri var: budağın başlanğıc koordinatları - **(x,y),** budağın uzunluğu – **L**, radianla ifadə olunmuş əymə bucağı – **angle** və çəkilən səviyyələrin sayı – **n**.

```
void Pifaqor (float x, float y, float L, float angle, int n)
{
float k = 0.7, x1, y1;
x1 = x + L*cos(angle);     // budağın ucunun koordinatları
y1 = y -  L*sin(angle);
line (x,y,x1,y1);                 // gövdənin çəkilməsi
if ( n<=1) return;              // əgər hər şey çəkilibsə, onda çıxış
Pifaqor (x1,y1, k*L, angle+M_PI/4, n-1);          // rekursiv çağırışlar
Pifaqor (x1, y1,k*L, angle-M_PI/4, n-1);
}
```

Əsas proqram əvvəlki proqrama oxşadır.

18. STRUKTURLAR

Struktur nədir?

Çox vaxt məlumatların emalı zamanı müxtəlif tip verilənləri özündə saxlayan bloklarla işləmək lazım gəlir. Məsələn, kitabxana kataloqunda kitab haqqında məlumat müəllifin adını (simvol sətri), kitabın adını (simvol sətri), nəşr ilini (tam ədəd), səhifələrin sayını (tam ədəd) və s. əhatə edir.

Yaddaşda bu məlumatları saxlamaq üçün adi massiv yararlı deyil, çünki massivdə bütün elementlər eyni tipdə olmalıdırlar. Əlbəttə ki, müxtəlif tipli massivlərdən istifadə etmək olardı, lakin bu rahat olmayacaqdır (yaxşı olardı ki, kitab haqqında bütün məlumatlar yaddaşın bir hissəsində yerləşərdi).

Müasir proqramlaşdırma dillərində müxtəlif tipli verilənləri özündə saxlayan verilənlərin xüsusi tipi var.

Özündə bir neçə sahə saxlayan (müxtəlif tipli elementlər və ya strukturlar) verilənlər tipi **struktur** adlanır.

Strukturların tətbiq sahəsi verilənlər bazaların, siyahıların və s. təşkilidir.

Elan və inisializasiya

Belə ki, struktur verilənlərin yeni tipidir, onu proqramın əvvəlində təsvir etmək lazımdır. Məsələn:

```
struct Book {
     char author[40];        //  müəllif, simvol sətri
     char title [80];          //  ad, simvol sətri
     int year;                    // nəşr ili, tam ədəd
     int pages;                 // səhifələrin sayı, tam ədəd
     };
```

Bu cürə təsvirdən sonra heç bir yaddaş ayrılmır, yaddaşda hələ ki real strukturlar yoxdur. Bu fraqment sadəcə olaraq kompilyatora onu göstərir ki, proqramda bu tipdə elementlər *ola bilər*. Yaddaşı ayırmaq üçün aşağıdakı cümləni yazmaq lazımdır:

```
Book b;
```

Bu operator yaddaşda **b** adlanan **Book** tipli struktura yer ayırır (Book – tipin adı, b isə konkret nümunənin adıdır). Elan zamanı struktur elementlərinin bəzilərinə və ya hamısına fiqurlu mötərizələrdə vergüllə ayırıb başlanğıc qiymət vermək olar. Sahələr elan olunmuş ardıcıllıqla doldurula bilər. Boş qalmış sahələrə ayrılmış yaddaş sıfırlaşdırılır.

```
Book b = {
   "Nizami Gencevi",
```

```
 "Xemse",
 1998 };
```

Struktur sahələri ilə iş

Adla müraciət

Bütün struktura müraciət etmək üçün onun adını yazmaq lazımdır. Ayrı-ayrı sahələrə müraciət etmək üçün strukturun adı, sonra **nöqtə**, daha sonra sahənin adı yazılmalıdır. Struktur elementləri ardıcıl olaraq daxil edilməlidir. Struktur sahəsi ilə tipi eyni olan dəyişən kimi işləmək olar. Ədədi dəyişənlər cəbri əməliyyatlarda iştirak edə bilərlər, sətirlərə isə standart funksiyaları tətbiq etmək olar.

```
Book b;
strcpy (b.author, "Nizami Gencevi "0;
b.year = 1998;
```

Ünvanla müraciət

Tutaq ki, strukturun yaddaşdakı ünvanı məlumdur. Bildiyiniz kimi, ünvanı göstəriciyə yazmaq olar. Struktur sahəsinə onun ünvanı ilə müraciət etmək üçün xüsusi **->** operatorundan istifadə edirlər.

```
Book b;     // struktur yaddaşdadır
Book *p;    // struktura göstərən göstərici
p = &b;     // strukturun ünvanını göstəriciyə yazmaq
strcpy ( p ->author, "Nizami Gencevi ");   // ünvanla müraciət
p ->year = 1998;
```

Daxiletmə və xaricetmə

Element-element daxiletmə və xaricetmə

Klaviaturadan daxiletmə və ekrana və ya mətn fayla xaricetmə zamanı strukturun hər sahəsi ilə adi dəyişənlərdəki kimi ayrıca işləmək lazımdır. Aşağıda göstərilmiş misalda verilənlər **Book** tipli **b** adlı struktura klaviaturadan daxil olunur və **books.txt** mətn faylının axırına yazılırlar.

```
Book b;
FILE *fp;
printf ( "Muellif: ");
gets ( b.author );
printf ( "Kitabin adi: " );
gets (b.title);
printf ( "Neshr ili, sehifelerin sayi ");
scanf ( "%d%d", &b.years, &b.pages );
fp = fopen ( "books.txt", "a" );    //  faylın axırına əlavə
```

```
fprintf (fp, “%s\n%s\n%d\n %d\n”,
          b.author, b.title, b.years, b.pages );
fclose ( fp );
```

Binar faylla işləmə

Strukturları binar fayllara yazmaq çox asandır, çünki bir dəfəyə bütöv strukturu və ya bir neçə strukturu oxumaq və ya yazmaq olar. Binar fayldan oxunma zamanı **fread** funksiyasına lazım olan yaddaş oblastının ünvanını (oxunmuş verilənləri hara yazmaq), bir blokun ölçüsünü və blokların sayını ötürmək lazımdır. Strukturun ölçüsünü avtomatik hesablamaq üçün **sizeof** operatorundan istifadə edilir.

```
Book b;
int n;
FILE *fp;
fp = fopen( “books.dat”, “rb” );
n = fread ( &b, sizeof(Book), 1, fp);
if ( n == 0 ) {
    printf ( “Faylin oxunmasi zamani sehv” );
    }
fclose ( fp );
```

Belə ki, **b** - **Book** strukturunun bir nümunəsidir, **sizeof(Book)** əvəzinə **sizeof(b)** yazmaq olardı. **fread** funksiyası müvəffəqiyyətlə oxunmuş elementlərin sayını (bizim misalda strukturların sayını) qaytarır. Ona görə də, əgər **n** dəyişəninin qiyməti sıfıra bərabərdirsə, bu o deməkdir ki, oxunma uğursuz oldu və səhv haqqında məlumatı ekrana çıxartmaq lazımdır.

Strukturu binar fayla yazmaq üçün **fwrite** funksiyasından istifadə edilir. Funksiyanın parametrləri **fread** funksiyasının parametrləri ilə üst-üstə düşür. Aşağıdakı fraqmentdə struktur **books.dat** binar faylının axırına əlavə olunur.

```
Book b;
FILE *fp;
//  burada məlumatları struktura daxil etmək lazımdır
fp = fopen( “books.dat”, “ab” );
fwrite (&b, sizeof(Book), 1, fp );
fclose( fp );
```

Kopyalama

Misal. Tutaq ki, yaddaşda iki eyni tipli struktur üçün yer ayrılmışdır. Birinə müəyyən məlumatlar yazılıb. Bütün məlumatları birinci strukturdan ikinci struktura köçürmək lazımdır.

Tutaq ki, strukturların tipi **Book**, adları isə **b1** və **b2**-dir. Bu məsələnin üç həll üsulu var. Ən çətin üsul – hər sahənin ayrıca köçürülməsindən ibarətdir:

```
Book b1, b2;
// burada b1 strukturu daxil edirik
strcpy (b2.author, b1.author );
strcpy( b2.title, b1.title );
b2.year = b1.year;
b2.pages = b1.pages;
```

Yaddaş bloklarının kopyalanması üçün xüsusi **memcpy** funksiyadan istifadə etmək olar. Bu funksiyadan istifadə etmək üçün proqrama **mem.h** başlıq faylı əlavə etmək lazımdır.

```
#include <mem.h>
Book b1, b2;
// burada b1 strukturu daxil edirik
memcpy(&b2, &b1, sizeof(Book) ); //hara yazmaq, haradan yazmaq ve ne qeder bayt
// yazmaq
```

Ən sadə üsul – üçüncüdür. Aşağıdakı sətri yazmaq kifayətdir:

```
b2 = b1;
```

Bu halda proqram bir strukturun bütün bitlərini digərinə köçürəcəkdir.

Strukturların massivləri

Böyük həcmli informasiyanın emalı üçün strukturlardan istifadə olunur, ona görə də proqramlarda strukturların massivlərindən istifadə olunur. Onlar adi qaydada elan olunurlar, lakin əvvəlcə strukturun özünü (yeni tip kimi) təsvir etmək lazımdır.

Strukturun sahəsinə müraciət etmək üçün nöqtədən istifadə olunur, lakin belə halda kvadrat mötərizələrin içində strukturun nömrəsini göstərmək lazımdır. Məsələn:

```
Book A[20];
...
A[12].pages = 50;
for (i=0; i<20; i++)      // bütün strukturlara görə dövr
  puts(A[i].title);       // bütün kitabların adlarını çap etmək
```

Əgər binar fayllardan istifadə olunursa, onda bütün strukturların daxil və xaric edilməsi bir sətirlə həyata keçirilir. Binar fayldan istifadə edərək, aşağıdakı məsələni həll etmək lazımdır.

Misal. **books.dat** faylına Book tipli strukturlar yazılmışdır. Məlumdur ki, onların sayı 100-dən çox deyil. Fayla yazılmış bütün strukturları oxuyub, bütün kitablar üçün nəşr ilini 2009-a dəyişib, onları yenidən həmin fayla yazmaq tələb olunur.

Belə ki, strukturların sayı 100-dən çox deyil, yaddaşda 100 struktura yer ayırırıq.

```
Book b[100];
```

Fayldan bir dəfəyə 100 strukturu oxumaq üçün aşağıdakı operator yazılmalıdır:

```
n = fread( &b[0], sizeof(Book), 100, fp);
```

Həqiqətən oxunmuş strukturların sayını təyin etmək məqsədilə **fread** funksiyasının qaytardığı **n** qiymətini (oxunmuş strukturların sayı) yadda saxlayaq. Proqramın tam yazılışı aşağıdakı kimi olacaqdır:

```
#include <stdio.h>
struct Book {                    // verilənlərin yeni tipinin elan olunması
      char author[40];
      char title[80];
      int year;
      int pages;
      };
main()
{
   Book b[100];               // strukturlar massivinə yer ayrılması
   int i, n;
   FILE *fp;
  fp = fopen( “books.dat”, “rb” );     // 100 strukturu oxuyuruq
  n = fread( &b[0], sizeof(Book), 100, fp);   /* həqiqətən oxunmuş strukturların sayının müəyyən edilməsi */
  fclose( fp );
  for ( i=0; i<n; i++)         // oxunmuş verilənlərin emalı
     b[i].year = 2009;
  fp = fopen(“books.dat”, “wb”);   // n strukturun fayla yazılması
  fwrite (b, sizeof(Book), n, fp);
  fclose( fp );
}
```

Yaddaşın dinamik ayrılması

Tutaq ki, ölçüsü qabaqcadan məlum olmayan strukturların massivini yaratmaq lasımdır. Bunun üçün:

1. uyğun tipli göstəricini elan etmək lazımdır;
2. **new** operatoru vasitəsi ilə yaddaşı ayırıb, yaddaş blokunun ünvanını yadda saxlamaq lazımdır;
3. yeni oblastı adi massiv kimi istifadə etmək lazımdır.

```
Book *B;
int n;
printf( “Bazada neche kitab var? “);
scanf( “%d”, &n );       // massivin ölçüsünü daxil edirik
B = new Book[n];      // yaddaşı ayırırıq
// burada B massivi ilə işləyirik
```

```
delete B;                  // yaddaşı boşaldırıq
```

Bəzən yaddaşda bir struktur saxlanmalıdır. Onun ünvanını əldə edib, onu göstəriciyə yazırıq. Strukturun sahəsinə necə müraciət etmək olar?

Variantlardan biri - göstəricidən istifadə etməkdir. Əgər **p** - **Book** tipli strukturun göstəricidirsə, onda **author** adlı sahəyə müraciət üçün **(*p).author** yazılmalıdır. Bu yazı o deməkdir ki, “**p** göstəricisi göstərən strukturdan **author** adlı sahə lazımdır”.

C dilində başqa variant da mümkündür: **p->author**. Aşağıdakı misalda bir struktur üçün dinamik yaddaş ayrılır, onun sahələrinin qiymətləri klaviaturadan oxunur, sonra isə struktur **books.txt** mətn faylının sonuna yazılır.

```
Book *p;
FILE *fp;
p = new Book;   // 1 struktur üçün yaddaş ayırmaq
printf( “Muellif  “);      // göstərici vasitəsi ilə sahələrin daxil edilməsi
gets(p->author);
printf(“Kitabin adi  “);
gets(p->title);
printf(“Nesh ili ve sehifelerin sayi   “);
scanf(“%d%d”, &p->year, &p->pages);
fp = fopen(“books.txt”, “a”);     // faylın sonuna yazmaq
fprintf(fp, “%s\n%s\n%d\n%d\n”,          // göstərici vasitəsi ilə müraciət
            p->author, p->title, p->year, p->pages);
fclose( fp );
delete p;        // yaddaşı boşaltmaq
```

Strukturların prosedura parametrlərində istifadə edilməsi

Digər verilənlər kimi strukturları prosedura və funksiyalarda parametr əvəzi kimi də istifadə etmək olar. **year** sahəsinə (kitabın nəşr ili) **2009** qiymətini yazan proseduranı nəzərdən keçirək.

Parametrlərin qiymətlərinin ötürülməsi

Əgər prosedura parametri aşağıdakı kimi elan olunubsa, onda proseduranın işi zamanı strukturun yaddaşda (stekdə) kopyası yaranır, və prosedura bu kopya ilə işləyir. Bu yanaşmanın iki çatışmayan cəhəti var:

- birincisi, prosedura sahələrin qiymətlərini dəyişdirə bilmir, və bunun nəticəsində proqram düz işləməyəcəkdir;
- ikincisi, strukturların ölçüsü böyük ola bilər və yeni kopyanın yaradılması stekdə (yaddaşda) yer çatışmamazlığına gətirib çıxarda bilər.

Parametrlərin istinadla ötürülməsi

Adətən, strukturları funksiya və proseduralara *istinad parametri* kimi ötürürlər (elan zamanı strukturun adından sonra **&** işarəsi yazılır).

```
void Year2009( Book &b )
{
  b.year = 2009;
}
```

Bu halda proseduraya strukturun ünvanı ötürülür və prosedura yaddaşda yerləşən real verilənlərlə işləyir. Prosedura tərəfindən edilən dəyişikliklər yaddaşa yazılmış verilənlərə təsir göstərir. Prosedura daxilində struktur ilə iş əsas proqramda görünən işdən fərqlənmir.

Parametrlərin ünvanla ötürülməsi

Strukturların istinadla ötürülməsi yalnız C++ dilində mövcuddur və C dilinin standart operatorlarına daxil deyil. Buna baxmayaraq, C dilində strukturları *ünvanla* ötürmək olar. Bu halda struktur daxilində struktur sahələrinə **->** operatoru vasitəsi ilə müraciət etmək lazımdır.

```
void Year2009( Book *b )    // parametr – strukturun ünvanıdır
{
b ->year = 2009;            // göstərici vasitəsi ilə ünvana müraciət
}
main()
{
     Book b;
     Year2009 (&b);         // strukturun ünvanı ötürülür
}
```

Açar əsasında çeşidləmə

Belə ki, strukturların tərkibində çoxlu sayda sahələr var, onda onları necə çeşidləmək olar? Bu çeşidləmə məsələnin qoyuluşundan asılıdır, lakin burada da ümumi qaydalara riayət etmək lazımdır.

Çeşidləmə məqsədilə strukturun bir sahəsi seçilir və o, **açar sahəsi** və ya **açar** adlanır. Bütün strukturlar *açar sahəsinin qiymətlərinə* görə nizamlanır, digər sahələrin qiymətləri nəzərə alınmır.

Digər problem ondan ibarətdir ki, çeşidləmə zamanı massiv elementləri yerlərini dəyişirlər. Strukturlar böyük həcmli ola bilərlər, və onda onların kopyalanması çox vaxt aparacaqdır. Ona görə də yerdəyişməni göstəricilər vasitəsi ilə yerinə yetirirlər.

Struktur tipli massivinin çeşidlənməsini *göstəricilər* vasitəsi ilə həyata keçirirlər.

Yaddaşda göstəricilər massivi (**p**) təşkil edilir. Əvvəlcə göstəricilər uyğun strukturla göstərirlər, yəni **p[i]** göstəricisi ***i*** nömrəli strukturla bağlıdır. Sonra isə göstəriciləri elə dəyişmək lazımdır ki, uyğun strukturların açar sahələri nizamlanmış şəkildə olsun. Məsələnin həlli üçün **Book** strukturuna göstərən **PBook** yeni tipi elan etmək lazımdır.

```
typedef Book *PBook;
```

Aşağıdakı proqramda strukturlar kitabların nəşr ilinə görə çeşidlənir. Çeşidləmə alqoritmi SortYear prosedurasında yazılıb. Proseduranın iki parametri var: göstəricilər massivi və strukturların sayı. Prosedurada “hava qabarcığı” üsulundan istifadə edilir.

```
void SortYear ( PBook p[ ], int n )
  {
  int i, j;
  PBook temp;     // köməkçi göstərici
  for (i=0; i<n-1; i++ )   // çeşidləmə
    for (j=n-2; j>=I; j--)
      if ( p[j+1]->year < p[j]->year)   // əgər kitabların nəşr illəri düzgün sıralanmayıbsa, onda
         { temp = p[j];                  // göstəricilər öz yerlərini dəyişirlər.
           p[j] = p[j+1];
           p[j+1] = temp;
          }
   }
```

Nəşr illərinə görə çeşidlənmiş kitablar haqqında məlumatları ekranda əks etdirən əsas proqram aşağıda göstərilmişdir:

```
#include <stdio.h>
const int N = 20;
struct Book {           // strukturun təsviri (yeni tip)
       char author[40];
       char title[80];
       int year;
       int pages;
       };
typedef Book *PBook;   // yeni tip – strukrurla bağlı göstərici
// burada SortYear prosedurası yazılmalıdr
main ()
{
Book B[N];        // strukturlar massivi
PBook p[N];        // göstəricilər massivi
// burada strukturlara məlumatları daxil etmək lazımdır
for ( i=0; i<N; i++ )    // götəricilərin başlanğıc yerləşməsi
    p[i] = &B[i];
SortYear (p, N);       // göstəricilərə görə çeşidləmə
for (i=0; i<N; i++)       // bütün strukturlara görə dövr
   printf( “%d: %s\n”, p[i]->year, p[i]->title);    // göstəricilər vasitəsi ilə məlumatların çapı
}
```

19. PROQRAMLARIN LAYİHƏLƏNDİRİLMƏSİ

Proqramların layihələndirmə mərhələləri

Məsələnin qoyuluşu

Proqramların layihələndirməsinin ən vacib mərhələlərindən biri məsələnin qoyuluşudur. Əvvəlcə proqram təbii dildə ifadə edilir. Əgər iri layihə yerinə yetirilirsə, onda proqrama aid bütün tələblər yazılı müqavilədə öz əksini tapmalıdır.

Qoyuluşu yaxşı və **pis** olan məsələləri fərqləndirmək olar. Yaxşı qoyulmuş məsələlərdə ilkin verilənlər aydın görünür, verilənlər arasında olan bütün əlaqələr təyin olunub. Bu da birmənalı nəticəyə gətirib çıxardır.

Əgər birmənalı həll üçün ilkin verilənlərin sayı kifayət deyilsə, yəni ilkin verilənlər və nəticə arasında əlaqələr tam aydın göstərilməyibsə, onda **məsələnin qoyuluşu o qədər yaxşı olmur.**

Məsələn, "Uşaq məktəbə getməlidir. Məktəbə çatmaq üçün ona nə qədər vaxt lazım olacaqdır?" məsələsi yaxşı qoyulmamış məsələlərdən biridir. Bu məsələnin həlli üçün kifayət qədər məlumat verilməyib. Amma, əgər bu məsələnin qoyuluşunda məktəbə qədər məsafə və şagirdin sürəti göstərilsə idi, onda onu asanlıqla həll etmək olardı.

Verilənlər modelinin yaradılması

Praktiki proqramlaşdırmada əsas problemlərdən biri – məsələnin formal modelinin (riyazi və məntiq dillərində), verilənlərin lazımı strukturun (dəyişənlər, massivlər, strukturlar) və əlaqələrin qurulmasıdır. Proqramlarda ən ciddi səhvlər verilənlər modelinin düzgün qurulmaması ilə bağlıdır.

Alqoritmin hazırlanması

Bu mərhələdə **alqoritmi** seçmək və ya yenidən hazırlamaq lazımdır.

Məsələnin həllinə gətirib çıxaran müəyyən edilmiş sonlu əməliyyatlar ardıcıllığı **alqoritm** adlanır.

Alqoritm qabaqcadan seçilmiş verilənlər modelini nəzərə almalıdır. Bəzən əvvəlki mərhələyə qayıtmaq lazım gəlir, çünki seçilmiş alqoritmi istifadə etmək üçün başqa cür verilənləri istifadə edilməlidir. Məsələn, massivlərin əvəzinə strukturlardan istifadə daha məqsədə uyğundur.

Proqramın hazırlanması

Tapşırığı başa düşmək üçün o, kompütere yaxın olan dillərdən birində tərtib olmalıdır, yəni alqoritm əsasında **proqram** yazılmalıdır.

Proqramlaşdırma dilində yazılmış alqoritm **proqram** adlanır. Çox vaxt kompütere yazılmış əmrlərin (instruksiyaların) yığımına da proqram deyilir.

Proqramı yazarkən proqramlaşdırma dilini seçmək vacibdir. Peşəkar, yəni yüksək səviyyəli proqramları, adətən, C və ya C++ dilində tərtib edirlər.

Proqramın düzənnəməsi

Proqramda səhvlərin axtarışı və düzəldilməsi **düzənnəmə (ing. debugging)** adlanır.

Proqramların düzənnəməsi üçün **debugger** adlanan xüsusi proqramlardan istifadə edilir. Debuggerin köməyi ilə aşağıdakıları yerinə yetirmək olar:

- proqramı addım-addım (hər əmrdən sonra dayanaraq) yerinə yetirmək;
- proqramda **kəsilmə nöqtələrini** qoymaq;
- yaddaşa yazılmış dəyişənləri və prosessor registrlərini müşahidə etmək və qiymətlərini dəyişmək;
- düzənnəmə zamanı **profiler** adlanan proqramdan da istifadə etmək olar. Bu proqram vasitəsi ilə proqram hissələrinin neçə vaxta yerinə yetirilməsini təyin etmək və ləng işləyən hissələri optimallaşdırmaq olar.

Sənədlərin hazırlanması

Proqramla bağlı sənədlər düzgün tərtib olunmalıdır. Adətən, istifadəçi üçün təlimat (*User manual*), bəzən proqramçılar üçün təlimat (*Programmer's manual*), alqoritmin və proqram xüsusiyyətlərinin ətraflı təsviri hazırlanır. Bu sənədlər istifadəçiyə aydın və sadə dildə təqdim olunmalıdır.

Proqramın sınaqdan keçirilməsi

Xüsusi hazırlanmış insanlar (tester) tərəfindən proqramın yoxlanılması onun **sınaqdan keçirilməsi** adlanır.

Sınağın məqsədi hazırlanma zamanı proqramda aşkar olunmayan səhvləri üzə çıxartmaqdır. Peşəkar proqramların sınağı iki mərhələdən ibarət olur:

- **alfa-testing** – istehsalçı firmanın əməkdaşları tərəfindən həyata keçirilən sınaq;
- **beta-testing** – digər firmalar və təşkilatlarda proqramın ilkin versiyasının sınağı; çox vaxt proqramın beta-versiyaları İnternet vasitəsi ilə sərbəst yayımlanır.

Şərti olaraq, proqramı o vaxt düz hesab etmək olar ki, nə vaxt seçilmiş ilkin test verilənlər üçün onun yerinə yetirilməsi düz nəticəyə gətirib çıxardır.

Proqramın sınağı üçün nəticələri qabaqcadan məlum olan etalon misalları hazırlamaq lazımdır.

Müşayiət etmə

Sifarişçiyə təhvil veriləndən sonra proqramda səhvlərin düzəldilməsi və istifadəçilər üçün məsləhətlərin verilməsi proqramın **müşayiət etməsi** adlanır.

"Aşağıdan yuxarıya doğru" proqramlaşdırma

Qabaqlar bu yanaşmadan tez-tez istifadə edilirdi. Proqramın hazırlanması ən sadə prosedura və funksiyalardan başlayır. Aşağı səviyyəli (sadə) prosedurаları yazaraq, onlardan daha iri və mürəkkəb prosedurаları və funksiyaları qururuq və s. (kiçik kublardan evin tikintisinə oxşayır). Son məqsəd – kiçik kublardan bütün proqramın tam yığılmasıdır.

Üstünlüklər:

- proqramı "sıfırdan" hazırlamaq asandır;
- "sadədən-mürəkkəbə" üsulundan istifadə edərək daha effektiv prosedurаları yazmaq olar.

Çatışmayan cəhətlər:

- Sadə prosedurаların hazırlanması zamanı onları bir növ məsələnin qoyuluşu ilə bağlamaq lazımdır;
- elə ola bilər ki, sonuncu mərhələdə hər hansı kublar (prosedurаlar) çatışmasın;
- proqram çox dolaşıq olur.

Struktur proqramlaşdırma

"Yuxarıdan aşağıya doğru" proqramlaşdırma

Hal-hazırda geniş istifadə olunan yanaşma – "yuxarıdan aşağıya doğru" proqramlaşdırmadır. Bu yanaşma **ardıcıl detallaşdırma üsulu** adlanır.

Əvvəlcə məsələ sadə alt məsələlərə bölünür və əsas proqram yaradılır. Hələ yazılmamış prosedura və funksiyaların əvəzinə heç bir iş görməyən, lakin parametrlərə malik olan prosedura və funksiyalar yazılır. Sonra isə hər alt məsələ daha kiçik alt məsələlərə bölünür və s. Bu proses bütün prosedurаlar yazılana kimi davam edilir.

Üstünlüklər:

- Belə ki, proqramın yazılışı əsas proqramdan başlanır prinsipial səhvlərin ehtimalı çox azdır;
- proqramın strukturu aydın və sadə olur.

Çatışmayan cəhətləri:

- elə ola bilər ki, müxtəlif bloklarda oxşar prosedurаları reallaşdırmağa lazım gələr, qaldı ki onları biri ilə əvəz etmək olardı.

Struktur proqramlaşdırmanın məqsədi

Zaman getdikcə proqramların daha mürəkkəb olması və onların qısa müddətə yazılması və redaktə edilməsi yeni struktur proqramlaşdırma üsullarının yaradılmasına təkan olmuşdur. Aşağıdakı məsələləri həll etməyə lazım gəldi:

- **proqramların etibarlığının artırılması**. Bunun üçün proqram elə yazılmalıdır ki, o asanlıqla sınaqdan keçirilsin və redaktə olunsun.
- **proqramların səmərəliliyin artırılması**. Bunun üçün proqram modulları elə yazılmalıdır ki, digər modullara təsir göstərmədən onları asanlıqla dəyişdirmək olsun.
- mürəkkəb proqramların **hazırlanma müddətinin və qiymətinin** azaldılması.
- **proqramların oxunaqlılığının** yaxşılaşdırılması. Proqram elə tərtib olunmalıdır ki, onun alqoritmini asanlıqla başa düşmək olsun.

Struktur proqramlaşdırmanın prinsipləri

Abstraksiya prinsipi. Proqram elə tərtib olunmalıdır ki, onun komponentləri (tərkib hissələri) asanlıqla başa düşülsün.

Modul prinsipi. Bu prinsipə əsasən proqram sonlu sadə fraqmentlərə bölünür və bu fraqmentləri müstəqil redaktə etmək və sınaqdan keçirmək olar.

Tabeçilik prinsipi. Proqram hissələri arasında əlaqə tabeçilik xarakterli olmalıdır. Bu isə proqramın "yuxarıdan aşağıya doğru" hazırlanmasına gətirib çıxarır.

Yerlilik prinsipi. Elə etmək lazımdır ki, hər modul (prosedura və ya funksiya) öz lokal dəyişənlərindən və əmrlərindən istifadə etsin. Mümkün qədər qlobal dəyişənlərdən istifadə etmək lazım deyil.

Strukturlu proqramlar

Struktur proqramlaşdırma prinsipləri əsasında hazırlanmış proqram aşağıdakı tələblərə cavab verməlidir:

- proqram bir-birindən asılı olmayan modul adlanan hissələrə bölünməlidir;

Modul - müstəqil blokdur. Onun kodu digər modulların kodlarından fiziki və məntiqi fərqlənir. Modul bir məntiqi funksiyanı yerinə yetirməlidir, başqa sözlə, o, müstəqil məsələni həll etməyə bacarmalıdır.

- proqram modulunun işi aşağıdakılardan asılı olmamalıdır:
  - hansı modula onun çıxış verilənləri ötürülməlidir;
  - müraciətlərin sayından;
- proqram modulunun ölçüsü 30-60 sətirdən çox olmamalıdır.
- modul bir giriş və bir çıxış nöqtəsinə malik olmalıdır.
- modullar arasındakı əlaqələr tabeçilik prinsipi əsasında qurulmalıdır.
- hər modulun əvvəlində onun təyinatı, dəyişənlərin (formal parametrlərin) təyinatı, ondan çağırılan və onu çağıran modullar haqqında şərhlər verilməlidir.

- şərtsiz keçid operatorundan (goto) istifadə olunmamalıdır. İstifadə olunursa, onda keçid yalnız modulun çıxış nöqtəsinə verilə bilər.
- modulda alqoritmin mürəkkəb hissələrində əlavə şərhlərdən istifadə etmək məsləhətdir.
- dəyişənlərin və modulların adları başa düşülən olmalıdır. Məsələn, orta_qiymet, elementlerin_sayi və s.
- bir sətirdə birdən çox operator yazılmamalıdır. Əgər hər hansı operatorun yazılışı üçün bir sətir azdırsa, onda onun davamını növbəti sətirdə boşluqlardan istifadə edərək yazmaq olar.
- iç-içə modulların sayı 3-dən çox olmamalıdır.

20. QRAFİKLƏRİN ÇƏKİLMƏSİ

Proqramın strukturu

Böyük proqramların yaradılmasını nümayış etmək məqsədi ilə mürəkkəb proyekti nəzərdən keçirək. Bu proyektə aşağıdakı məsələlər daxildir:

- ekranda koordinat oxlarını qurub, onları nişanlamaq;
- verilmiş iki funksiyanın qrafikini qurmaq;
- funksiyaların kəsişmə nöqtələrinin koordinatlarını müəyyən etmək;
- əyrilərlə məhdudlaşdırılmış bağlı sahəni ştrixləmək;
- ştrixlənmiş fiqurun sahəsini hesablamaq.

Bu cürə işi bir günə yerinə yetirmək olmaz. Böyük proqramların hazırlanması məsələnin sadə alt məsələlərə bölünməsindən başlayırlar. Bizim misal üçün alt məsələlər artıq sadalanmışdır. Fərz edək ki, hər bir alt məsələni bir xüsusi prosedura həyata keçirir. Beləliklə, əsas proqramın strukturunu təyin edib, əsas proqramı yazmaq olar.

```
#include <stdio.h>                    // standart daxiletmə və xaricetmə
#include <graphics.h>                 // qrafiki funksiyalar
#include <math.h>                     // riyazi funksiyalar
//  burada sabitlər və qlobal dəyişənlər elan olunur

// burada bütün funksiyalar və prosedurlar təsvir olunur
//-----------------------------------------------------
//       Esas proqram
//-----------------------------------------------------
main()
{
initwindow (800,600);             // qrafika üçün pəncərə yaratmaq
Axes();                           // koordinatlar oxlarının qurulması və nişanlaması
Plot();                           // qrafiklərin qqurulması
Cross();                          //  kəsişmə nöqtələrinin təyin edilməsi
Hatch();                          // ştrixləmə
Area();                           // sahənin hesablanması
getch();                          // düymənin basılmasını gözləmək
closegraph();                     // qrafiki pəncərənin bağlanması
}
```

Beləliklə, əsas proqram yalnız prosedura və funksiyaların çağırışlarından ibarətdir. Bu proseduralar hələ ki yazılmayıb, onları bir-bir yazacayıq. Proqramın hissə-hissə düzənnəməsi üçün hazır olmayan proseduraların qabağında **//** simvolunu (şərh deməkdir) qoyaq. Proseduralar hazır olduqca, yəni onların kodu proqramın yuxarı hissəsində yazılandan sonra hazır proseduranın adından qabaq **//** işarələrini silmək lazımdır.

Proqramın tərtibatı

Proqramı asanlıqla başa düşmək üçün onun tərtibatında xüsusi qaydalardan istifadə edilir. Bu tərtib qaydaları proqramın gedişinə heç bir təsir göstərmir, lakin onların köməyi ilə proqramda olan xətaları asanlıqla aşkar edib düzəltmək mümkündür. Bu qaydalar aşağıdakılardır:

- Hər prosedura və funksiyanın başlığı (təsviri) olmalıdır. Başlıqda funksiyanın adı, onun işinin təsviri, giriş və çıxış parametrləri, funksiya üçün - qaytarılan qiymət, funksiya tərəfindən çağırılan digər funksiya və proseduraların adları, funksiyanı çağıran prosedura və funksiyalar göstərilməlidir.
- **for, while, do-while** dövrələrin gövdəsi fiqurlu mötərizələrlə birlikdə əsas başlıqdan 2-3 simvol sağa sürüşdürülür. Onda dövrün əvvəlini və sonunu yaxşı görmək olur. Açılan mötərizə bağlanan mötərizə ilə bir səviyyədə olmalıdır. Bu isə çatışmayan mötərizələri asanlıqla aşkar etməyə imkan yaradır.

```
while ( a<b )
   {
    // dövrün gövdəsi
    }
```

- Analoji boşluqlar şərti (**if-else**) və seçim (**switch**) operatorlarında qoyulur.

```
if ( a<b )
  {
   // “əgər” bloku
   }
else
   {
   // “əks halda” bloku
   }
```

```
switch ( k )
  {
   case 1:    //  blok “1”
   case 2:    // blok “2”
   default:   //  susmaya görə
   }
```

- Prosedura və funksiyaların adları başa düşülən olmalıdır.
- Dəyişənlərin adları başa düşülən olmalıdır. Məsələn, sayğac rolunu oynayan dəyişənə **count**, cəmi özündə saxlayan dəyişənə isə **sum** adı verilməlidir.
- Proqramı asanlıqla oxumaq üçün operatorların yazılışında boşluqlardan istifadə etmək məsləhətdir. Məsələn, aşağıdakı operator daha asan oxunur, nəinki ondan sonra yazılan.

```
while ( a < b )  {  /* operatorlar  */ }
```

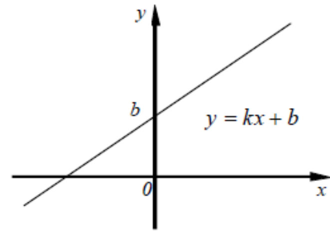
```
while (a<b) {/*operatorlar*/}
```

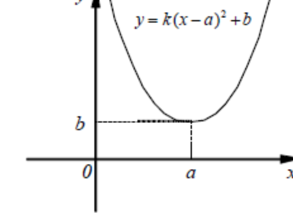
- Prosedura və funksiyalarda ən vacib blokları boşluqlarla və ya “**-**“ simvollardan ibarət ayırıcı sətirlərlə ayırırlar.

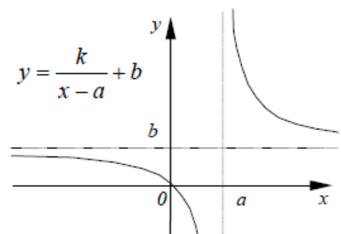
Funksiyaların yazılış qaydaları

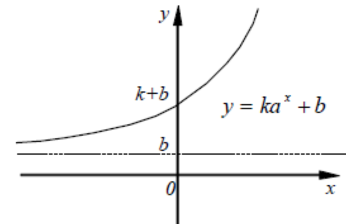
Birbaşa yazılmış funksiyalar

Fərz edək ki, asılı olmayan dəyişən – **x**, funksiyanın qiyməti isə **y**-dir. Ən sadə halda y=f(x) asılılığı məlumdur, yəni ***x***-i bilərək ona uyğun ***y*** qiymətini təyin etmək olar. Sadə funksiya qrafiklərinə düz xətt, parabola, hiperbola, eksponensial funksiya aiddir.



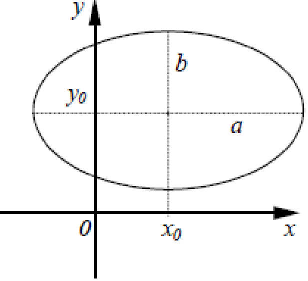






Qeyri aşkar funksiya

Elə funksiyalar vardır ki, onları aşkar **y=f(x)** şəkilində təsvir etmək olmur. Adətən belə funksiyalar **f(x, y)=0** şəkilində göstərilir. Nümunə kimi **ellipsı** və ya **çevrəni** göstərmək olar. Ellipsin tənliyi aşağıdakı kimi yazılır:



$$\frac{(x-x_0)^2}{a^2}+\frac{(y-y_0)^2}{b^2}=1$$

Burada $\mathbf{x_0}$ və $\mathbf{y_0}$ -ellips mərkəzinin koordinatlarıdır, **a** və **b** – yarım oxların uzunluğudur. Xüsusi a=b olduğu halda ellips çevrəyə keçir.

Çox vaxt qeyri aşkar funksiyaları 2 və ya daha çox aşkar funksiyaların kombinasiyası kimi təsvir etmək olur. Məsələn, çevrə üçün belə yazmaq olar:

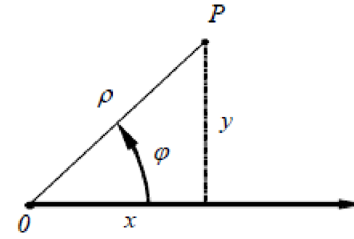
$$y=y_0\pm b\sqrt{1-\frac{(x-x_0)^2}{a^2}}$$

Ellipsin yuxarı hissəsi "+"-la yazılmış, aşağı hissəsi isə "-"-la yazılmış düstura uyğundur.

Ellipsləri və ya çevrələri ekranda çəkərkən arqumentlərin təyin oblastını yoxlamaq lazımdır. Məsələn, yuxarıda yazılmış düsturda kök altı ifadə mənfi ola bilməz, yəni $(x-x_0)^2\le a^2$ və ya $x_0-a\le x\le x_0+a$ olmalıdır.

Polyar koordinatlarla verilmiş funksiyalar

Bəzi mürəkkəb əyrilər üçün **polyar koordinat sistemindən** istifadə edərək sadə ifadələri almaq mümkündür. Burada koordinatların başlanğıcından **0 – polyusdan** və polyusdan üfüqi sağa çıxan **polyar oxdan** istifadə edilir. Polyar ox vasitəsi ilə bucaq hesablanır. Müstəvidə olan istənilən **P** nöqtəsini 2 koordinatla: ρ - nöqtədən polyusa kimi məsafə ilə və φ - polyar ox və **OP** parça arasında olan polyar bucaqla təyin etmək olar.



Dekart sistemindən polyar sisteminə və əksinə keçmək üçün aşağıdakı ifadələrdən istifadə edilir:

$$\begin{cases}\rho = \sqrt{x^2+y^2}\\ \varphi = arctan\frac{y}{x}\end{cases} \qquad \begin{cases}x = \rho cos\varphi\\ y = \rho sin\varphi\end{cases}$$

Məsələn, mərkəzi koordinat sistemində olan çevrənin tənliyi polyar koordinat sistemində çox asan görünür: **ρ=R**. Göründüyü kimi, radius bucaqdan asılı deyil.

Polyar koordinat sistemində müxtəlif növ spirallar da asan təsvir olunur. Məsələn, Arximed spiralı **ρ=aφ,** loqarifmik spiral isə $\boldsymbol{\rho} = \frac{\boldsymbol{a}}{\boldsymbol{\varphi}}$ tənliyi ilə ifadə olunur. Burada $\boldsymbol{a}$ – müəyyən sabitdir.

Parametrik şəkildə verilmiş funksiyalar

Parametrik şəkildə verilmiş funksiyalarda uyğun **x** və **y** üçüncü **parametr** adlanan **t** dəyişəni vasitəsi ilə ifadə olunur:

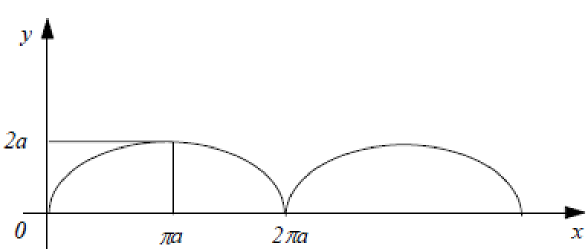
$$\begin{cases}x = f_1(t)\\ y = f_2(t)\end{cases}$$

Məsələn, mərkəzi koordinatlar sisteminin əvvəlində olan ellipsin tənliyi parametrik şəkildə aşağıdakı kimi yazılır:

$$\begin{cases}x = acost\\ y = bsint\end{cases}$$

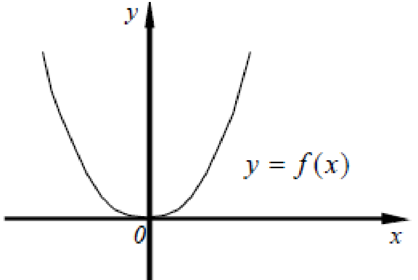
Parametrik şəkildə **sikloid** adlanan əyrini də asanlıqla təsvir etmək olar. λ=1 olduğu halda şəkildə göstərilmiş klassik sikloid əyrisini alırıq, λ<1 olduğu halda qısaldılmış, λ>1 olduğu halda isə uzadılmış sikloid əyrisini alırıq.

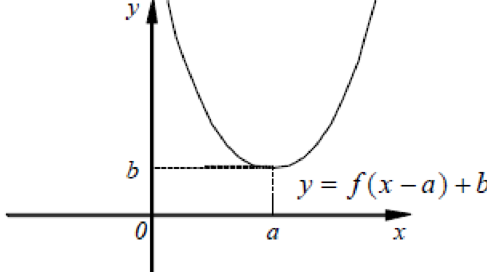
$$\begin{cases}x = a(t - \lambda sint)\\ y = a(1 - \lambda cost)\end{cases}$$



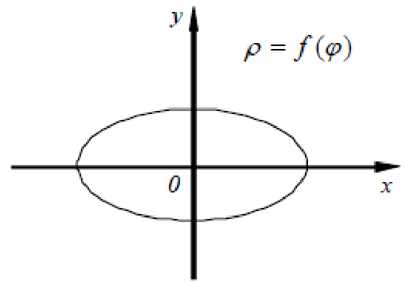
Koordinatların çevrilməsi

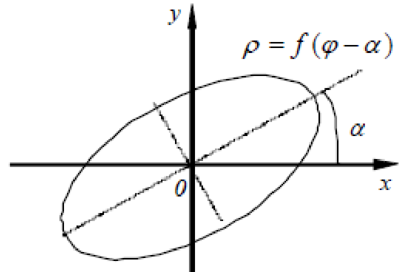
Çevrilmənin ən sadə yolu **paralel köçürülmədir**. Qrafiki **x** oxuna görə **a** qədər, **y** oxuna görə **b** qədər sürüşdürmək üçün funksiyanı **x-a** nöqtəsində hesablayıb nəticənin üzərinə **b** gəlmək lazımdır.





Digər sadə çevrilmə - α bucaqlı dönmədir. Belə halda polyar koordinatlardan istifadə etmək daha məsləhətdir. Həqiqətən, əgər dönmə saat əqrəblərinin gedişinə qarşı aparılırsa, onda $\rho=f(\varphi-\alpha)$ və **x**, **y** hesablamaq lazımdır.





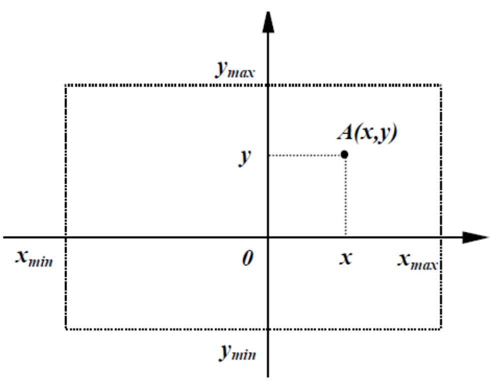
Koordinat sistemi

Koordinat sisteminin təsviri

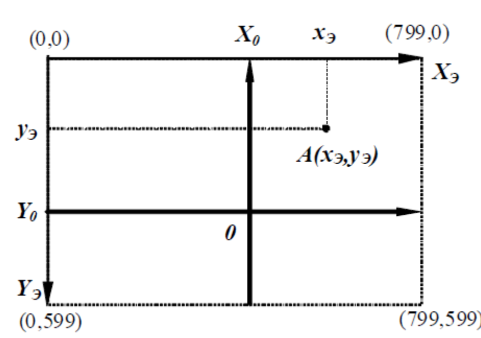
Kompüterdə funksiyaların qrafiklərinin təsviri zamanı iki növ koordinat sistemindən istifadə edilir:

- **riyazi koordinat sistemi**, hansı ki, funksiya bu sistemdə təyin olunub;
- **ekran üçün koordinat sistemi**.

Bu təsvirlər arasında böyük fərq var. Riyazi koordinat sistemində müstəvi sonsuz hesab olunur, ekranda isə biz bu müstəvinin yalnız bir düzbucaq hissəsini göstərə bilərik. Bundan əlavə, ekrandakı koordinat sistemində **x** oxu sağa, **y** oxu isə aşağı istiqamətlənib (riyazi təsvirdə **y** oxu yuxarı yönəlib).



riyazi koordinat sistemi



ekrandakı koordinat sistemi

Tutaq ki, qrafiki rejimdə ekranın ölçüləri 800 x 600 pikseldir. Ekranın digər ölçüləri üçün düsturlarda müəyyən dəyişikliklər etmək lazımdır. Fərz edək ki, müstəviyə pəncərədən baxırıq və müstəvinin **[x_{min} x_{max}]** və **[y_{min} y_{max}]** qiymətlərlə məhdudlaşdırılmış düzbucaqlının bir hissəsini görürük. **A(x,y)** nöqtəsinin piksellərlə ifadə olunmuş ekran koordinatlarını x_e və y_e kimi ifadə edəcəyik.

Miqyas və koordinatların çevrilməsi

Ekranda koordinat oxlarını qurmaq üçün koordinat sisteminin başlanğıcını, yəni **(X_0, Y_0)** nöqtəni seçmək lazımdır ((0,0) koordinatlı nöqtə). Qrafiki böyüdüb kiçiltmək üçün **miqyas əmsalını – k** seçmək lazımdır. Bu əmsal ekranda təsvir olunan parçaların riyazi koordinat sistemi ilə müqayisədə nə qədər kiçik və ya böyük olduğunu göstərəcəkdir. Faktiki olaraq, **k** – ekran koordinat sistemində vahid uzunluqlu parçanın uzunluğudur.

X_0, Y_0 və **k** qiymətlərini bilərək ekranın ekstremal qiymətlərini hesablamaq mümkündür:

$$x_{min} = -\frac{X_0}{k}, \qquad x_{max} = \frac{(800 - X_0)}{k},$$

$$y_{min} = -\frac{(600 - Y_0)}{k}, \qquad y_{max} = \frac{Y_0}{k}.$$

Proqramda funksiyaların əksəriyyəti **X_0, Y_0** və **k** qiymətlərindən istifadə edəcəklər, ona görə də onları qlobal sabitlər (konstantalar) kimi elan etmək məsləhətdir. Qeyd edək ki, **X_0, Y_0** - tam ədədlərdir, lakin **k**-ın qiyməti kəsr də ola bilər.

```
const int X0 = 100, Y0 = 400;
const float k =  3.5;
```

İndi isə istənilən **A(x,y)** nöqtəsinin koordinatlarını **(x_e, y_e)** ekran koordinatlarına çevirək. **x** oxunun hər bir parçası **k** dəfə dartılır və ekranın sol yuxarı küncünə görə **X_0** qədər sürüşdürülür. **y** oxu üçün bu çevrilmələr eynidir, lakin nəzərə almaq lazımdır ki, **y** oxu aşağıya doğru istiqamətlənib. Ona görə də düsturda minus işarəsi əmələ gəlir. Beləliklə, riyazi koordinat sistemindən ekran koordinat sisteminə keçmək üçün aşağıdakı düsturlardan istifadə etmək lazımdır:

```
X = X0 + k*x;
```

```
Y = Y0 – k*y;
```

Koordinatların çevrilməsi çox istifadə olduğu üçün bu çevrilmələri funksiya şəkilində tərtib edib, əsas proqramdan qabaq yerləşdirmək lazımdır.

```
// ---------------------------------------------------------------------------
// SCREENX – x koordinatının ekran koordinatlarına çevirmə
// ---------------------------------------------------------------------------
int ScreenX (float x)
{
 return X0+k*x;
}
// ---------------------------------------------------------------------------
// SCREENY – y koordinatının ekran koordinatlarına çevirmə
// ---------------------------------------------------------------------------
int ScreenY (float y)
{
 return Y0 – k*y;
}
```

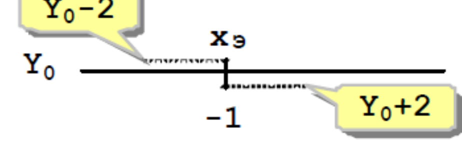
Qeyd edək ki, yuxarıda göstərilən funksiyalar həqiqi parametri qəbul edib tam tipli nəticə qaytarırlar. Yuvarlaqlaşdırma C dilində avtomatik olaraq aparılır.

Koordinat oxları

Yuxarıdakıları nəzərə alaraq koordinat oxlarını belə çəkmək olar:

```
void Axes ( )
{
  line (X0, 0, X0, 599);     // şaquli ox
  line (0, Y0, 799, Y0);     // üfüqi ox
}
```

İndi isə koordinat oxlarına yazıları və cizgiləri əlavə edək. **x** oxunun müsbət hissəsini **1** addımla cizgilərlə (ştrixlərlə) bölək. Belə ki, vahid uzunluqlu parçanın ekrandakı uzunluğu **k** piksellərə bərabərdir, onda x oxunun müsbət hissəsində (sıfırı nəzərə almayaraq) **(800-X0)/k** cizgi olacaqdır. $\mathbf{x_e}$ nöqtənin ekrandakı koordinatı (riyazi sistemdə **x=i**) $\mathbf{X_0+i*k}$ olacaqdır.



İndi şəkilə cizgini və yazını qoyaq. Bir cizgili və yazılı **x** oxunun bir hissəsi şəkildə göstərilmişdir. Cizgini göstərmək üçün kiçik ölçüdə şaquli parça çəkilməlidir. Qeyd edək ki, qrafiki rejimdə ekranda tam və həqiqi ədədləri yox, yalnız simvol sətirləri əks etdirmək olar. Yazıları yazmamışdan əvvəl onları sətrə çevirib, sonra isə ekranda göstərməliyik. Çevirməni **sprintf** funksiyası vasitəsi ilə həyata keçirək. Bu funksiya printf və fprintf funksiyalarını xatırladır, lakin onlardan fərqli olaraq məlumatları ekrana və ya fayla yox, göstərilmiş sətrə çıxardır. Əlbəttə, bu sətrə yaddaşda yer ayırmalıdır.

```
int i;
```

```
char s[10];                    // 10 simvolluq sətr
sprint (s, "%d", i);           // nəticəni s adlı sətrə yazmaq
```

sprintf funksiyanın xüsusiyyəti ondan ibarətdir ki, onun birinci parametri nəticə yazılan sətirdir.

İkinci addımı **outtextxy** funksiyası ilə həyata keçirək. Bu funksiyaya parametrlər kimi mətnin yuxarı sol küncünün koordinatları və simvol sətri ötürülür. Bir rəqəm 8×13 piksellik yer tutur. Hesab edək ki, yazı 2 simvolluq, yəni 16×13 ölçülü yer tutur. Yazını cizgidən bir qədər aşağı yerləşdirməliyik.

```
outtextxy (xe-8, Y0+4, s);
```

İndi isə bütün oxları çəkib, onların üzərində cizgiləri və yazıları yerləşdirək. Cizgiləri **1** addımlı etmək üçün **i**-nin qiyməti **0**-dan **(800-**X_0**)/k**-ya kimi dəyişməlidir və hər addımda cizgini çəkib yazını yerləşdirmək lazımdır.

```
void Axes( )
{
int i, xe;
char s[10];
line (X0, 0, X0, 599);     // oxlarının çəkilməsi
line (0, Y0, 799, Y0);
for (i=0; i<=(800-X0)/k; i++)     // bütün cizgilərə görə dövr
  {
   xe = ScreenX (i);
   line (xe, Y0-2, xe, Y0+2) ;    // cizgilərin çəkilməsi
   sprintf ( s, "%d", i);              // ədədin sətrə çevrilməsi
   outtextxy ( xe-8, Y0+4, s );   // ədədin xaric edilməsi
  }
// burada digər oxların cizgilənməsini etmək lazımdır
}
```

Qrafiklərin çəkilməsi

Standart üsul

Düstur şəkilində verilmiş $y=f_1(x)$ funksiyanın qrafikini (daha doğrusu, ekranda ayrılmış düzbucağa düşən hissəsini) çəkək. **x**-a görə **y**-in hesablanmasını ayrı funksiya kimi tərtib edək. Bu funksiyanın bir həqiqi tipli parametri olacaqdır və həmin nöqtədə hesablanmış qiyməti nəticə kimi qaytaracaqdır. Funksiya mürəkkəb də ola bilər. Məsələn, $y=\sqrt{x}$ funksiya aşağıdakı kimi yazılacaqdır:

```
#include <math.h>
float F1 (float x)
{
  return sqrt(x);        // mütləq qoşmaq lazımdır
```

```
}
```

sqrt funksiyasından istifadə etmək üçün proqramının əvvəlində **math.h** başlıq faylını qoşmaq lazımdır.

İndi isə **[x_{min} x_{max}]** intervala düşən bütün **x**-lər üçün qrafikin nöqtələrini qurmaq lazımdır. Burada intervala düşən nöqtələrin sayı sonsuz olduğu üçün **x**-in qiymətlərini müəyyən **h** addımla dəyişdirmək lazımdır. Onda addımı necə seçmək lazımdır?

Belə ki, ekranda 800 şaquli xətt var (800×600), onda 800-dən artıq addım etmək mənasızdır. Nəzərə alaraq ki,

$$x_{min} = -\frac{X_0}{k}, \qquad x_{max} = \frac{(800 - X_0)}{k}$$

onda addım aşağıdakı kimi seçilməlidir:

$$h = \frac{x_{max} - x_{min}}{800} = \frac{1}{k}$$

Qrafiki qurarkən funksiyanın təyin oblastını yoxlamaq lazımdır, əks halda sıfıra bölmə və ya mənfi ədəddən kökaltı və s. almağa ehtimalı artacaqdır. Mürəkkəb funksiyalar üçün təyin oblastının hesablanmasını C dilində tərtib etmək lazımdır. Əgər verilmiş x təyin oblastına daxildirsə, onda funksiya **1**, əks halda - **0** qaytaracaqdır. Bizim misaldakı funksiya üçün proqramın bu hissəsi 2 üsulla yazıla bilər:

```
int ODZF1 ( float x )
{  if ( x != 0 ) return 1;
     else       return 0;
}
```

```
int ODZF1 ( float x )
{
   return ( x !=0 );
}
```

Sağ tərəfdə yazılmış fraqment daha önəmlidir.

Əgər nöqtə ekrana düşürsə, onda onu qurmaq olar. Nöqtələri ekranda yerləşdirmək üçün riyazi koordinat sistemindən ekran koordinat sisteminə keçmək lazımdır. Bundan ötrü **Point** adlanan xüsusi proseduradan istifadə edəcəyik. Bu prosedura çevirmələri həyata keçirib ekrana düşən bütün nöqtələri verilmiş **color** rəngi ilə çəkir. Bu proseduranı qrafik çəkən PlotF1 və PlotF2 proseduralardan əvvəl yerləşdirmək daha düzgündür.

```
void Point ( float x, floa y, int color)
{
   int xe, ye;
   xe = ScreenX(x);
   ye = ScreenY(y);
   if ( xe >=0  &&  xe <800  &&  ye >=0   &&  ye <600 )
      putpixel (xe, ye, color);
}
```

İndi isə yuxarıda göstərilmiş bütün funksiyaları nəzərə alaraq qrafiki çəkən proseduranı yazmaq olar:

```
void Plot()
{
  float x, h, xmin, xmax;
```

```
  h = 1/k;
 xmin = -X0/k;
 xmax = (800 – X0)/ k;
 for ( x =min; x<=xmax; x+=h )
     if ( ODZF1(x) ) Point (x, F1(x), RED);
}
```

Əgər bir neçə qrafik qurmaq lazımdırsa, onda **y**-ləri hesablamaq və təyin oblastları yoxlamaq üçün uyğun funksiyalar çağırılmalıdır. Bütün bu işləri bir dövrdə görmək olar.

Polyar koordinatlarla verilmiş funksiyalar

Əgər funksiya polyar koordinat sistemində verilibsə, onda dövrdə asılı olmayan koordinat **x** yox, dönmə bucağı - φ olacaqdır. Bucaq radianla ölçülür. 90 dərəcəli bucaq π/2 radiandır (C dilində π - **M_PI** kimi işarə edilir). 0-dan 2π-ə qədər dəyişən bucaq tam dövrə uyğundur.

Əgər lazım gələrsə, onda funksiyanın təyin oblastını müəyyən edən funksiyasından istifadə etmək lazımdır. Bucağın dəyişmə addımı elə seçilməlidir ki, ekrandakı qrafikin ayrı-ayrı nöqtələrdən ibarət oldu bilinməsin.

```
void PlotPolar ()
{
   float phi, x, y, ro,
         phiMin = 0.,            //  minimal bucaq
         phiMax = 2*M_PI,        // maksimal bucaq
         h  = 0.001;             // bucağın dəyişmə addımı
   for ( phi =phiMin; phi <=phiMax; phi +=h )
      if ( ODZF1(phi) )  {       // təyin oblastının yoxlanılması
         ro = F1(phi);           // polyar koordinatla ifadə edilmiş funksiya
         x = ro*cos(phi);        // polyar koordinatlardan
         y = ro*sin(phi);        // Dekart koordinatlara keçid
         Point(x, y, RED);       //  ekranda qirmızı nöqtənin qoyulması
         }
}
```

Parametrik şəkildə verilmiş funksiyalar

Əgər funksiya parametrik şəkildə verilibsə, onda dövrün daxilində asılı olmayan dəyişən **t** olacaqdır. Bu parametrin dəyişmə diapazonunu və addımını eksperimental təyin etmək lazımdır. **t**-in hər qiyməti üçün **x** və **y** hesablanır (fərz edək ki, bunu $\mathbf{F_x(t)}$ və $\mathbf{F_y(t)}$ funksiyalar edir). Parametrik şəkildə verilmiş funksiyanın ekranda qurulması əvvəlki misallara oxşardır.

```
void PlotParametr()
{
    float t, tMin = -10., tMax = 10.,  h = 0.001, x, y;
```

```
for ( t=tMin; t<=tMax; t+=h )
    if ( ODZF1(t))   {
    x = Fx(t);
    y = Fy(t);
    Point(x, y, RED);
    }
}
```

Kəsişmə nöqtələrin təyin edilməsi

Tutaq ki, iki funksiyanın kəsişmə nöqtələrini təyin etmək lazımdır. Əgər **f_1(x)** və **f_2(x)** funksiyalar verilirsə, onda onların kəsişmə nöqtələri aşağıdakı tənlik əsasında təyin olunur.

$$f_1(x) = f_2(x)$$

Təəssüf ki, bu cürə tənliyi nadir hallarda analitik üsulla həll etmək olar. Əgər funksiyalar qeyri xəttidirsə, onda analitik həll mövcud deyil və tənliyin həllini **ədədi üsullarla** təyin edirlər. Ədədi üsullar təqribi üsullardır. Bu o deməkdir ki, tənliyin həllində müəyyən qədər **xəta** olur.

Birbaşa izafə seçim üsulu

Tənliyin primitiv həlletmə üsulu aşağıdakı kimi olur. Tutaq ki, qabaqcadan məlumdur ki, **[a, b]** intervalında funksiyaların bir kəsişmə nöqtəsi var. ε dəqiqliyi ilə kəsişmə nöqtəsinin koordinatını - **x^*** təyin etmək lazımdır.

[a, b] intervalı ε uzunluqlu parçalara bölək və bu parçaların içərisindən elə $[x_1^*, x_2^*]$ interval təyin edək ki, o parçada tənliyin kökü olsun, yəni əyrilər kəsişsin. Bunun üçün **a**-dan **b**-ə qədər bütün **x**-ləri alaraq **g_1·g_2** hasili yoxlamaq lazımdır. **g_1** və **g_2** aşağıdakı ifadələrlə təyin olunur:

$$g_1 = f_1(x) - f_2(x), \qquad g_2 = f_1(x + \varepsilon) - f_2(x + \varepsilon).$$

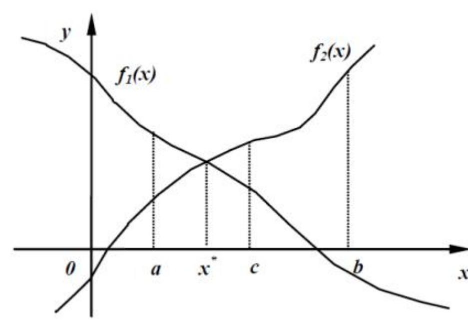
Əgər **g_1** və **g_2** funksiyaların işarələri fərqlidirsə, onda $[x_1^*, x_2^*]$ intervalında tənliyin həlli var.

Başqa cürə bu məsələni belə həll etmək olar: elə x təyin etmək lazımdır ki, **g_1**-in qiyməti minimal olsun. Onda x-in qiymətini təqribi tənliyin həlli kimi qəbul etmək olar.

Dixotomiya üsulu

Tənliklərin ən sadə həlletmə üsulu – **parçaların bölünməsi** və ya **dixotomiya üsuludur**.

Tutaq ki, **[a,b]** parçasında funksiyaların bir kəsişmə nöqtəsi var. **[a,b]** parçasının orta nöqtəsini təyin edirik – c. Sonra isə **[a, c]** parçasında funksiyaların kəsişməsini yoxlayırıq. Əgər kəsişmə varsa, onda növbəti axtarışı bu parçada, əks halda isə **[c, b]** parçasında aparırıq.



Qeyd edək ki, əgər qrafiklərin kəsişməsi **[a,b]** parçasındadırsa, onda funksiyaların fərqi işarəsini dəyişir, yəni əgər **a** nöqtəsində $\mathbf{f_1(x)}$ funksiyasının əyrisi $\mathbf{f_2(x)}$ funksiyasının əyrisindən yuxarı yerləşirsə, onda **c** nöqtəsində vəziyyət əksinə dəyişir. Deyilənləri aşağıdakı bərabərsizliklə ifadə etmək olar:

$$\left(f_1(a) - f_2(a)\right) \cdot \left(f_1(c) - f_2(c)\right) < 0$$

Əgər bərabərsizlik doğrudursa, onda **[a,b]** parçasında kəsişmə var.

Parçanın bölünməsini ε dəqiqliyinə çatdıqda, yəni **[a,b]** parçasının uzunluğu ε-an kiçik olduğu halda prosesi dayandırmaq lazımdır. Aşağıda ε dəqiqliyi ilə dixotomoya üsulunu realizə edən funksiya göstərilmişdir.

```
// -----------------------------------------------------------------------------
// Solve funksiyası [a,b] parçasında kəsişmə nöqtəsini təyin edir
// Giriş : a, b – parçanın sərhədləridir, a<b
//          eps – həlletmənin dəqiqliyi
// Çıxış: x – f1(x)=f2(x) tənliyin həlli
// -----------------------------------------------------------------------------
float Solve ( float a, float b, float eps )
{
float c, fa, fc;
while (fabs(b-a)>eps) {            // hələ ki dəqiqliyə çatmamışıq
    c = (a+b) /2.;                 //  [a,b] parçasının ortası
    fa = F1(a)-F2(a);              // x=a nöqtəsində funksiyaların fərqi
    fc = F1(c)-F2(c);              // x=c nöqtəsində funksiyaların fərqi
    if (fa*fc < 0 ) b=c;           // axtarış oblastı daraldırıq
    else          a=c;
  }
return (a+b)/2.;                    // nəticə - parçanın ortası
}
```

Nəticə olaraq funksiya kəsişmə nöqtəsinin qiymətini (**x**) qaytarır. Həmin nöqtədə funksiyanın **y** qiymətini hesablayıb alınmış koordinatları ekrana çıxartmaq olar. Aşağıdakı **Cross** prosedurası bu əməliyyatları həyata keçirir.

```
void Cross ( )
{
    float y;
    int xe, ye;
    char s[30];
    x1 = Solve(1, 2, 0.001);         // x koordinatın təyini
    y = F1(x1);                      // y koordinatın təyini
    xe = Screen(x1);                 // ekran koordinatların hesablanması
    ye = Screen(y);
    sprintf( s, "x1:%5.2f", x1);    // koordinatların ekrana çıxarılması
    outtextxy(xe+5, ye+2, s);
    sprintf(s, "y1: %5.2f", y );
    outtextxy (xe+5, ye+5, s);
}
```

Burada **x1** – qlobal dəyişəndir. Bu dəyişənə birinci kəsişmə nöqtəsinin koordinatı yazılır. Əgər daha çox kəsişmə nöqtələri varsa, onda bir dənə də **x2** qlobal dəyişəni əlavə etmək lazımdır. Qlobal dəyişənlər bütün proseduralardan əvvəl elan olunmalıdır.

```
float x1, x2;
```

Digər proseduralarda bu dəyişənləri istifadə etmək üçün onları qlobal etmək lazımdır.

Solve funksiyasının bir çatışmayan cəhəti var – o, **f_1(x)** və **f_2(x)** funksiyalarla bağlıdır. Çağırılan **funksiyaların ünvanlarını Solve** funksiyasının parametrlərində vermək daha yaxşı olardı.

Bunu etmək üçün verilənlərin yeni tipini, yəni **funksiya tipli göstəricini** elan etmək lazımdır. Yeganə məhdudlaşdırıcı şərt ondan ibarətdir ki, proseduraya ötürülən bütün funksiyalar eyni tipli olmalıdırlar, yəni eyni tipli parametrlərə malik olmalıdırlar (bizim misalda, **float x**) və qaytardıqları qiymətlər də eyni tipli olmalıdırlar. *func* adlanan yeni tipi təyin etmək üçün proqramın əvvəlində **typedef** əmrindən istifadə edilir.

```
typedef float (*func) (float x);
```

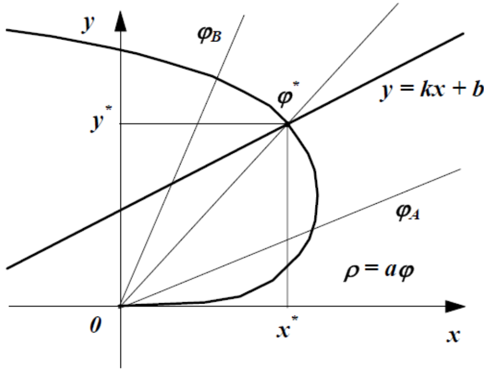
Bu əmr onu göstərir ki, yeni **func** tipi təyin edilir. Bu tip funksiyaya göstəricidir. Funksiya parametr kimi həqiqi ədədi qəbul edib həqiqi nəticə qaytarır. Yeni tipi nəzərə alaraq, Solve funksiyasını aşağıdakı kimi yazmaq olardı:

```
float Solve (func ff1, func ff2,
             float a, float b, float eps )
{
   ...
   fa = ff1(a) – ff2(a);
   fc = ff1(c) – ff2(c);
   ...
}
```

Bu funksiyasının çağırışı zamanı onun birinci iki parametrlərində funksiyaların adları yazılmalıdır. Məsələn,

```
x1 = Solve(F1, F2, 1., 2., 0.0001 );
```

Polyar koordinatlarla verilmiş funksiyalar



Bəzən elə vəziyyət yarana bilər ki, bir funksiya polyar koordinatlarla, digəri isə əyani şəkildə verilsin. Bu halda kəsişmə nöqtələrin təyini mürəkkəbləşir.

Məsələnin həllini misalda göstərək. Tutaq ki, spiralın - $\rho = a\varphi$ və düz xəttin - $y = kx + b$ kəsişmə nöqtəsini təyin etmək lazımdır. Dixotomiya üsulundan istifadə edərək bu məsələni həll edək.

Əvvəlcə hər iki qrafiki quraraq $φ_A$ və $φ_B$ bucaqlarını təyin edək. Bu bucaqlar arasında kəsişmə nöqtəsi yerləşir. Bizim misal üçün **$φ_A$=0** və **$φ_B$=π/2** qəbul etmək olar. Məqsəd - $φ^*$ bucağı və ona uyğun dekart x^* və y^* koordinatlarını təyin etməkdir.

Qeyd edək ki, **$φ<φ^*$** olduğu halda spiral düz xətdən aşağı keçir. Buna görə də, əgər müəyyən φ üçün ρ-nun və ona uyğun $x_ρ$,$y_ρ$ qiymətlərini və $y_1 = kx_p + b$ düz xəttin ordinatını hesablamaq mümkündürsə, onda $y_ρ < y_1$ və ya şərti yerinə yetiriləcəkdir. Analoji olaraq, **$φ>φ^*$** olduğu halda spiral xətdən yuxarı keçir, ona görə də $y_ρ > y_1$ və ya $y_ρ - y_1 > 0$ şərtlər yerinə yetiriləcəkdir.

Beləliklə, parçanın yarıya bölmə üsulunu (dixotomoya üsulu) tətbiq etmək olar, çünki φ bucağı $φ^*$ keçdikdən sonra $y_ρ - y_1$ fərqi işarəsini dəyişir. Məsələnin həll alqoritmini aşağıdakı kimi yazmaq olar:

- $φ^*$ yerləşən **[$φ_A$, $φ_B$]** intervalı təyin etmək; $φ^*$ bucağı kəsişmə nöqtəsinə uyğundur.
- intervalın orta qiymətini təyin etmək: $φ_c = \frac{φ_A+φ_B}{2}$;
- $ρ(φ_A)$ və ona uyğun $y_ρ(φ_A), y_1(φ_A)$ hesablamaq;
- $ρ(φ_C)$ və ona uyğun $y_ρ(φ_C), y_1(φ_C)$ hesablamaq;
- əgər $y_ρ(φ_A) - y_1(φ_A)$ *və* $y_ρ(φ_C) - y_1(φ_C)$ fərqlərin qiymətlərinin işarələri fərqlidirsə (kəsişmə var), onda axtarışı **[$φ_A$, $φ_C$]** intervalında, əks halda **[$φ_C$, $φ_B$]** intervalında aparmaq.
- **[$φ_A$, $φ_B$]** intervalın uzunluğu verilmiş dəqiqlikdən kiçik olarsa, onda axtarış dayandırmalıdır.

Polyar koordinatlarla verilmiş funksiya üçün φ bucağına görə **x, y** kəsişmə koordinatlarının hesablanmasını prosedura vasitəsi ilə aparmaq daha rahat olar. Prosedura nəticə olaraq iki qiyməti, yəni **x** və **y** qaytarmalıdır, ona görə də bu parametrlər **dəyişən parametrlər** kimi əsas proqrama ötürməlidir (onların adlarının qabağında **&** işarəsi qoyulmalıdır).

```
void Spiral ( float phi, float &x, float &y )
{
   float rho, a=1;
   rho = a* phi;                // polyar radiusu hesablayırıq
   x = rho* cos(phi);           // Dekart koordinatları hesablayırıq
   y = rho* sin(phi);
}
```

Bu proseduranı iki dəfə çağırmaq lazımdır – bir dəfə $φ_A$ bucağı üçün, ikinci dəfə isə $φ_C$ bucağı üçün. Kəsişmə nöqtəsini təyin edən funksiya tam şəkildə aşağıda göstərilmişdir. Burada **eps** – prosesi dayandırmaq üçün təyin edilmiş dəqiqlikdir.

```
void SolvePolar (float phiA, float phiB, float eps, float &x, float &y)
{
 float phiC, xA, yA, xC, yC, fA, fC;
while (fabs (phiB-phiA) > eps)  {
    phiC = (phiA + phiB )/2.;          // orta bucaq
    Spiral (phiA, xA, yA);             // phiA bucağı üçün (xA, yA)
```

```
  Spiral (phiC, xC, yC );          //  phiC bucağı üçün (xC, yC)
  fA = yA – F1(xA);                //  phiA bucağı üçün fərqlər
  fC = yC – F1(xC);                //  phiC bucağı üçün fərqlər
  if ( fA*fC < 0 ) phiB = phiC;    // axtarış intervalını daraldırıq
  else             phiA = phiC;
  }
x = xA;
y = yA;
}
```

Parametrik şəkildə verilmiş funksiyalar

Tutaq ki, indi bir əyri parametrik, digəri isə əyani şəkildə verilmişdir. Asılı olmayan dəyişən – **t**-dır. Buna görə də, qrafikləri qurub tənliyin kökü yerləşən **[t_A, t_B]** intervalını təyin etmək lazımdır. İntervalı ardıcıl yarı bölərək kökü təyin etmək olar. Tutaq ki, **F_x(t)** və **F_y(t)** funksiyaları verilmiş **t** qiyməti üçün parametrik əyrinin koordinatlarını hesablayırlar. Onda alqoritm aşağıdakı kimi olur:

- **[t_A, t_B]** intervalının ortasını- t_c təyin etmək;
- **t_A** və **t_C** parametrləri üçün əyrinin **(x_A, y_A)** və **(x_c, y_c)** koordinatlarını təyin etmək;
- y_A və y_c qiymətləri ikinci funksiyanın **x_A** və **x_c** nöqtələrindəki qiymətlər ilə müqayisə etmək. Əgər $\boldsymbol{f_2(x_A) - y_A}$ və $\boldsymbol{f_2(x_c) - y_c}$ fərqlərin işarələri fərqlidirsə, onda $[\boldsymbol{t_A}, \boldsymbol{t_c}]$ intervalında kəsişmə nöqtəsi mövcuddur. Əgər fərqlərin işarələri eynidirsə, onda kəsişmə nöqtəsi $[\boldsymbol{t_c}, \boldsymbol{t_B}]$ intervalında yerləşir.

```
void SolveParameter (float tA, float tB, float eps, float &x, float &y)
{
  float tC, xA, ya, xC, yC, fA, fC;
  while ( fabs(tB-tA) > eps )  {
    tC = (tA +tB)/2.;
    xA = Fx(tA);   yA = Fy(tA);
    xC = Fx(tC);   yC = Fy(tC);
    fA = yA – F2(xA);
    fC = yC – F2(xC);
    if ( fA*fC < 0 ) tB = tC;
    else             tA = tC;
    }
x  = xA;
y  = yA;
}
```

Ümumi hal

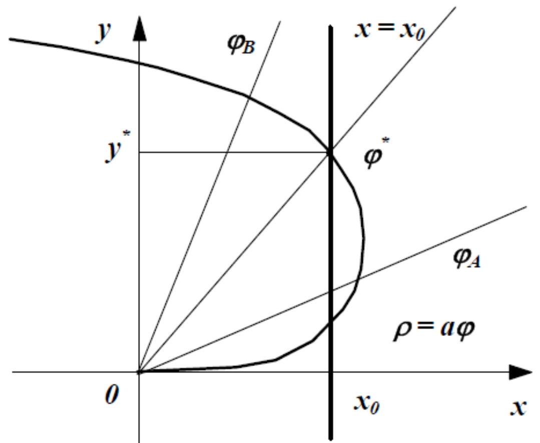
Ümumi hal üçün (məsələn, əyrini hər ikisi parametrik şəkildə verilmişdir) aşağıdakı yanaşmadan istifadə olunur:

- tənliyin həlli üçün asılı olmayan dəyişəni seçmək;
- əyrilərin kəsişmə nöqtəsində işarəsini dəyişən funksiyanı təyin etmək;

- asılı olmayan dəyişən üçün kəsişmə nöqtəsini özündə saxlayan **[A, B]** intervalını təyin etmək. Bu interval elə seçilməlidir ki, kəsişmə nöqtəsindən başqa funksiya intervalın heç bir nöqtəsində işarəsini dəyişməsin;
- verilmiş interval üçün ədədi üsullardan birini (məsələn, dixotomoya üsulu) tətbiq etməklə tənliyin həllini müəyyən etmək.

Şaquli xətlə kəsişmə

Əvvəlki bölmədə müzakirə olunan alqoritmi şəkildə göstərilən funksiyalara tətbiq etmək olar. Burada funksiyalardan biri polyar koordinatlarla verilmişdir, digəri isə - şaquli xətdir.



- asılı olmayan dəyişən kimi φ-i seçək;
- kəsişmə nöqtəsində işarəni dəyişən funksiya: $\boldsymbol{x_\rho(\varphi) - x_0}$;
- qrafik əsasında $[\boldsymbol{\varphi_A}, \boldsymbol{\varphi_B}]$intervalı təyin edirik; dixotomiya üsulunu tətbiq edirik.

Çətinlik $[\boldsymbol{\varphi_A}, \boldsymbol{\varphi_C}]$ intervalında kökün olmasının yoxlanmasından ibarətdir. Bunun üçün aşağıdakı kimi edirlər:

- φ_A və φ_C bucaqları üçün $\mathbf{x_A}$ və $\mathbf{x_C}$ əyrinin nöqtələrini hesablayırlar;
- əgər $\boldsymbol{x - x_A}$ və $\boldsymbol{x - x_C}$ fərqləri müxtəlif işarələrə malikdirsə, onda kəsişmə nöqtəsi $[\boldsymbol{\varphi_A}, \boldsymbol{\varphi_C}]$ parçasında, əks halda $[\boldsymbol{\varphi_C}, \boldsymbol{\varphi_B}]$ parçasında yerləşir.

Parametrik şəkildə verilmiş əyrilər üçün şaquli xətlə kəsişmə nöqtəsi analoji olaraq hesablanır. Bu halda asılı olmayan dəyişənin rolunu parametr oynayır və $\boldsymbol{x_A}$ və $\boldsymbol{x_C}$ koordinatlar $\boldsymbol{F_x(t)}$ funksiyası vasitəsilə hesablanır.

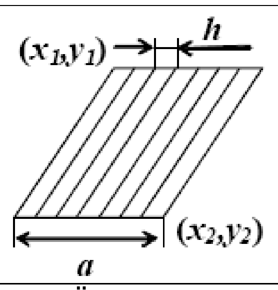
Qapalı oblastın ştrixlənməsi

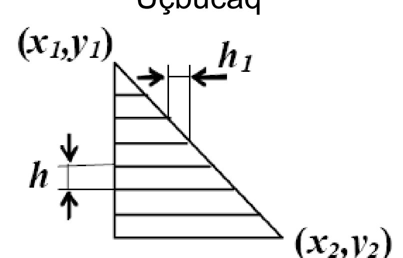
Fərz edək ki, **N** xətt vasitəsilə verilmiş fiquru ştrixləmək lazımdır (fiqurun sərhədləri nəzərə alınmır). Proqramların əksəriyyətində **ştrixləmə addımı,** yəni xətlər arasında məsafə verilməlidir. Əgər **L** uzunluqlu intervalda sərhədləri nəzərə almayaraq **N** xətt çəkilməlidirsə, onda addım $\frac{L}{N+1}$ bərabər olacaqdır.

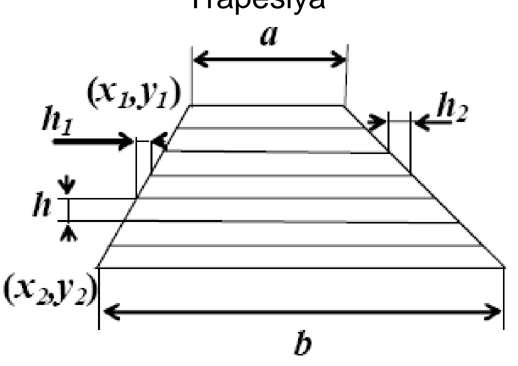
Fiqur	Ştrixləmə üsulu və proqram

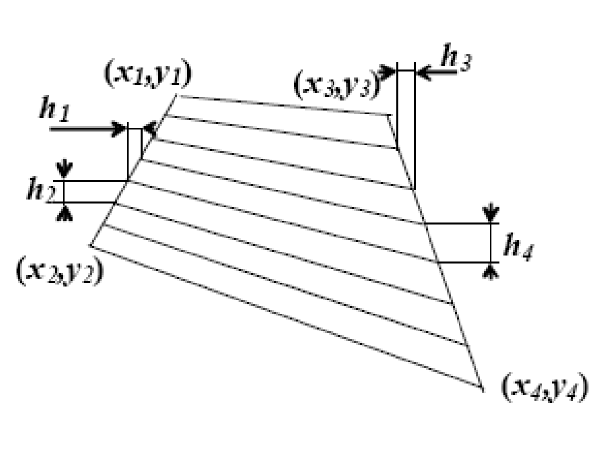
Düzbucaq
(x_1,y_1) h (x_2,y_2)
	Dövrdə **x**-in qiymətini **x1**-dən **x2**-ə qədər **h** addımla dəyişir. Xətlər – şaqulidir:
	`for ( x = x1; x<=x2; x+=h )`
	`line (x, y1, x, y2);`

Paraleloqram	Xətlər paralel və əyridir. Parçanın yuxarı

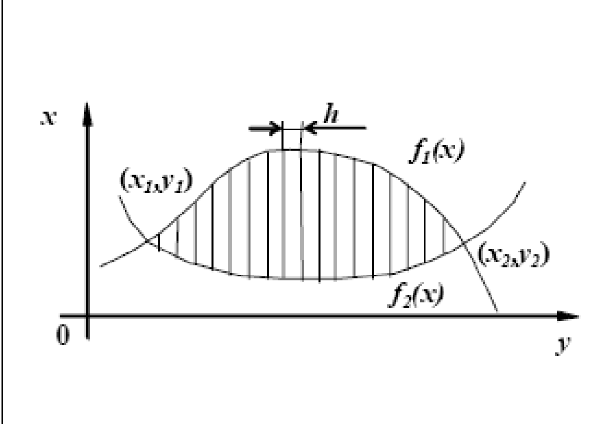

 	
	nöqtəsi aşağı nöqtəyə nisbətən **a** qədər sağa sürüşdürülüb:
	
	`for (x=x1; x<=x1+a; x+=h)`
	`    line (x, y1, x-a, y2);`

Üçbucaq

 	
	Burada çətinlik ondan ibarətdir ki, hər bir növbəti xətt üçün **x** koordinatı əvvəlki koordinatdan $\boldsymbol{h_1} = \frac{x_2 - x_1}{N+1}$ qədər sürüşdürülür və eyni zamanda **y** koordinatı da **h** addımla dəyişir:
	
	`h1=(x2-x1)/(N+1);`
	`xe = x1;`
	`for (y = y1; y<=y2; xe +=h1, y +=h )`
	`    line (x1, y, xe, y);`

Trapesiya

 	
	Parçanın başlanğıc və son koordinatı (**x**) dəyişir:
	
	`h1 = (x1-x2)/(N+1);`
	`h2 = (b – a - x1 + x2)/(N+1);`
	`xs = x1; xe = x1 +a;`
	`for (y =y1; y<=y2; xs -=h1, xe +=h2, y +=h)`
	`   line (xs, y, xe, y );`

İki parça

 	
	Ştrixləməni elə həyata keçirmək lazımdır ki, hər iki parça eyni ölçülü N+1 parçaya bölünsün. Bütün koordinatlar uyğun addımlarla sinxron dəyişməlidir.
	
	`h1 = (x1 – x2) / (N +1 );`
	`h2 = (y2 – y1) / (N + 1);`
	`h3 = (x4 – x3) / (N + 1);`
	`h4 = (y4 -  y3) / (N + 1);`
	`xs = x1; xe = x3; ye = y3;`
	`for  (y=y1;  y<=y2;  xs  -=h1,  xe  +=h3,  y  +=h2,  ye +=h4)`
	`   line (xs, y, xe, y);`

Əyri xətti trapesiya	Adətən $f_1(x)$ və $f_2(x)$ funksiyalar riyazi



koordinat sistemində verilir, ona görə də parçanın çəkilməsi üçün onun koordinatlarını ekran koordinatlarına çevirmək lazımdır:

```
for ( x=x1; x<=x2; x+=h)
  {
  x0 = Screen(x);
  y01 = ScreenY( F1(x) );
  y02 = ScreenY( F2(x) );
  line (x0, y01, x0, y02 ) ;
  }
```

Qapalı oblastın sahəsi

Ümumi yanaşma

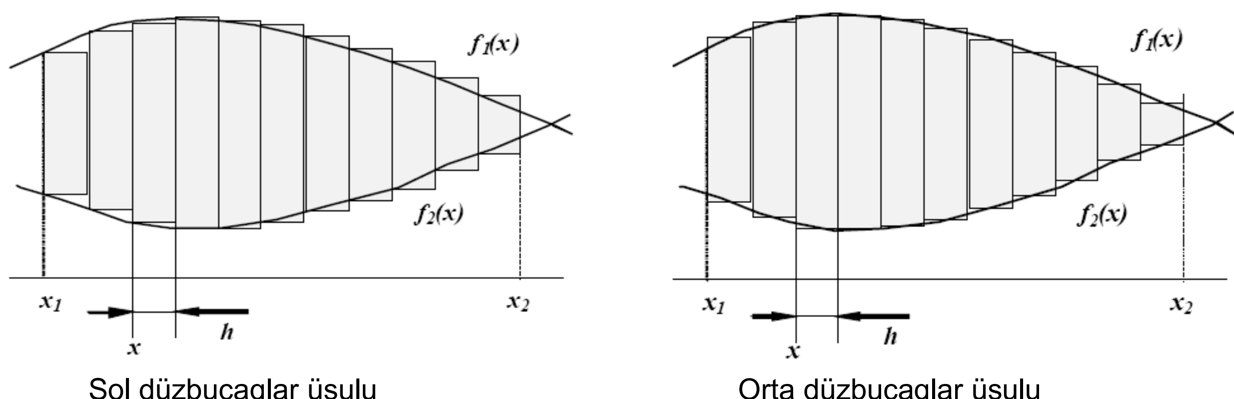
Fiqurların sahəsinin hesablanması üçün çoxlu sayda üsul mövcuddur. Ən geniş istifadə edilən 4 üsulu nəzərdən keçirək və onları müqayisə edək. Bütün bu üsullar ədədi üsullardır. Buna baxmayaraq, istənilən sahəni lazımı dəqiqliyi ilə (təqribən) hesablamaq olar.

Faktiki olaraq, sahə müəyyən inteqrala bərabərdir və çox vaxt analitik üsullarla hesablana bilər.

Tutaq ki, verilmiş fiqur sadə fiqurlardan ibarətdir və onların sahələrini asanlıqla hesablamaq olar. Onda bütün fiqurların sahələrini hesablayıb və sonradan toplayıb verilmiş fiqurun sahəsini almaq olar. Lakin, əgər fiqurun sərhədləri əyri xətlərdisə, onda onu çoxbucaqlılar toplusu kimi dəqiq təsvir etmək olmaz. Beləliklə, bu yanaşmadan istifadə edərək sahəni müəyyən xəta ilə hesablamaq olar. Xətanın miqdarını xeyli azaltmaq olar, məsələn 0,00001.

Düzbucaqlar üsulu

Sahənin hesablanması üçün istifadə olunan ən sadə üsul – düzbucaqlar üsuludur. Fiqur şəkildə göstərdiyimiz kimi düzbucaqlara bölünür. Adətən, bütün düzbucaqların eni - **h** eyni götürülür, lakin bu məcburi deyil.



Sol düzbucaqlar üsulu   Orta düzbucaqlar üsulu

Sol sərhədi **x** koordinata malik olan düzbucağı nəzərdən keçirək. Düzbucağın hündürlüyünü müxtəlif üsullarla təyin etmək olar. aşağıdakı üç üsuldan daha tez-tez istifadə edilir:

- hündürlük bərabərdir $f_1(x) - f_2(x)$ – **sol düzbucaqlılar** üsulu;
- hündürlük bərabərdir $f_1(x+h) - f_2(x+h)$ – **sağ düzbucaqlılar** üsulu;
- hündürlük bərabərdir $f_1\left(x+\frac{h}{2}\right) - f_2(x+\frac{h}{2})$ – **orta düzbucaqlılar** üsulu.

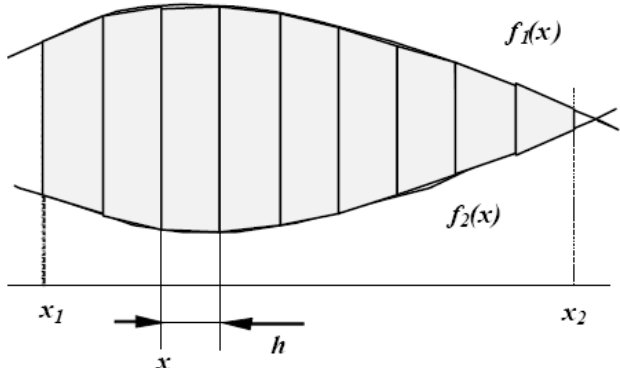
İsbat etmək olar ki, ən dəqiq nəticəni (xətanın miqdarı çox azdır) orta düzbucaqlılar üsulu verir.

Bütöv fiqurun sahəsini hesablamaq üçün sol sərhədin koordinatları $\mathbf{x_1}$-dən $\mathbf{x_2}$**-h** kimi dəyişən bütün ayrı-ayrı düzbucaqlıların sahələri toplanmalıdır. Belə ki, bütün düzbucaqlıların hündürlükləri **h**-a bərabərdir, dövrdə bütün hündürlükləri toplayıb, sonradan alınmış nəticəni **h**-a vurmaq olar. Onda proqram daha sürətlə işləyəcəkdir. Aşağıda orta düzbucaqlılar üsulu əsasında sahəni hesablayan proqram göstərilmişdir.

```
float S = 0., x, h = 0.001;
for ( x = x1; x<=x2-h;  x +=h )
   S +=f1(x+h/2)-f2(x+h/2);
 S *=h;
```

Əgər hesablamaların dəqiqliyini artırmaq lazım gələrsə, onda **h**-ın qiyməti azaldılmalıdır. **Sol və sağ düzbucaqlılar** üsullarının xətası **h**-a mütənasibdir. Bu o deməkdir ki, əgər **h**-ın qiyməti 2 dəfə azalırsa, onda xəta da 2 dəfə azalacaqdır. Orta düzbucaqlılar üsulunun xətası $\boldsymbol{h^2}$ mütənasibdir, yəni $\boldsymbol{h}$-ın qiymətini 2 dəfə azaltmaqla xətanın miqdarını 4 dəfə azaldılmış oluruq.

Trapesiyalar üsulu



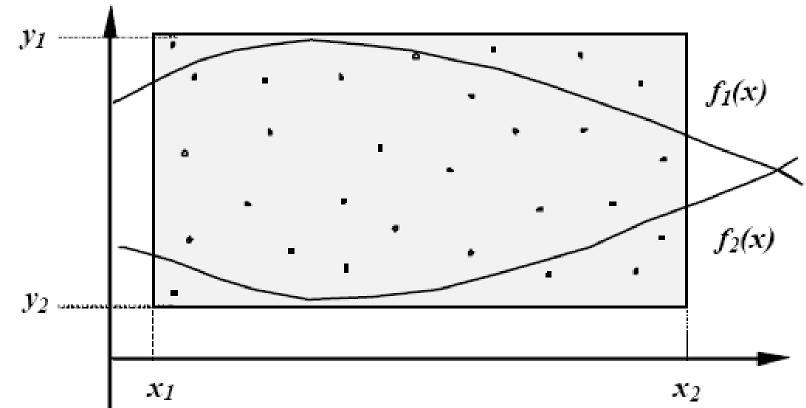
Əyri xətti fiqurun “zolaqlarını” əvəz edən sadə fiqurların arasında **trapesiyanı** göstərmək olar. Bir trapesiyanın sahəsi oturacaqların yarı cəminin və hündürlüyünün hasilinə bərabərdir. Trapesiyaların hündürlüyü eynidir və h-a bərabərdir, oturacaqlar isə uyğun olaraq $f_1(x) - f_2(x)$ və $f_1(x+h) - f_2(x+h)$ bərabərdir. Qeyd edək ki, bir trapesiyanın sağ oturacağı digər trapesiyanın sol oturacağına bərabərdir. Bu faktı hesablamalar zamanı nəzərə almaq olar. Ümumi sahənin hesablanması üçün bütün trapesiyaların sahələri toplanmalıdır və ya onların oturacaqları toplanıb **h/2**-yə vurulmalıdır. Trapesiyaların oturacaqları (birinci trapesiyanın sol və axırıncı trapesiyanın sağ oturacaqları istisna olmaqla) cəmə iki dəfə daxildir. Buna görə də proqram aşağıdakı kimi yazıla bilər:

```
float S, x, h = 0.001;
S = (f1(x1) – f2(x1) + f1(x2) – f2(x2)) / 2.;
for (x =x1+h; x <=x2-h; x +=h)
  S +=f1(x) – f2(x);
S *=h;
```

Trapesiyalar üsulunun xətası h^2 proporsionaldır. Maraqlıdır ki, trapesiyalar üsulunun dəqiqliyi orta düzbucaqlar üsulunun dəqiqliyindən 2 dəfə azdır.

Monte-Karlo üsulu

Təsadüfi ədədlər vasitəsi ilə mürəkkəb modelləşdirmə məsələləri təqribi həll edən üsul Monte-Karlo üsuludur. Xüsusi hal kimi, Monte-Karlo üsulunu verilmiş fiqurun sahəsinin hesablanması üçün istifadə edək.



Üsulun ideyası çox sadədir. Əvvəlcə sahəsi təyin edilən bütöv fiqur sahəsi asanlıqla hesablanan digər fiqurla əhatə edilir. Çox vaxt konteyner rolunu düzbucaq oynayır.

Təsadüfi bərabər paylanmış ədədlər generatoru vasitəsi ilə **düzbucağın daxilinə** düşən nöqtənin **(x,y)** təsadüfi koordinatları generasiya edilir və nöqtənin fiqurun üzərinə düşməyi yoxlanılır. Bu cürə sınaqlar **N** dəfə aparılır (dəqiqliyi artırmaq məqsədi ilə N-nin qiyməti böyük olmalıdır, məsələn, yüz min, və hətta bir neçə million).

Tutaq ki, **N** nöqtədən **M** nöqtəsi fiqurun üzərinə düşüb. Onda, nəzərə alaraq ki, düzbucağın daxilində nöqtələr **bərabər** paylanıb, fiqurun sahəsini aşağıdakı kimi hesablamaq olar:

$$S = \frac{M}{N}(x_2 - x_1)(y_2 - y_1)$$

Burada $(x_2 - x_1)(y_2 - y_1)$ - düzbucaqlı konteynerin sahəsidir.

```
int i, N = 100000, M = 0;
float S, x, y;
for ( i=1; i <=N; i ++)  {
   x = RandFloat(x1, x2);     // təsadüfi x koordinatı
   y = RandFloat (y1, y2);    // təsadüfi y koordinatı
   if ( InsideFigure(x,y))     // əgər nöqtə fiqurun daxilindədirsə,
      M++;                     // sayğacı artırmaq
   }
S = M*(x2 – x1)*(y2-y1)/N;
```

Yuxarıda göstərilmiş proqram **RandFloat** funksiyasından istifadə edir. Bu funksiya verilmiş intervaldan bərabər paylanmış ədədləri qaytarır:

```
float RandFloat (float min, float max)
{
return (float) rand()*(max –min) / RAND_MAX + min;
}
```

Digər (məntiqi) funksiya **InsideFigure**-dir. Əgər **(x, y)** nöqtəsi fiqurun daxilinə düşürsə, onda funksiya 1 (cavab “yes”), əks halda 0 (cavab “No”) qaytarır.

```
int InsideFigure (float x, float y )
{
return f1(x) >=y && y >=f2(x);
}
```

Monte-Karlo üsulu dəqiq nəticə vermir. Üsulun dəqiqliyi düzbucaqda yerləşən təsadüfi nöqtələrin bərabər paylanmasından asılıdır.

Şübhəsiz ki, bu üsuldan yalnız o vaxt istifadə etmək lazımdır ki, nə vaxt digər həlletmə üsullar mövcud deyil, yaxud da, onlar çox mürəkkəbdir.

Monte-Karlo üsulu vasitəsi ilə asanlıqla və rahat həll olunan məsələyə baxaq:

Misal 1. **N** üçbucağın təpə nöqtələrinin koordinatları verilir. Bütün üçbucaqların birləşməsi nəticəsində alınan fiqurun sahəsini hesablamaq lazımdır.

Mürəkkəblik. Üçbucaqlar kəsişə bilər, ona görə də fiqur bir-neçə hissədən ibarət və mürəkkəb quruluşa malik ola bilər.

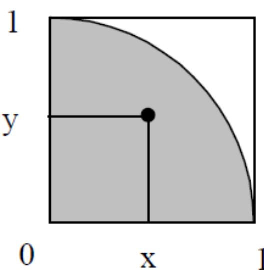
Həll. Məsələnin həlli 2 mərhələdən ibarətdir:

1. Bir-bir yoxlamaqla bütün üçbucaqları özündə saxlayan düzbucağı təyin edirik, yəni hər ox üçün təpə koordinatların maksimal və minimal qiymətlərini təyin edirik;
2. Monte-Karlo üsulundan istifadə edirik. Burada əgər nöqtə heç olmasa bir üçbucağa düşürsə, **InsideFigure** funksiyası 1 qaytarmalıdır.

Misal 2. Monte-Karlo üsulundan istifadə edərək $\boldsymbol{\pi}$ ədədini hesablayın.

Həll. Məlumdur ki, dairənin sahəsi $S = \pi R^2$. Burada R - dairənin radiusudur. Onda $\pi = \frac{S}{R^2}$, yəni dairənin radiusunu bilərək və onun sahəsini ədədi üsulla təyin edib $\boldsymbol{\pi}$ ədədinin qiymətini təqribən hesablamaq olar.

Mərkəzi (0,0)-da olan və radiusu 1-ə bərabər olan çevrəni seçək. Bu çevrənin dörddə birini nəzərdən keçirək. Monte-Karlo üsulundan istifadə etmək üçün təsadüfi generator vasitəsi ilə uzunluğu 1 olan kvadratı bərabər paylanmış nöqtələrlə doldurmaq lazımdır (fərz edək ki, nöqtələrin sayı **N**-ə bərabərdir). Bu nöqtələrin bir hissəsi çevrənin içinə düşəcəkdir (onları **M** kimi işarə edək). Onda bütöv çevrənin sahəsi təqribən $\boldsymbol{\frac{4M}{N}}$ -ə bərabər olacaqdır. Belə ki, radius 1-ə bərabərdir, onda bu nisbət təqribən $\boldsymbol{\pi}$-yə bərabər olacaqdır.



Beləliklə, bu məsələnin həlli üçün iki əməliyyatı N dəfə yerinə yetirmək lazım gələcəkdir:

1. kvadrata düşən nöqtənin təsadüfi (x,y) koordinatların hesablanması (x və y – (0, 1) intervalında dəyişən təsadüfi ədədlərdir);
2. (x,y) nöqtənin çevrənin daxilinə düşdüyünü yoxlamaq lazımdır, yəni $x^2 + y^2 \le R^2 = 1$ bərabərsizlik yerinə yetirilir, yoxsa yox.

21. ƏDƏDİ ÜSULLAR

Tam ədədli alqoritmlər

Bu qrupa tam ədədlərlə işləyən klassik alqoritmlər daxildir.

Evklid alqoritmi (I)

Evklid (3 əsir e.ə.) alqoritmi iki natural ədədin ən böyük ortaq bölənin (ƏBOB) hesablanması üçün nəzərdə tutulmuşdur. Orijinal Evklid alqoritmi aşağıdakı bərabərliyə əsaslanır:

$$ƏBOB(a,b) = ƏBOB(a-b,b) = ƏBOB(a,b-a).$$

Mənfi ədədlərlə işləməmək üçün alqoritmi aşağıdakı kimi ifadə etmək olar:

Evklid alqoritmi. İki ədədin ən böyüyünü böyük və kiçik ədədlərin fərqi ilə əvəz edirik. Bu proseduranı iki ədədin bir-birinə bərabər olanına kimi aparırıq. Nəticədə alınmış ədəd – ƏBOB-dur.

ƏBOB-ü hesablayan funksiyanı iki variantda yazmaq olar:

rekursiv

```
int EBOB( int a, int b)
{
   if (a == b) return a;
   if ( a < b)
        return EBOB (a, b-a);
   else
        return EBOB (a-b, b);
}
```

qeyri-rekursiv

```
int EBOB (int a, int b)
{
    while ( a !=b )  {
      if (a > b ) a -=b;
      else        b -=a;
    }
return a;
}
```

Qeyri-rekursiv alqoritm daha üstündür, çünki o, daha sürətlə işləyir və yaddaşdan (stekdən) az istifadə edir.

Evklid alqoritmi (II)

Ədədlər bir-birindən çox fərqlənəndə birinci Evklid alqoritmi yavaş işləyir (məsələn, ƏBOB(2, 1998) hesablanması üçün 998 addım lazım olacaqdır). Buna görə də, adətən modifikasiya olunmuş Evklid alqoritmindən istifadə edilir. Bu alqoritm aşağıdakı bərabərliyindən istifadə edir:

$$ƏBOB(a,b) = ƏBOB(a,b\%a) = ƏBOB(a\%b,b)$$

Modifikasiya olunmuş Evklid alqoritmi. İki ədədin ən böyüyünü böyük ədədin kiçik ədədə bölmə qalığı ilə əvəz edirik. Bu proseduranı kiçik ədədin sıfıra bərabər olunana kimi aparırıq. Nəticədə alınmış ikinci ədəd – ƏBOB-dur.

C dilində rekursiya olmayan variantı tərtib edək:

```
int EBOB (int a, int b)
{
   while ((a !=0 ) && (b !=0)) {
      if (a>b)  a =a%b;
      else      b =b%a;
      }
return a+b;
}
```

Ən kiçik ortaq bölünən

Hər iki ədədə qalıqsız bölünən ən kiçik ədədə İki ədədin **ən kiçik ortaq bölünəni (ƏKOB)** deyilir.

Əgər verilmiş ədədlərin ƏBOB-u bilinirsə, onda onların ən kiçik ortaq bölünəni çox asanlıqla təyin etmək olar. Tutaq ki, $\boldsymbol{x = x_1 \cdot d}$ və $\boldsymbol{y = y_1 \cdot d}$. Burada $\boldsymbol{d} = \text{Ə}\boldsymbol{BOB(x, y)}$. Onda

$$\text{Ə}KOB(x, y) = xy_1 = x_1y = \frac{xy}{\text{Ə}BOB(x, y)}.$$

ƏBOB-ün hesablanması üçün Evklid alqoritmindən istifadə etmək lazımdır.

Eratosfen xəlbiri

Qədim Yunan riyaziyyatçılar tərəfindən həll olunan digər məsələ - **(1, N)** intervalında yerləşən bütün sadə ədədlərin təyini ilə bağlıdır. İndiyənədək ən sürətlə işləyən *alqoritm Eratosfena* (275-195 il e.ə.) məxsusdur. Əfsanəyə görə, Eratosfen bütün natural ədədləri papirusda sıraya düzüb, hər ikinci ədədi deşirdi (yəni 2-ə bölünəni). Sonra o, hər üçüncü, dördüncü və s. ədədləri deşmişdi. Bu proseduradan sonra qalan “deşilməmiş” ədədlər sadə ədədlərdir, yəni onlar yalnız özlərinə və 1-ə bölünürlər.

Papirusun əvəzinə **A** massivindən istifadə edək. Bu massivdə **A[i]** elementi $(\boldsymbol{i} = \overline{\mathbf{1}, \boldsymbol{N}})$ iki mümükün olan qiymət ala bilir:

1, ədəd “deşilməyib” və sadə ədədə namizəddir;

0, ədəd “deşilib” və ona daha baxılmır.

Yaddaşı qənaət etmək üçün tam tipli dəyişənlər əvəzinə (onlar yaddaşda 4 bayt yer tuturlar) simvol tipli (məsələn, unsigned char 1 bayt yer tutur) dəyişənlərdən istifadə etmək olar.

2-dən $\sqrt{\boldsymbol{N}}$-ə kimi bütün ədədləri nəzərdən keçirərək **2k, 3k, ...** və s., yəni **k**-ya bölünən ədədləri “aradan götürək”. Bu alqoritmi realizə edən proqram aşağıda göstərilmişdir. Qeyd edək ki, yaddaşda **N+1** elementə yerin ayrılması daha rahatdır, onda **A[1] ÷A[N]** elementlərdən istifadə etmək olur. **A[0]** elementi istifadə olunmur.

```
#include <stdio.h>
```

```
main()
{
unsigned char *A;
int i, k, N;
printf( “Maksimal ededi daxil edin “);
scanf( “%d”, &N);
A = new unsigned char[N+1];      // yaddaşda massivə yer ayrılması
if ( A == NULL) return 1;           // səhv olduğu halda çıxış
for ( i=1; i<=N; i++) A[i]=1;
    for (k=2; k*k <=N; k++ )
      if ( A[k] !=0 )
        for ( i=k+k; I <=N; i+=k)  A[i]=0;
for (i=1; i<=N; i++)
    if ( A[i] == 1 ) printf ( “%d\n”, i);
}
```

Alqoritmin əsas **üstünlüyü** **-** onun yüksək sürətlə işləməsi, əsas **qüsuru** isə - yaddaşın böyük həcmindən istifadə edilməsidir. Bu alqoritm proqramlaşdırmanın əsas problemlərindən birini nümayiş edir: sürətin artırılması yaddaşın həcminin artırılmasına gətirib çıxardır və əksinə, yaddaşın qənaəti sürətin azaldılmasına gətirib çıxardır. Real şəraitdə çox vaxt güzəştlərə gedirlər.

Çoxmərtəbəli tam ədədlər

int tipli yaddaş xanasının birinə yerləşməyən çoxmərtəbəli tam ədədlərlə C dilində necə işləmək olar? Yəni bu ədədlər modula görə 2147483647-dan böyükdürlər. Şübhəsiz ki, belə ədədlərə yaddaşda bir neçə xana ayrılmalıdır, lakin bu əməliyyatları əl ilə realizə etməli olacayıq. Burada 2 yanaşma var:

1. ədəd simvol massivi şəkilində saxlanılır. Massivin hər elementi ədədin bir rəqəmini (0÷9) təsvir edir;
2. ədəd tam tipli massivdə saxlanılır. Massivin hər elementi bir və ya bir neçə rəqəmi təsvir edir.

İkinci yanaşmaya daha ətraflı baxaq. Yaddaşın qənaəti üçün bir xanada mümkün qədər çox rəqəm saxlamaq məqsədə uyğundur. Özü də nəzarət etmək lazımdır ki, aralıq əməliyyatların nəticələri seçilmiş verilənlər tipinin maksimal qiymətlərini keçməsin.

Misal üçün 12345678901234567890 ədədi nəzərdən keçirək. Bu ədədi aşağıdakı kimi yazmaq olar:

$$12 \cdot (10^6)^3 + 345678 \cdot (10^6)^2 + 901234 \cdot (10^6)^1 + 567890 \cdot (10^6)^0$$

Bu yazılış əsası **d=1000000** olan say sisteminin yazılışına uyğundur. Bu say sistemində rəqəmlər çoxmərtəbəli onluq ədədlərdir. Verilmiş ədədin yazılışı üçün **4 int** tipli yaddaş xanası lazım olacaqdır.

İndi isə belə ədədlərlə cəbri əməliyyatları yerinə yetirək. Ümumi halda yazılmış **A** ədədin **B** ("qısa" ədəd) ədədə vurulmasına baxaq:

$$A = a_n d^n + a_{n-1} d^{n-1} + \cdots + a_2 d^2 + a_1 d + a_0$$

Nəticədə üçüncü **C** ədədi alırıq. Bu ədəd əsası **d** olan say sistemində təsvir oluna bilər:

$$C = c_m d^m + c_{m-1} d^{m-1} + \cdots + c_2 d^2 + c_1 d + c_0$$

Aydındır ki, $c_0 = (a_0 B)\%d$. Növbəti mərtəbəyə isə əlavə olunur $r_1 = \frac{a_0 B}{d}$ (bölmə qalığı nəzərə alınmır). Növbəti mərtəbələr aşağıdakı ifadələr əsasında hesablanır:

$$c_0 = (a_0 B)\%d \qquad r_1 = \frac{a_0 B}{d}$$

$$c_1 = (a_1 B + r_1)\%d \qquad r_2 = \frac{a_1 B + r_1}{d}$$

$$c_2 = (a_2 B + r_2)\%d \qquad r_3 = \frac{a_2 B + r_2}{d}$$

... ...

Əməliyyatları o vaxt dayandırırıq ki, nə vaxt iki şərt ödənilir:

1. A ədədinin bütün rəqəmləri emal olunub;
2. Növbəti mərtəbəyə keçid (r_i) sıfıra bərabərdir.

Rahatlıq üçün ədədin faktiki uzunluğu ayrı dəyişənində saxlanılır.

100! hesablanması

100 faktorialın qiymətini hesablayaq:

$$100! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot 99 \cdot 100$$

Faktorialı təqribən qiymətləndirib demək olar ki, onun qiyməti $\mathbf{100^{100}}$-dən kiçikdir, ona görə də onun yazılışında **201**-dən çox olmayan rəqəm var. Bundan əlavə, demək olar ki, ədədin sonunda **24** sıfır olacaqdır, çünki sıfırlar 10-a bölünən ədədlərə vurmaqla (10 dənə sıfır), 2 və 5 ilə bitən ədədlərin vurulması nəticəsində (10 dənə sıfır), 25, 50, 75 və 100 ədədlərinin cüt ədədlərə vurulması nəticəsində (4 əlavə sıfır) alınacaqdır.

Belə ki, 10! **8-**lə bitir (axırda yerləşən sıfırları nəzərə almasaq), onda 100! 4-lə bitir (8^{10} ədədinin sonuncu rəqəmi 4-dür, çünki 8^2 4-lə, 8^5 8-lə bitir).

Say sisteminin əsasının düzgün seçilməsini yoxlayaq. Əgər **d=1000000**-dirsə, onda bu ədəd **int** tipli xanaya yerləşəcəkdir. Vurma zamanı ən böyük alınan ədəd 1000000·100 ola bilər, o da tam ədədlərə ayrılmış diapazona **(-2^{31}... 2^{31}-1)** yerləşir.

```
const int d = 1000000;        // d – say sisteminin əsasıdır
int A[40] = { 1 };            //  A[0]=1, qalanlari A[i]=0
int i, k, len = 1, r, s;      // len - ədədin uzunluğudur, r - qalıqdır
for ( k=2; k<=100; k++)  {    // 2,3, ..., 100-ə vururuq
i =0;                         // ən kiçik (0) mərtəbədən başlayırıq
r = 0;                        // əvvəlcə növbəti mərtəbəyə keçirilən ədəd r=0
while ( i<len                 //  hələ ki bütün mərtəbələrə baxılmayıb
```

```
        || r>0)  {              // və ya növbəti mərtəbəyə keçirilən ədəd varsa
        s =A[i]*k+r;         // mərtəbəni vururuq və üzərinə keçirilən ədədi toplayırıq
        A[i]=s%d;          // mərtəbədə qalan ədəd
        r= s / d;            // növbəti mərtəbəyə keçən ədəd
        i++;               // növbəti mərtəbəyə keçid
       }
len=i;            // ədədin uzunluğu
}
for (i=len-1; i>=0; i--)     // ekrana xaricetmə
   if (i==len-1)  printf("%d", A[i]);     // 123 -> 123
   else           printf("%.6d", A[i]);  // 123 -> 000123
```

Qeyd edək ki, ən böyük mərtəbədən başqa uzun ədədin bütün mərtəbələri "%.6d" formatla ekrana çıxarılır. Bu format onu göstərir ki, ekrana mütləq 6 simvol çıxarılmalıdır və əgər çıxarılan ədədin uzunluğu 6 simvoldan azdırsa, onda o, soldan sıfırlarla tamamlanır. Sağ tərəfdən olan sıfırlar həmişə yazılır.

Çoxhədlilər

Məlumdur ki, aşağıdakı görünüşə malik olan funksiyaya **çoxhədli** deyilir:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Çox vaxt çoxhədlinin qiymətini müəyyən $x = x_0$ nöqtəsində hesablamaq lazım gəlir. Əgər bütün əməliyyatlar birbaşa yerinə yetirilsə, onda **n** toplama və $\mathbf{1 + 2 + \cdots + (n-1) + n = \frac{n(n+1)}{2}}$ vurma əməliyyatı lazım olacaqdır. Lakin bu məsələnin həlli üçün daha səmərəli üsul təklif olunmuşdur.

Qorner sxemi

Çoxhədlini aşağıdakı şəkildə təsvir edək:

$$f(x) = (\ldots(a_n x + a_{n-1})x + a_{n-2})x + \cdots a_2)x + a_1 x + a_0$$

Belə halda verilmiş **x** üçün funksiyanın qiymətini hesablamaq üçün cəmi **n** toplama və **n** vurma əməliyyatı tələb olunur. Bu alqoritm **Qorner alqoritmi** adlanır və o, daha sürətlə işləyir. Aşağıda Qorner sxemini realizə edən və C dilində yazılmış proqram göstərilmişdir:

```
float Gorner (float x, float a[ ], int n)
{
  float  v=0.;
  for ( int i=n; i>=1; i--)
     v=(v+a[i])*x;
  v +=a[0];
  return v;
}
```

Ardıcıllıqlar və sıralar

Ardıcıllıqlar

Müəyyən qaydalara uyğun yerləşdirilmiş elementlər yığımına **ardıcıllıq** deyilir.

Ədədlər ardıcıllığına baxaq. Ardıcıllığın bütün elementlərinə 1-dən başlayaraq nömrə verilir. Ardıcıllığı təsvir etmək üçün 2 üsuldan istifadə edirlər:

1. *Rekurent düstur*. Bu düstur əsasında **n** nömrəli elementi hesablamaq üçün əvvəlki elementlərin qiymətlərini bilmək lazımdır. Məsələn, 1,3,5,7,... ardıcıllığı $a_n = a_{n-1} + 2$ rekurent düsturla vermək olar.
2. *n-ci elementin düsturu*. Bu üsulda ardıcıllığın **n**-ci elementini hesablamaq üçün düstur verilir. Əvvəlki ardıcıllıq üçün $a_n = 2n - 1$. Düsturun sağ tərəfində yazılmış ifadə yalnız **n**-dən asılıdır. Məsələn, **ədədi silsilədə** n-ci element aşağıdakı düstur əsasında hesablanır:

$$a_n = a_1 + (n-1)d$$

Burada a_1 – birinci element, d – fərqdir. **Həndəsi silsilə** üçün $a_n = a_1 q^{(n-1)}$. Burada q – silsilənin əmsalıdır.

Aşağıdakı cədvəldə ardıcıllıqların nümunələri göstərilmişdir:

№	ardıcıllıq	a_n
1	1,3,5,7,...	2n-1
2	1,3,7,15,...	2^n-1
3	2,9,28,65,...	n^3+1
4	$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots$	$\frac{1}{n}$
5	$1, 1, \frac{5}{7}, \frac{7}{15}, \ldots$	$\frac{2n-1}{2^n-1}$

Ardıcıllıqlarla bağlı ən çox rast gəlinən məsələlər bunlardır:

- Verilmiş qiymətdən böyük olan azalan ardıcıllığın elementlərin sayını təyin edin;
- Müəyyən şərtə uyğun olan ardıcıllığın birinci elementini təyin edin;
- Ardıcıllığın müəyyən elementlərinin cəmini və ya müəyyən şərtə uyğun olan elementlərin cəmini təyin edin.

Misal. Cədvəldə verilmiş 5-ci ardıcıllıq üçün:

1. 10^{-5} –dən böyük olan elementlərin cəmini təyin edin;
2. Cəmə daxil olan elementlərin sayını təyin edin;
3. 10^{-5}-dən kiçik olan ən birinci elementi təyin edin.

Bu cürə məsələlərin həlli zamanı alqoritmin səmərəliliyinə və dəqiqliyinə diqqət yetirmək lazımdır. Əgər hər addımda ədədləri bir-birinə vurmaq və 2^n hesablamaq lazım gəlirsə, onda bu cürə proqram səmərəli olmayacaqdır.

Bunu başqa cürə etmək olar: 2*n-1 və 2^n ifadə edən iki əlavə **u** və **d** dəyişənləri daxil edək. Hər addımda birinci dəyişən **2**-yə, ikinci dəyişən isə **2** dəfə artacaqdır. Bu məsələnin

həlli üçün aşağıda iki proqram göstərilmişdir. Onlardan biri digərindən 3 dəfə daha tez işləyir. Hər iki proqramda cavab **s, n** və **a** dəyişənlərdə saxlanılır.

```
int n;
float a, s;
n=0; s=0;
while (1)  {
n++;
a=(2*n-1)/(pow(2,n)-1);
if ( a < 1.e-5 ) break;
s += a;
}
```

```
int n;
float a, s, u, d;
n=0; s=0;
u=1; d=2;
while (1)  {
n++;
a = u/(d-1.);
if (a < 1.e-5) break;
u +=2; d *=2;
}
```

Rekurent ardıcıllıqlar

1202-ci ildə Fibonaççi adı ilə tanınan italyan riyaziyyatçı Leonardo Pizanskiy aşağıdakı məsələni təklif elədi:

Fibonaççi məsələsi. Dovşanların bir cütü hər ay iki bala verir – bir erkək və bir dişi, hansılar ki, 2 aydan sonra yenə də iki bala verirlər və s. Soruşulur, bir ildən sonra dovşan cütlüklərin sayı nə qədər olacaqdır, əgər hal-hazırda 1 cütlük var?

Hər ay ərzində dovşanların sayı aşağıdakı ardıcıllıq əsasında dəyişir:

1,1,2,3,5,8,13,21,34,...

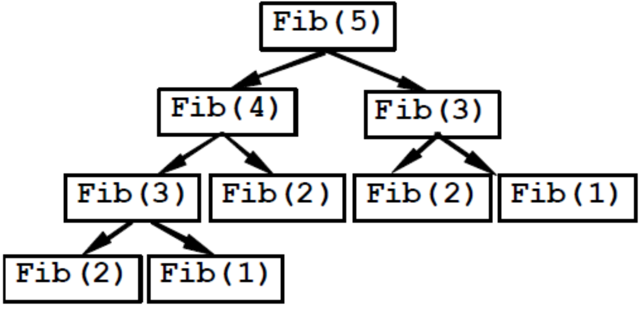
Bu ardıcıllıq **Fibonaççi ardıcıllığı** adlanır. Bu ardıcıllıq **n-**ci elementi rekurent düsturla təyin edir. Düsturda n-ci element əvvəlki 2 element əsasında hesablanır. Hesabatı başlamaq üçün başlanğıc qiymətlər təyin olunmalıdır:

$$a_1 = 1, \quad a_2 = 1,$$

$$a_n = a_{n-1} + a_{n-2} \ , n > 2$$

Bu cürə ardıcıllıq rekursiv funksiya vasitəsi ilə çox asanlıqla realizə olunur:

```
int Fib (int n)
 {
 if ( n < 3 ) return 1;
 else
    return Fib(n-1) + Fib(n-2);
 }
```



Lakin bu proqram səmərəli deyil, çünki, məsələn, **Fib(5)**-in hesablanması üçün funksiya 9 dəfə çağırılmalıdır, bu da səmərəsizdir.

Vəziyyətdən çıxış yolu elə funksiyanın yazılmasıdır ki, rekursiyadan yox, iterasiyadan (dövrdən) istifadə etsin. Qeyd edək ki, a_n elementinin hesablanması üçün $\mathbf{a_{n-1}}$ və $\mathbf{a_{n-2}}$ elementlərin qiymətləri lazımdır, yəni onları yadda saxlamaq lazımdır (proqramda onlar **a1** və **a2** kimi işarə edilmişdir). **a1** və **a2**-nin başlanğıc qiymətləri kimi, uyğun olaraq, **0** və **1** götürülmüşdür.

```
int Fib ( int n )
{
  int a2 = 1, a1 = 0, a, i;
  for ( i=1; i<=n; i++)
    {
    a = a1 + a2;           // növbəti elementi hesablayırıq
     a2 = a1; a1 = a;       // qabağa irəlləmə
     }
 return a;
}
```

Oxşar məsələlər

Misal. Aşağıda göstərilmiş ifadənin qiymətini hesablayın.

$$S = \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{\cfrac{\cdots}{+\cfrac{1}{99 + \cfrac{1}{100}}}}}}$$

Bu məsələnin həlli üçün hesablamaları aşağıdan yuxarıya doğru aparmaq lazımdır.

$s_{100} = \frac{1}{100}$, $s_{99} = \frac{1}{99+\frac{1}{100}} = \frac{1}{99+s_{100}}$ və s.

$s_{100}, s_{99}, s_{98}, \dots, s_1$ ədədləri ardıcıllığın elementləridir. Bu ardıcıllığın **n** nömrəli elementin rekurent düsturu aşağıdakı kimidir:

$$a_n = \frac{1}{101 - n + a_{n-1}}$$

Bu alqoritmin C dilində reallaşdırılması aşağıda göstərilmişdir:

```
float s = 0;
for ( int k = 100; k>=1; k -- )
s = 1/( k + s );
```

Sıralar

Sonsuz sayda elementlərdən ibarət ardıcıllığın cəmi **sıra** adlanır.

Əlbəttə, reallıqda sonsuz sayda elementləri cəmləmək mümkün deyil, buna görə də sonsuz cəmi **sonlu cəm** ilə əvəz edirlər. Bu cəmə birinci **k** element daxildir.

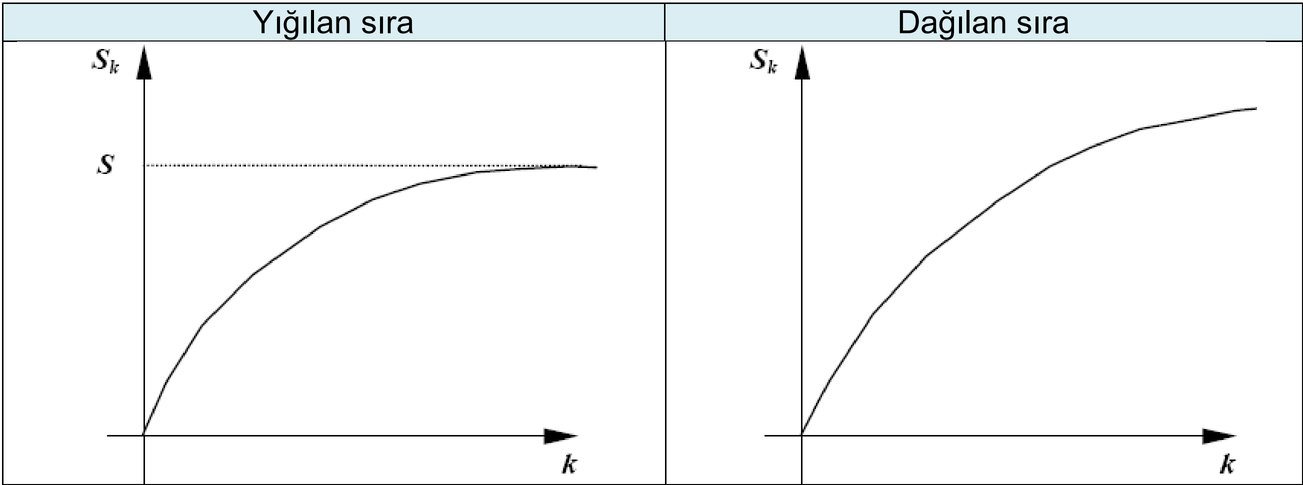
$$S = \sum_{1}^{\infty} a_n \approx S_k = \sum_{1}^{k} a_n$$

Sıraların iki növü mövcuddur: **dağılan ardıcıllıq** və **yığılan ardıcıllıq**. Dağılan ardıcıllıqlar üçün sonlu cəm - S_k modula görə qabağcadan verilmiş ədəddən böyük olur. Dağılan ardıcıllığın limiti olmur. Dağılan ardıcıllığın nümunəsi kimi **harmonik** sıranı göstərmək olar:

$$S = 1 + 2 + 3 + 4 + \cdots$$

Sonlu limiti olan ədədi ardıcıllığa yığılan ardıcıllıq deyilir, yəni elementlərin sayı (**k**) artdıqca sonlu cəm müəyyən sonlu qiymətə (limitə) yaxınlaşır.

Yığılan sıra	Dağılan sıra



Yığılan sıralar üçün, adətən, verilmiş ε dəqiqliyi ilə sıranın cəmini hesablamaq olur, yəni

$$|S - S_k| = \left|\sum_{k+1}^{\infty} a_n\right| < \varepsilon$$

Ümumiyyətlə, bu məsələ mürəkkəb məsələdir, lakin müəyyən görünüşə malik olan sıralar üçün onu asanlıqla həll etmək olar.

Əgər sırada işarələr dəyişirsə və onun elementləri azalılar, onda o, yığılır. Cəmin hesablanmasında buraxılan xəta axırıncı atılan elementin modulundan böyük olmur.

İşarələri dəyişən sırada elementlərin işarələri növbələnilər, məsələn:

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \cdots + \frac{(-1)^{n+1}}{n} + \cdots$$

Funksiyaların hesablanması

Kompüterdə riyazi funksiyaların çoxu dəyişəndən asılı olan **funksional sıralar** vasitəsi ilə hesablanır. Bu isə sinuslar cədvəlini kompüterin yaddaşında saxlamamasına və istənilən bucaq üçün bilavasitə sinusun hesablanmasına imkan yaradır. Həmişəki kimi, yaddaşın qənaəti sürətin azaldılmasına gətirib çıxardır. Aşağıda ən məşhur olan funksional sıralar göstərilmişdir:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots + \frac{x^n}{n!} + \cdots$$

$$sinx = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + \frac{(-1)^{n+1}x^{2n-1}}{(2n-1)!} + \cdots$$

$$cosx = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots + \frac{(-1)^{n+1}x^{2n}}{(2n)!} + \cdots$$

Misal üçün, **sin(x)** funksiyasının 10^{-6} dəqiqliyi ilə hesablanması üçün funksiya tərtib edək. Belə ki, sin funksiyasının sırası işarə dəyişəndir və sıra elementlərinin qiymətləri azalır, onda mütləq qiymətləri (modulları) 10^{-6}-dan böyük olan bütün elementləri cəmləmək lazımdır. İşarə dəyişkinliyini nəzərə almaq üçün 1 və ya -1 qiyməti alan dəyişəndən istifadə etmək lazımdır.

```
double sin1 (double x )
{
double s = 0, a, xx = x,
    fact  =1, n2 = 1;
int z=1;
do {
    a = z * xx / fact;
    s +=a;
    z = -z;
    xx *= x*x;
    n2 +=2;
    fact  *=(n2-1)*n2;
    }
while ( fabs(a) > 1.e-6 );
return s;
}
```

s – sıranın sonlu cəmidir
a – sıranın elementidir
xx – x^{2n-1}
fact – (2n-1)!

Aşağıdakı cədvəldə müxtəlif bucaqlar üçün 10^{-6} dəqiqliyi ilə sinusun hesablanmasında istifadə olunan sıra elementlərinin sayı göstərilmişdir.

bucaqlar diapazonu (modula görə)	tələb olunan sıra elementlərinin sayı
$0^0 - 90^0$	1 – 7
$90^0 - 180^0$	7 - 9
$180^0 - 270^0$	10 – 12
$270^0 - 360^0$	12 – 14

İstənilən bucağın sinusunu $0^0 - 90^0$ intervalında dəyişən sinus vasitəsi ilə hesablamaq olar:

$$sin\alpha = \sin(180^0 - \alpha) = -\sin(-\alpha) = -\sin(\alpha - 180^0)$$

Beləliklə, istənilən halda sinusun hesablanması üçün 7-dən çox olmayan sıra üzvlərindən istifadə etmək lazımdır.

Tənliklərin ədədi həlli

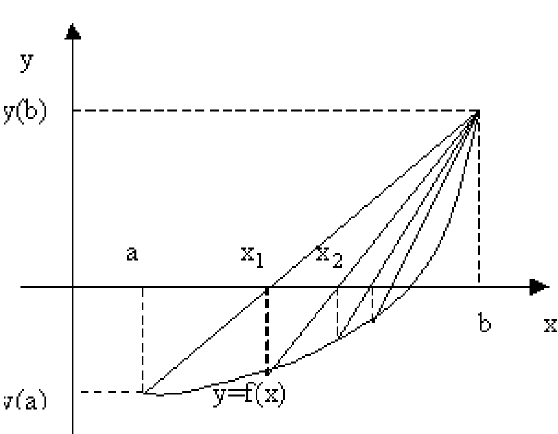
Tənliklərin çoxunu analitik üsullarla həll etmək olmur. Belə hallarda **təqribi ədədi üsullardan** istifadə edirlər. Bu üsullar təqribidir, çünki tənliyin dəqiq həlli – x^* olmadığı üçün ona yaxın olan və dəqiq qiymətdən ε qədər fərqlənən təqribi həll (kök) təyin edilir.

$\boldsymbol{f(x) = 0}$ tənliyi nəzərdən keçirək. Burada ***f(x)*** – bir dəyişənli funksiyadır. Tənliklərin həlli üçün istifadə edilən ədədi üsullardan birinə - **parçanın yarı bölünmə** və ya

dixotomiya üsuluna artıq baxmışıq. Bu üsulun üstünlüyü ondan ibarətdir ki, əmin olmaq olar ki, xəta verilmiş qiymətdən çox olmur. Digər tərəfdən, bu üsuldan istifadə etmək üçün bir və yalnız bir kökü özündə saxlayan intervalı qabaqcadan təyin etmək lazımdır.

Xordalar üsulu

Dixotomiya üsulu kimi xordalar üsulu **[a,b]** intervalında tənliyin kökünü təyin edir. Əsas şərtlərdən biri həmin parçada kökün olmasıdır. Parçanın qıraqlarında kəsilməz funksiyanın işarələri fərqli olmalıdır.



Kökün növbəti yaxınlaşması - **x [a,b]** parçasının ortasına yox (dixotomiya üsulu), xordanın **((a,f(a))** və **(b, f(b))** nöqtələri birləşdirən xətt) [a,b] parçası ilə kəsişməsinə bərabərdir. İki nöqtədən keçən xordanın tənliyi aşağıda göstərilmişdir:

$$\frac{x-a}{b-a}=\frac{y-f(a)}{f(b)-f(a)}$$

Kəsişmə nöqtəsində y=0. Tənlikdə bunu nəzərə alaraq alırıq:

$$x_1=a-\frac{(b-a)f(a)}{f(b)-f(a)}$$

Sonra isə dixotomiya üsuldakı kimi $[a,x_1]$ və $[x_1,b]$ intervalları yoxlanılmalıdır, yəni kəsişmə olduğu intervalı təyin edirik və ona uyğun **a** və ya **b** nöqtəsini dəyişirik.

İterasiyaları nə vaxt bitirmək lazımdır? Adətən iki meyardan biri qəbul olunur:

1. ardıcıl yanaşmalar arasında olan fərq x_k-x_{k-1} verilmiş ε dəqiqliyindən kiçik olmalıdır;
2. $|f(x_k)|<\varepsilon_1$ (x_k nöqtəsində olan funksiyanın qiyməti verilmiş ε_1 qiymətindən kiçik olmalıdır).

Alqoritmin xətası funksiyanın özündən asılıdır.

```
float Chord ( float a, float b, float eps, float eps1)
{
  float x, x0=a, fa, fb, fx;
while (1)  {
    fa = F(a); fb = F(b);
    x = a – (b-a)*fa / (fb-fa);       // növbəti yaxınlaşma
    fx = F(x);
    if ( fabs(fx) < eps1 ) break;   // 2-ci şərtin yoxlanılması
    if (fa*fx < 0 ) b =x;               // axtarış intervalını daraldırıq
    else          a =x;
    if ( fabs(x-x0) < eps ) break;   // 1-ci şərtin yoxlanılması
    x0 = x;
    }
return x;
```

```
}
```

eps və **eps1** parametrləri **x** və həmin nöqtədə funksiyanın dəqiqliyini ifadə edirlər. Bu funksiyanı daha da optimallaşdırmaq olardı, çünki f(a) və f(b) qiymətlərdən biri əvvəlki addımdan məlumdur.

Nyuton üsulu (toxunanlar üsulu)

Təcrübədə ən çox istifadə olunan üsul – Nyuton üsuludur, çünki o, tez yığılandır və kökün tapılması üçün **x**-in dəyişmə intervalını yox, onun ilkin yaxınlaşmasını (**x_0**) tələb edir.

Kökün növbəti yaxınlaşması kimi əyridən çəkilən toxunanın OX oxu ilə kəsişməsi olan x-in qiyməti qəbul edilir. Toxunan (x_0, f(x_0)) nöqtəsindən çəkilir.

$$tg\alpha = f'(x_0) = \frac{f(x_0)}{x_0 - x_1}$$

İterasiyanın k addımında alırıq:

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

Nyuton üsulu yüksək yığılma sürətinə malikdir. 10^{-6} dəqiqliyinə 5-6 iterasiyaya çatmaq olur. Onun çatışmayan cəhəti ondan ibarətdir ki, hər addımda törəmə hesablanmalıdır, lakin elə ola bilər ki törəmənin ifadəsi məlum olmasın (törəmə cədvəl şəkilində verilir). Bəzən modifikasiya olunmuş Nyuton üsulundan istifadə edirlər. Bu üsulda bütün addımlarda funksiyanın törəməsi **x_0** nöqtəsində hesablanır. x-in başlanğıc qiymətinin seçilməsi də mühüm məsələdir, çünki o, düz seçilməsə Nyuton üsulu heç vaxt yığılmayacaq, yəni proses sonsuz davam edəcəkdir.

Aşağıda göstərilmiş funksiya iki əlavə funksiyaya müraciət edir – **F(x)** (**x** nöqtəsində funksiyanın qiymətini hesablayır) və **DF(x)** (**x** nöqtəsində törəmənin qiymətini hesablayır).

```
float Newton ( float x0, float eps)
{
   float f1;
   do   {
          f1  = F(x0)/DF(x0);
          x0 -=f1;
          }
   while ( fabs(f1) > eps );
  return x0;
}
```

Nyuton üsulundan qeyri xətti tənliklər sisteminin həlli üçün də istifadə olunur. Burada dəyişənlərin sayı çoxdur və düsturlar daha mürəkkəb olur.

İterasiyalar üsulu

f(x)=0 tənliyindən aşağıdakı tənliyə keçmək olar:

$$x + bf(x) = x$$

Burada b – sıfıra bərabər olmayan sabitdir. $\varphi(x) = x + bf(x)$ işarə etsək, onda aşağıdakı tənliyi alırıq:

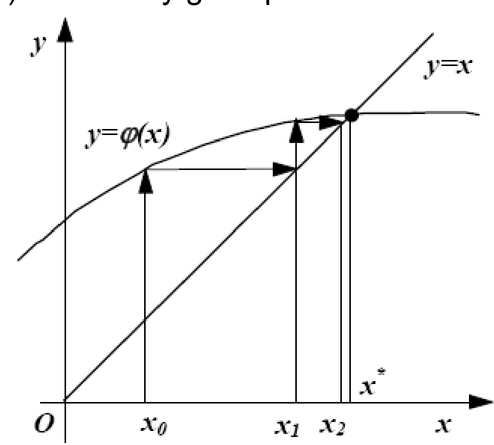
$$x = \varphi(x)$$

Bu tənlik əsasında iterasiyalı düsturu almaq olar:

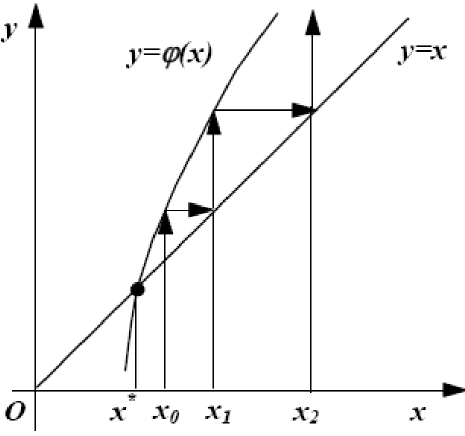
$$x_k = \varphi(x_{k-1})$$

İndi isə $\mathbf{x}^*$ qiymətinin alınması üçün (proses yığılır) prosedura hansı şərtlərə uyğun olmalıdır təyin edək. 4 variantı nəzərdən keçirək:

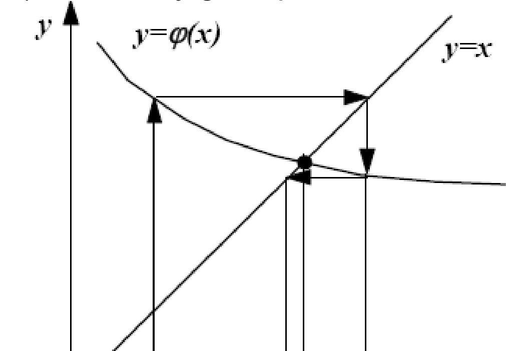
a) birtərəfli yığılan proses



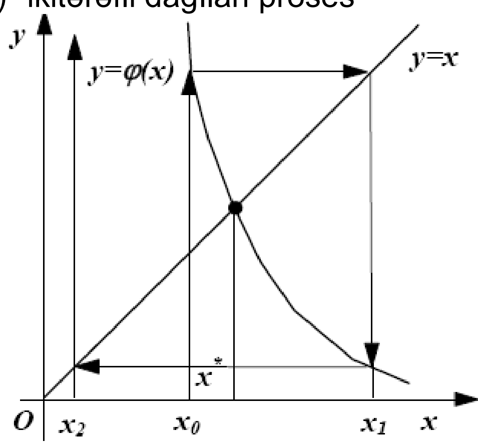
b) birtərəfli dağılan proses



c) ikitərəfli yığılan proses



d) ikitərəfli dağılan proses



Göstərmək olar ki, əgər aşağıdakı şərt ödənilirsə, onda proses yığılır:

$$|\varphi'(x)| < 1,$$

özü də, əgər $0 < \varphi'(x) < 1$ şərti ödənilirsə, onda yığılma birtərəflidir, $-1 < \varphi'(x) < 0$ olduğu halda yığılma ikitərəflidir. Nəzərə alaraq ki, $\varphi(x) = x + bf(x)$, aşağıdakı ifadəni almaq olar:

$$\varphi'(x) = 1 + bf'(x)$$

Ən sürətli yığılma $\varphi'(x) = 0$ olduğu halda baş verir. Bu hal üçün alırıq:

$$b = -\frac{1}{f'(x)}$$

Bu isə Nyuton düsturu deməkdir. Beləliklə, bütün iterasiya üsulları arasında Nyuton üsulu ən yüksək yığılma sürətinə malikdir.

Belə ki, ümumi halda prosesin yığılmasına heç bir zəmanət vermək olmaz (bu üsulun əsas çatışmayan cəhətlərindən biridir), proqramda iterasiyaların sayını müəyyən qiymətlə məhdudlaşdırmaq lazımdır. İterasiyalar üsulunu reallaşdıran funksiya aşağıdakı cədvəldə göstərilmişdir. Funksiya o vaxt nəticə verir ki, nə vaxt iki ardıcıl yaxınlaşma arasında olan fərq verilmiş ε dəqiqliyindən kiçik olsun, yaxud da iterasiyaların sayı maksimum həddə (**n**) çatmışdır. Prosesin dağılması haqqında informasiyanı əsas proqrama ötürmək üçün **n** parametri dəyişən götürülüb. Funksiyanın çıxışında bu parametr iterasiyaların sayına bərabərdir. Əgər **n** verilmiş qiyməti keçirsə, onda nəticə düz deyil və **b** sabiti dəyişilməlidir.

```
float Iter (float x0, float eps, int &n)
{
   int i=0;
   float x=x0;
   do  {
        x0=x;          // x-in əvvəlki qiymətini yadda saxla
        x = F(x0);     // iterasiya addımı
        i ++;
        if ( i > n ) break;    //  proses dağılır, çıxış
       }
   while ( fabs(x-x0) > eps );   // hələ ki, həll tapılmayıb
  n = i;
  return x;
}
```

Funksiyanın çağırışı aşağıdakı kimi ola bilər:

```
int n = 100;
float x;
...
x = Iter(1.2, 0.0001, n );
if ( n > 100)
    printf ( " Proses dagilir ");
```

Funksiyanın parametr əvəzi istifadəsi

Hesablamalarda istifadə edilən funksiyanı proseduraların parametrlərinə daxil etmək olar. Əvvəlcə verilənlərin yeni tipini, yəni funksiyanı elan etmək lazımdır. Funksiya bir həqiqi qiymət alıb həqiqi qiymət qaytarır.

```
typedef float (*func) (float x );
```

Sonra isə bu **func** tipini funksiyaların parametrlərində istifadə etmək olar. Məsələn, iterasiya üsulu üçün funksiyanı aşağıdakı kimi modifikasiya etmək olar:

```
float Iter ( func F, float x0, float eps, int &n)
```

```
{
...
}
```

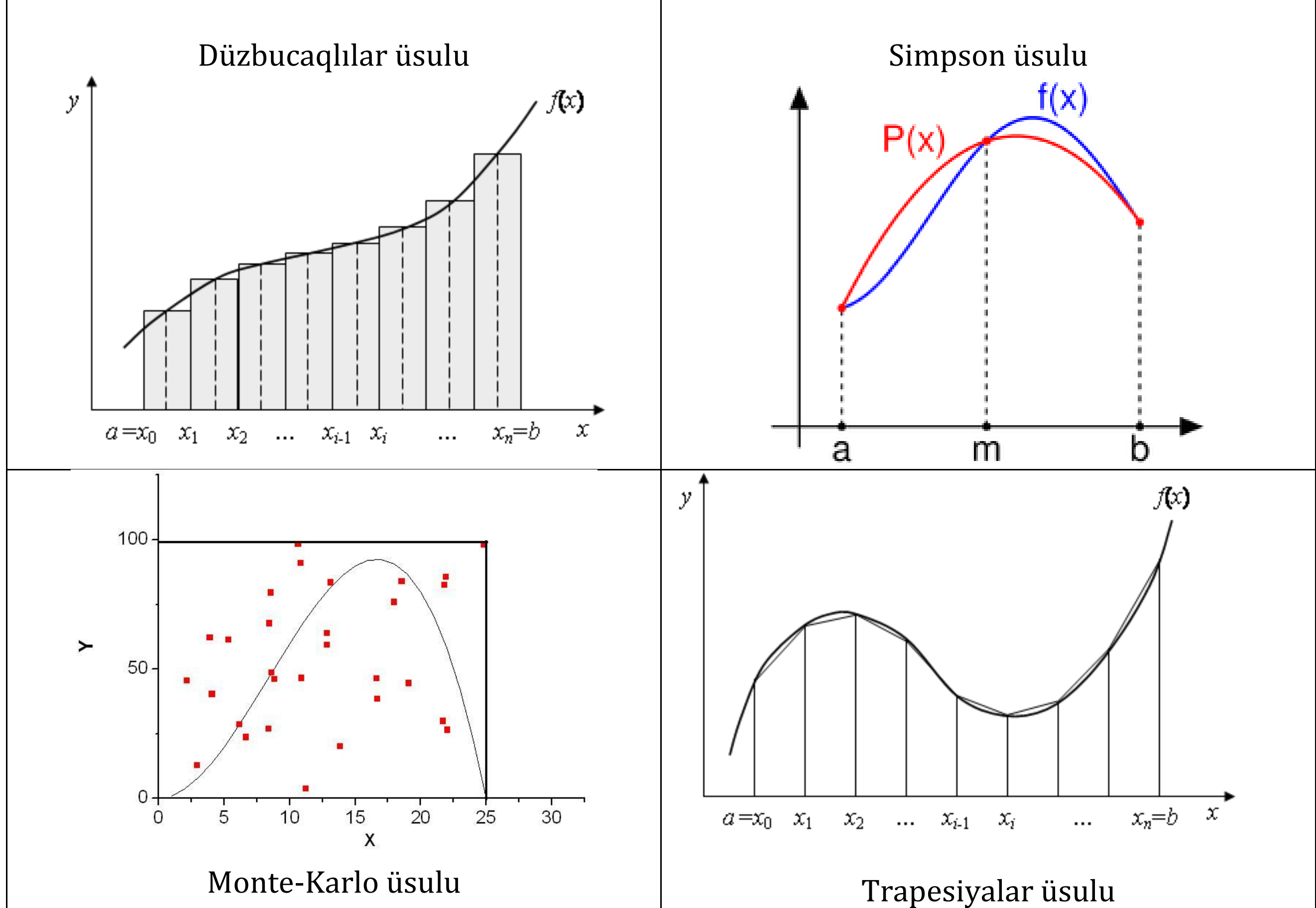
Funksiyanın gövdəsi dəyişilməz olaraq qalır. Funksiyanın çağırışı zamanı birinci parametr kimi müəyyən düstur əsasında hesablamaları həyata keçirən funksiyanın adını ötürmək lazımdır.

Müəyyən inteqralların hesablanması

Analitik halda ibtidai funksiyası olmayan və ya çox mürəkkəb olan aşağıdakı müəyyən inteqralı sonlu intervalda hesablayaq:

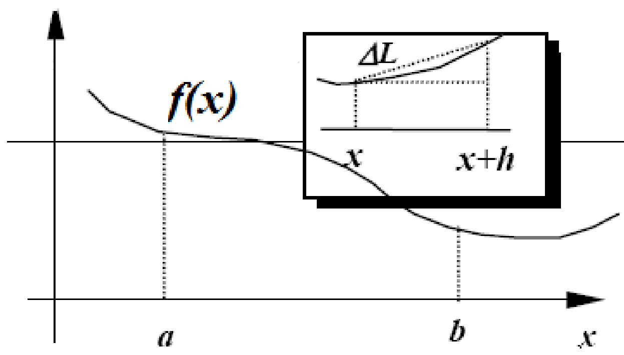
$$J = \int_a^b f(x)dx$$

Riyazi baxımından inteqral $y = f(x)$ əyri altında olan fiqurun sahəsinə bərabərdir. Buna görə də inteqralın hesablanması üçün sahələrin hesablanmasında istifadə olunan bütün üsulları tətbiq etmək olar (düzbucaqlılar, trapesiyalar, Monte-Karlo, Simpson üsulu).



Əyrinin uzunluğunun hesablanması

Tutaq ki, bir dəyişənli $y = f(x)$ funksiya verilmişdir. **[a,b]** intervalında funksiyanın uzunluğunu (**L**) tapmağa tələb olunur.



Ali riyaziyyatda bu məsələni müəyyən inteqral vasitəsi ilə həll edirlər.

$$L = \int_a^b \sqrt{1 + [f'(x)]^2}dx$$

Lakin bu inteqralı analitik şəkildə açmaq olmur və buna görə də təqribi üsullardan istifadə olunur. Əyrinin $[x, x+h]$ hissəsinə baxaq. Burada h – kiçik bir addımdır. Təqribən hesab etmək olar ki, əyrinin bu hissəsinin uzunluğu təqribən iki nöqtəni birləşdirən parçanın uzunluğuna bərabərdir. Pifaqor teoreminə görə aşağıdakını alırıq:

$$\Delta L = \sqrt{h^2 + [f(x+h) - f(x)]^2}$$

Əyrinin [a,b] parçasında tam uzunluğunu almaq üçün kiçik parçaların uzunluqları toplanmalıdır. Bunu həyata keçirən funksiya aşağıda göstərilmişdir:

```
float CurveLength ( func F, float a, float b )
{
float x, dy, h = 0.0001, h2 = h*h, L = 0;
for ( x = a; x < b; x +=h)
   {
    dy = F(x+h) – F(x);
   L += sqrt ( h2 + dy*dy);
   }
return L;
}
```