

QAFQAZ
UNİVERSİTETİ



Borland
C++
ilə
Obyektyönlü
Proqramlaşdırma

Azərbaycan Respublikası Təhsil
Nazirliyinin 20.10.2006-cı il tarixli, 760
saylı əmri ilə dərs vəsaiti kimi tövsiyə
edilmişdir.

Etibar Seyidzadə

BAKİ - 2007

Elmi redaktor : t.e.n., dos. Xəlil İsmayilov
("Qafqaz" Universiteti)

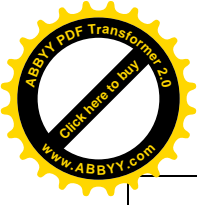
Rəyçilər : f.-r.e.d., prof. Fəxrəddin İsayev
("Qafqaz" Universiteti)
t.e.n., Abzətdin Adamov
("Qafqaz" Universiteti)

Korrektor : Vəfa Seyidova
Dizayner : Sahib Kazımov

Seyidzadə Etibar Vəqif oğlu

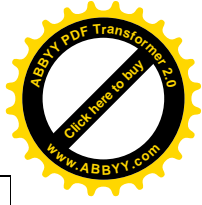
Borland C++ ilə
Obyektyönlü Proqramlaşdırma

© Seyidzadə E.V. 2007

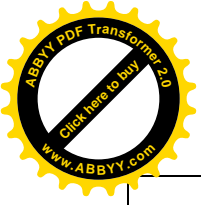


MÜNDƏRİCAT

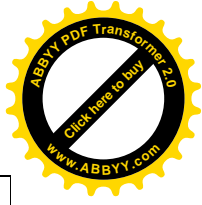
I FƏSİL	9
OBJEKTİYÖNLÜ PROQRAMLAŞDIRMA	9
1.1 Proqram Layihələndirmə	9
1.2 Proqram Xüsusiyyətləri	10
1.3 Modul Strukturunun Şərtləri	14
1.4 Obyektyönlü Proqramlaşdırmanın Əsasları	15
II FƏSİL	25
C-DƏ YENİLİKLƏR VƏ C-YƏ ƏLAVƏLƏR	25
2.1 Eyni Adlı Müxtəlif Arqumentli Funksiyalar	25
2.2 Operatorların Təyini	31
2.3 Aktiv Qiymət Vermək	33
2.4 Təqdimat (Referans) Tip Təyinedicisi	37
2.5 Gizlənmiş Dəyişənləri Görmək	41
2.6 C++-da Prototiplərin Təyin Edilməsi	43
2.7 Struktur Tiplər	44
2.8 Şərh Operatoru	45
2.9 new və delete Opratorları	46
2.10 inline Makroları	49
III FƏSİL	51
OBJEKTlər	51
3.1 Obyekt Nədir?	51
3.2 Layihələndirici	55
3.3 Müraciət Haqqı	57



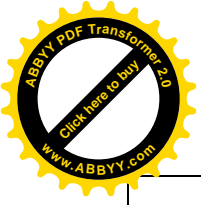
3.4 Yoxedicici (Destructor)	62
3.5 Standart Obyekt Tipləri	67
3.6 Layihələndirici Üzərinə Yükləmə	67
3.7 Obyektlərə Mənimətmə	74
IV FƏSİL	85
OBJEKTləRİN XÜSUSİYYƏTLƏRİ	85
4.1 Obyekt Üzvləri Olan Obyektlər	85
4.2 Friend (Dost) Təyinedicisi	90
4.3 Obyektlərin Operatorlara Yüklənməsi	97
4.4 this Lokal Dəyişkəni	101
4.5 Ümumi Ortaq Dəyişkənlər	103
4.6 Statik (Static) Funksiyalar	107
4.7 const Funksiyaları	112
4.8 İç-içə Təyinlər	112
4.9 Obyekt Göstəriciləri	115
4.10 Obyekt Massivi	117
V FƏSİL	121
OBJEKT TÖRƏTMƏK	121
5.1. Törətmə Əməliyyatı	121
5.2 Siniflərin Törədilməsi	122
5.3 Müraciət Haqları və Nüfuz Etmə	127
5.4 Dinamik Yükləmə	129
5.5 Qaydalı Funksiyalar	133
5.6 Misallar	138
5.6.1 Curve	138
5.6.2 LineDemo	143



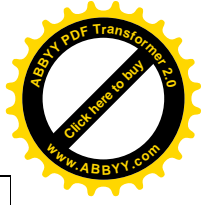
5.7 C++ Metod Çağırış Sistemi.....	161
5.8 Mövcud Olandan Törənən Siniflər	164
VI FƏSİL.....	169
ŞABLONLAR HAZIRLAMAQ.....	169
6.1 Şablonlar	169
6.2 Şablon Funksiyalar	172
6.3 Şablon Obyektlər	176
VII FƏSİL.....	181
AXINLAR	181
7.1 Axın Nədir?.....	181
7.2 Standart Axınlar.....	181
7.3 Axınlara Nizamlanmış Məlumat Yazılması	185
7.3.1 Genişlik Nəzarəti	185
7.3.2 Yerləşmə Nəzarəti.....	187
7.3.3 Boşluq Nəzarəti	188
7.3.4 Tam Ədədlərin Əsaslarına Nəzarət.....	189
7.3.5 Həqiqi Ədədə Nəzarət	190
7.4 Axınlardan Nizamlanmış Məlumat Oxunması.....	192
7.5 Səhvlərə Nəzarət.....	195
7.6 Fayl Üzərindəki Axınlar.....	195
7.6.1 Fayla Yazma	196
7.6.2 Fayldan Oxuma.....	200
7.7 Obyektlər və Axınlar	202
VIII FƏSİL.....	211
CLASS KİTABXANASI	211



8.1 Container Class Kitabxanası.....	211
8.2 Təyin Olunmuş Siniflər	212
8.3 Təyinlər və Tiplər	214
8.3.1 Tip və Sinif Kodları.....	214
8.3.2 Səhv Kodlarının Təyini.....	216
8.3.3 Başlıq Faylları və Təyin Edilmiş Siniflər	217
8.4 Siniflər.....	218
8.4.1 Object	218
8.4.2 Error	227
8.4.3 Sortable	228
8.4.4 String.....	229
8.4.5 BaseDate	232
8.4.6 Date	235
8.4.7 BaseTime.....	239
8.4.8 Time	242
8.4.9 Association	244
8.5 Məlumatlar Sturukturu Sinifləri.....	247
8.5.1 Container	247
8.5.2 Stack	251
8.5.3 Deque.....	255
8.5.4 Queue.....	258
8.5.5 PriorityQueue.....	259
8.5.6 Collection.....	261
8.5.7 List.....	263
8.5.8 DoubleList	265
8.5.9 HashTable.....	268
8.5.10 Btree	270
8.5.11 Bag.....	277



8.5.12 Set	278
8.5.13 Dictionary	278
8.5.14 AbstractArray	279
8.5.15 Array	283
8.5.16 SortedArray	285
8.6 Yeniləyicilər (Iterators).....	290
8.6.1 DoubleListIterator.....	293
8.7 Misal	294



Ö N S Ö Z

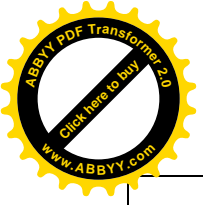
Bu kitabda obyekt yönü proqramlaşdırmanın əsasları, Borland C++ proqramlaşdırma dilinin xüsusiyyətləri şərh edilmişdir.

Vəsaitdən obyekt yönü proqramlaşdırmağı öyrənmək istəyən tələbələr, müəllimlər, həmçinin proqramçılar faydalana bilərlər. Vəsait mümkün qədər sadə dildə yazılmışdır. Kitabdakı mövzular misallarla müşayiət olunmuşdur ki, bu da mövzunu asan mənimsəməyə kömək edir.

Onu da qeyd edək ki, kitab respublikamızda bu mövzuda azərbaycan dilində yazılmış ilk vəsaitdir. Buna görə də kitabda bir çox yeni terminlərin istifadəsində çətinliklər qarşıya çıxmış və bu çətinliklərin aradan qaldırılmasına cəhd göstərilmişdir.

Kitabın hazırlanmasında lazımi şərait yaratdığına görə Qafqaz Universitetinin rəhbərliyinə, dəyərli məsləhətlərinə görə f.r.e.d., professor Fəxrəddin İsayevə, t.e.n., dosent Xəlil İsmayılova, t.e.n. Abzətdin Adamova təşhif etdiyinə görə Vəfa Seyidovaya, dizayner Sahib Kazımovaya, kitabın çapında göstərdikləri dəstəyə görə Müşfiq İbrahimova və Sədi Əbdürrəhmanova öz səmimi təşəkkürümü bildirirəm.

Müəllif



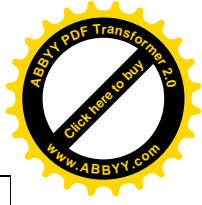
I FƏSİL

OBYEKTİYÖNLÜ PROQRAMLASHDIRMA

1.1 Proqram Layihələndirmə

Kompüter texnologiyasının paralel inkişaf edən iki əsas sahəsi vardır: texniki vasitələr (**hardware**) və proqram təminatı (**software**). Texniki vasitələr nə qədər sürətlə inkişaf etsə də, proqram təminatı ilə təchiz edilmədən heç bir faydası olmaz. Proqramın texniki vasitələr olmadan işləməsini də düşünmək olmaz. Lakin yaxşı texniki vasitə olmazsa, uyğun olaraq yaxşı proqram da yazılmaz. İstifadəçilərin istədiklərini texniki vasitələrlə yerinə yetirmələri üçün isə hər zaman yaxşı proqram təminatına ehtiyac vardır.

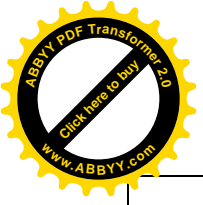
Proqram təminatı çox əhəmiyyətli olan, lakin bunun qiyməti çox gec başa düşülən bir mövzu olmuşdur. Yaxın keçmişdə kompüter ehtiyacı olan şəxslər işlərini görə biləcək proqram, sonra da bu proqramı işlədəcək bir kompüter axtarır və proqramı kompüterin ayrılmaz bir xüsusiyyəti kimi görürdülər. Bu bir növ kompüter satıcılarının ehtiyac duyulan proqramları özlərinin



yazmalarından və öz kompüterləri xaricində bu proqramı satmamalarından qaynaqlanırdı. Hazırda isə müstəqil proqramlaşdırma ilə məşğul olan firmaların qurulması bu səhv düşüncəni aradan qaldırmışdır. Bu həmçinin proqram təminatı və texniki vasitələrin bir-birindən fərqli bir şey olduqları həqiqətinin anlaşılmasını təmin etmişdir.

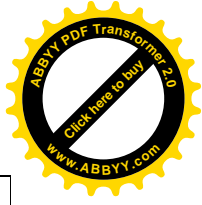
1.2 Proqram Xüsusiyyətləri

- Doğruluq (**correctness**) – verilən tapşırıqların tam olaraq yerinə yetirilməsidir. Proqramı layihələndirmədən əvvəl onun hansı tapşırıqları yerinə yetirəcəyini müəyyən etmək lazımdır. Proqram hazır olduqdan sonra bu təyin olunan xüsusiyyəti tam təmin etməlidir;
- Dayanıqlıq (**robustness**) – gözlənilməz hadisələr nəticəsində proqramın icrası kəsilməməli, səhv əməliyyatları yerinə yetirməməlidir. Proqram, ən yaxşı halda olsa belə, üzərinə qoyulan tapşırıqlardan başqa işləri görməməlidir. Proqramçının səhvlərinə görə proqramın icrasının kəsilməməsi üçün tədbirlər görülməlidir;
- Genişlənəbilmək (**extendibility**) – gələcəkdə verilən tapşırıqların dəyişdirilməsi və ya

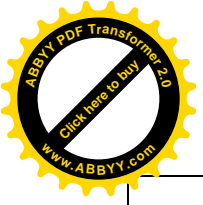


yenilərinin əlavə edilməsi asan olmalıdır. *Bunun üçün:*

- sadə layihələr hazırlanaraq mürəkkəb layihələrdən qaçmaq lazımdır (**design simplicity**);
- proqramı bir mərkəzdən asılı olaraq idarə etmək əvəzinə modul strukturundan istifadə edərək yerli bir idarə etmə forması seçilməlidir (**decentralization**).
- Təkrar istifadə olunma (**reusability**) – hazırlanan layihənin, yazılan proqramın və ya heç olmazsa modulların başqa proqramlar tərəfindən istifadə edilə bilməsidir. Buna layihə daxilində istifadə edilən elementlərin yeni layihədə də istifadə edilə bilməsini əlavə etmək lazımdır;
- Uyğunluq (**compatibility**) – proqramın müxtəlif kompüter sistemlərində ortaq xüsusiyyətlərə malik olmasıdır. *Bunun üçün müxtəlif standartların tətbiq edilməsi lazımdır:*
 - məlumatlar faylı formatının uyğunluğu;
 - məlumatlar strukturunun uyğunluğu.
- Menyü, dialoq, rəsm, düymə kimi istifadəçi mühitinin (**user interface**) uyğunluğu;



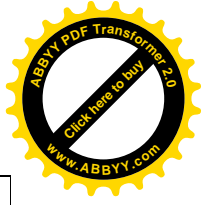
- Mənbələrin istifadə edilməsi (**efficiency**) – kompüterin malik olduğu bütün avadanlıqları səmərəli şəkildə tam istifadə etməsidir. İstifadə edə bilmədikdə də digər proqramlar üçün istifadəsiz qalmasına yol verməsidir;
- Daşınabilmə (**portability**) – bir proqram hazırlanmış olduğu kompüterdən başqa digər kompüterlərdə də istifadə oluna bilməlidir. Bu iki formada ola bilər:
 - Qaynaq uyğunluğu (**source compatible**) – proqramın yazıldığı əməliyyat sistemindən başqa bir sistemə daşınıb yenidən kompilyasiya olunaraq işləməsi;
 - İkilik kod uyğunluğu (**binary compatible**) – proqramın yazıldığı mühitdə kompilyasiya olunaraq icra oluna bilən fayl (**executable file**) əldə edildikdən sonra başqa bir mühitə daşınaraq işlədilməsi. Proqramın təkmilləşdirilməsi baxımından əsl əsas götürülərək proqram kodunun daşınabilən olmasıdır.
- Nəzarət oluna bilmə (**verifiability**) – bir proqramın səhv hallarla qarşılaşması zamanı onun icrasının davam etməsinə və hətta heç icra olunmamasına səbəb olan səhvlər ortaya çıxdığı zaman istifadəçiyə və proqramçıya səhvin hansı



səbəbdən baş verdiyi mövzusunda kifayət qədər məlumatın verilməsidir;

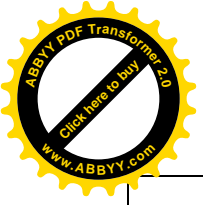
- Tamlıq (**integrity**) – icazəsiz müdaxilələr və dəyişdirmələr qarşısında proqram, məlumatlar, fayl kimi proqram komponentlərinin qorunmasıdır. Məsələn, məlumatlar və ya indeks fayllarının itməsi (silinməsi) zamanı proqram icra olunarkən bunu müəyyənləşdirib bildirir;
- Asan istifadə edilmə (**easy of use**) – proqramdan istifadə edən şəxsin onu asanlıqla öyrənməsi, istifadə edə bilməsi, nəticələrini tədqiq edə bilməsi, səhvlərini düzəldə bilməsidir;
- Birlikdə işləmək (**interoperability**) – bir proqramın ehtiyacı olduğu başqa bir proqramı çağırma bilməsi xüsusiyyətidir. Bu halda iki proqram ardıcıl olaraq işləməklə bərabər bir-biri ilə məlumat mübadiləsi edə bilməlidirlər.

Yuxarıda göstərilən şərtləri təmin etmək əsasən proqramçı mühəndisin vəzifəsi olmaqla bərabər, modul strukturundan istifadə etmək *genişlənmə bilmə, təkrar istifadə olunma, uyğunluq, daşıma bilmə* problemlərini həll etməyə imkan verir. Bu halda ən azı bəzi proqram modullarını yenidən yazmağa ehtiyac qalmır.



1.3 Modul Strukturunun Şərtləri

- Parçalanabilmə (**modular decomposability**) – bir problemi alt hissələrə ayıraraq layihələndirməkdir. Məsələn, riyazi əməliyyatların yerinə yetirildiyi bir proqramda massivlərin istifadə edilməsi üçün bir massiv modulu təyin edərək massivlə əlaqədar əməliyyatların hamısını bu modulda yazmaq. Eyni şəkildə ehtiyac olarsa, matris, vektor, kompleks ədəd kimi təyinlər üçün də modul yazaraq problemi kiçik hissələrə ayırmaq;
- Birləşdiriləbilmə (**modular composability**) – bir-birindən xəbərsiz hazırlanan modulların bir yerə yığılması zamanı çatışmayan və ya tam olmayan modulların olmamasıdır;
- Aydınlıq (**modular understandability**) – müxtəlif şəxslər tərəfindən yazılmasına baxmayaraq oxunduğu zaman proqramın aydın olmasıdır. Proqramın yenidən baxılması və ya təkmilləşdirilməsi zamanı çox əhəmiyyətli olan bu xüsusiyyəti saxlamaq üçün modul proqramının yazılmasından başqa proqram daxilində nəyin nə üçün istifadə edildiyinin, bu istifadə nəticəsində nə olacağının aydın bir şəkildə şərh olunması lazımdır;

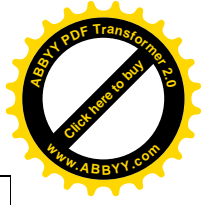


- Qorunma (**modular protection**) – bir modulun işləmə formasına və məlumatlara digər bir modulun icazə verilmiş hallardan başqa müdaxilə etməməsi, yazılan modulların ümumi cəhətləri olmasına baxmayaraq bunların bir-birindən fərqləndirilməsi lazımdır;
- Davamlılıq (**modular continuity**) – problemin təyinindəki kiçik dəyişikliklər bir və ya bir neçə hissənin dəyişməsinə səbəb olarkən proqram strukturu kimi istifadə olunan modullar arasındakı vasitələr (məsələn, funksiya prototipləri) dəyişdirilməməlidir.

İndi də modulluluğun təmin olunması üçün istifadə olunacaq obyekt yönü proqramlaşdırmanın xüsusiyyətlərini gözdən keçirək.

1.4 Obyekt yönü Proqramlaşdırmanın Əsasları

Bir sistem daxilində müxtəlif xarakterli obyektlər ola bilər. Bu obyektlərin tamamilə bir-birindən fərqli xüsusiyyətləri ilə bərabər, eyni və ya oxşar xüsusiyyətləri də vardır. Xüsusiyyətlər və davranışlarının müxtəlif olmalarına baxmayaraq eyni imkanlara malik ola bilərlər.

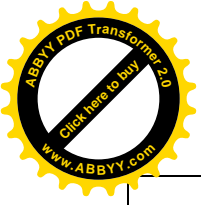


Məsələn, bir idarədə kompüter, telefon, faks, katibə, məmur, müdir kimi obyektlər ola bilər. Əsasən bunların hər biri bir obyektidir. Hər birinin öz funksiyası vardır. Lakin ümumi xüsusiyyətləri də vardır. Katibə, məmur, müdir hər biri bir insandır. Bu insan olma xüsusiyyətidir. Hər biri müəssisədə müxtəlif məbləğdə maaşla işləyirlər. Gördükləri iş ümumi bir işdir, lakin hər birinin öz işi vardır. İstək eyni, davranışlar isə müxtəlifdir. İşlərini icra edərkən istifadə etdikləri məlumatlar da eyni dərəcədə fərqlidir.

Bu bənzətmə ilə *obyekt - müəyyən işləri yerinə yetirən, bu məqsədlə də müxtəlif funksiyalardan ibarət olan bir strukturdur*. Bu struktur daxilində dəyişkənlər ola bilər. Lakin əsasən vəzifəsini müəyyən edəcək funksiyaları tərkibində saxlayır. Bu xüsusiyyətə paketləşdirmə (**encapsulation**) deyilir.

Bunun digər bir xüsusiyyəti də paketləşdiriləcək funksiyaların necə işləyəcəyi müəyyən edilmədən, sadəcə necə istifadə ediləcəyi müəyyən edilə bilər ki, bu da vacibdir. Buna da mücərrədləşdirmə (**abstraction**) deyilir. Paketləşdirmə və mücərrədləşdirmə, obyekt müəyyən etmək üçün kifayət olan iki funksiyaadır.

Obyektlərin digər xüsusiyyətlərindən biri də törəmə xüsusiyyətidir. Obyekt təyin edilərkən, əvvəlcədən təyin olunmuş başqa bir obyektə özünə baza olaraq seçə bilər. Bu hal yeni təyin olunan obyektin özünə baza seçdiyi



obyektin xüsusiyyətlərindən istifadə etmə imkanlarına uyğun gəlir, bununla bərabər yeni obyektin baza obyekti ilə eyni xüsusiyyət daşımasına səbəb olur.

Yeni obyekt yeni xüsusiyyətlər qazana bildiyi kimi, bu xüsusiyyətləri təkmilləşdirə və dəyişdirə də bilər. Bu xüsusiyyətə **törəmə** (*derivation*), xüsusiyyətləri almağa isə miras alma (*inheritance*) adı verilir.

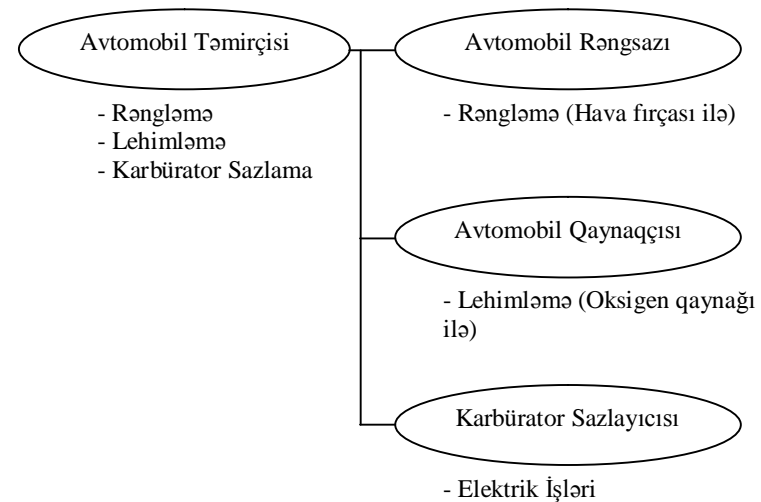
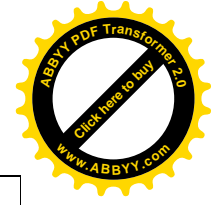
Buna aşağıdakı misalı göstərə bilərik:

Bir avtomobil təmirçisi avtomobil rəngləmə (hava fırça ilə), lehimləmə və karbürator sazlama işlərini yerinə yetirmiş olsun.

Avtomobil Təmirçisi

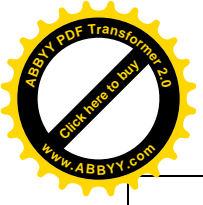
- Rəngləmə
- Lehimləmə
- Karbürator Sazlama

Bu avtomobil təmirçisi obyektidir. Bu avtomobil təmirçisinin üç oğlu olduğunu və onların hər birini yetişdirdikdən sonra, bir sahə üzrə ixtisaslaşdığını fərz edək.



Burada Avtomobil Rəngsazı, Avtomobil Qaynaqçısı və Avtomobil Elektriki törəmə obyektlərdir. Avtomobil Təmirçisi isə baza obyektidir. Törəmə obyektlər (Avtomobil Rəngsazı, Avtomobil Qaynaqçısı və Avtomobil Elektriki) baza obyektinin, yəni Avtomobil Təmirçisinin xüsusiyyətlərini göstərəcəklər. Uyğun olaraq avtomobil rəngləyəcək, lehimləyəcək və karbürator tənzimləyəcəklər.

Avtomobil Rəngsazından avtomobili rəngləməsini istədiyimiz zaman, o avtomobili hava fırçası ilə rəngləyəcəkdir. Lakin Avtomobil Təmirçisi bu işi yalnız adi fırça ilə görəcəkdə. Avtomobil Təmirçisi ilə Avtomobil Rəngsazının gördükləri iş rəngləmə işidir.



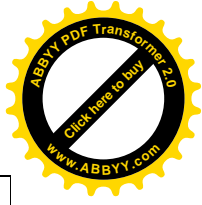
Avtomobil Rəngsazının adı fırça ilə rəngləməyi bacarmasına baxmayaraq hava fırçası ilə rəngləyir. Lakin istədiyi zaman adı fırça ilə də rəngləyə bilər.

Eyni şəkildə Avtomobil Qaynaqçısı da oksigen qaynağı ilə lehimləyir. Avtomobil Təmirçisi isə sadəcə qövs qaynağı ilə lehimləyə bilər. Avtomobil Qaynaqçısı eyni zamanda rəngləmə işlərini də bacarır. Bu qabiliyyəti Avtomobil Təmirçisindən miras almış və sadəcə fırça ilə rəngləməyi bacarır.

Üçüncü qolu təşkil edən Avtomobil Elektriki isə tamamilə başqa bir xüsusiyyətə malikdir. Avtomobil Təmirçisi elektrik işlərini görə bilmədiyi halda Avtomobil Elektriki bu işləri görə bilər. Avtomobil Elektriki eyni zamanda rəngləmə, lehimləmə və karbürətor tənzimləmə işlərini Avtomobil Təmirçisindən öyrəndiyi qədər görə bilər. Təbii ki, əgər lazım gələrsə, Avtomobil Elektrikinə məsələn, rəngləmə bacarığının ləğv edilməsi uyğun görülərsə, heç bir iş görməməsi təmin oluna bilər.

Burada növbəti üç xüsusiyyət nəzəri cəlb edir:

1. Törənmiş obyektlər baza obyektinin xüsusiyyətlərini qoruyub saxlayaraq istifadə edə bilirlər;
2. Törənmiş obyektlər törəndikləri obyektlərin (baza obyektlərinin) xüsusiyyətlərini dəyişdirə bilirlər;

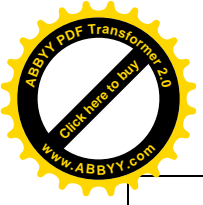


3. Törənmiş obyektlər yeni xüsusiyyətlər qazana bilirlər.

Bir sistem daxilində istər bir obyektədən törənmiş obyektlər olsun, istərsə də bir-birindən fərqli obyektlər olsun, bu obyektlərin oxşar xüsusiyyətləri ola bilər və bu xüsusiyyətlər eyni adla verilir. Bu da eyni adlı, lakin müxtəlif obyektlərin üzvü olan obyektlərin meydana gəlməsinə səbəb olur. Bu hal obyektlər arasındakı oxşarlıqları göstərir. Buna oxşarlıq (**polymorphism**) deyilir. Əslində bir obyektədən törənən siniflər arasında oxşarlığın olması vacibdir.

Obyektəyönlü proqramlaşdırmada obyektlərin malik olduqları məlumatları və funksiyaları qoruyaraq, birbaşa istifadə etməyə icazə verməmələri, başqa bir alt xüsusiyyətdir. Burada miras qoymağın əksi olan bir əməliyyatdan söhbət gedir. Bir obyekt bəzi xüsusiyyətlərini saxlayıb sadəcə özü istifadə edir. Digər obyektlərin istifadə etməsinə icazə verməz və ya məhdudlaşdırar. Bu, dörd müxtəlif hal ilə şərh edilə bilər:

1. Xüsusi (**private**) – bir üzv məlumatın və ya funksiyanın yalnız üzv olduğu obyekt daxilində istifadə olunması;
2. Qorunmuş (**protected**) – bir üzv dəyişkəninə və ya funksiyasına üzv olduğu obyekt xaricində o obyektədən törənən obyektlərin sadəcə istifadə edə bilməsi;

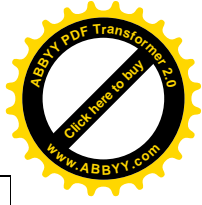


3. Ümumi (**public**) – bir üzvün bütün obyektlər tərəfindən ortaqlı istifadə edilməsi;
4. Dost (**friend**) – bir obyektin başqa bir obyektə dost elan edərək üzvlərinin hamısının bu obyekt tərəfindən istifadəsinə icazə verməsi.

Obyektliyi proqramlaşdırma digər əsas xüsusiyyət isə dinamik əlaqələndirmədir (**dynamic binding**). Bu xüsusiyyətlə törənən bir obyektin ünvanını törəndiyi obyektlərdən birinin göstərici (**pointer**) dəyişkənində mənimsətmək mümkündür. Bu halda ünvanı mənimsədilən obyekt, göstəricisinə mənimsədildiyi baza obyektə kimi davranacaq, həmçinin özünəməxsus xüsusiyyətlərini də göstərəcəkdir.

Bu halı belə şərh etmək olar: məsələn, bir müəssisənin baxış bölməsinə bir avtomobil təmirçisi işə alınacaqdır. Bu işə (göstərici dəyişkən) Avtomobil Təmirçisindən başqa Avtomobil Rəngsazı, Avtomobil Qaynaqçısı və Avtomobil Elektriki də müraciət edə bilər və bu işə alın bilər. Çünki, bunların kökündə avtomobil təmirçiliyi durur. Məlumdur ki, bu işə alınacaq şəxsin avtomobil rəngləmə, lehimləmə və karbüratör tənzimləmə işlərindən başı çıxacaqdır. Bu işi üç mütəxəssis birlikdə də görə bilər.

İndi də bu işə Avtomobil Qaynaqçısının alındığını fərz edək. Bu halda təmirçidən rəngləmək tələb olunarsa, o fırçadan istifadə edərək rəngləmə işini yerinə

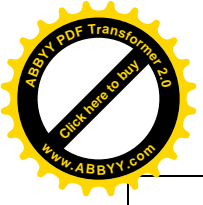


yetirəcəkdir. Çünki təmirçi kimi işə götürülən Avtomobil Qaynaqçısı, Avtomobil Təmirçisindən öyrəndiyi (miras aldığı) fırça ilə rəngləməyi bacarır. Lakin lehimləmək tələb olunduğunda, oksigen qaynağı ilə lehimləyəcəkdir. Çünki əsas bacardığı iş də elə budur.

Əgər bu işə Avtomobil Elektriki alınsaydı, qaynaq işlərini yalnız qəvs qaynağı ilə görərdi. Çünki, miras alma yolu ilə öyrəndiyi lehimləmə işi budur. Bu, rəngləmə və karbüratör tənzimləmə işləri də ola bilər. Avtomobil Elektrikinə bildiyi daha bir şey vardır ki, bu da elektrik işləridir. Lakin təmirçi kimi işə alınan Avtomobil Elektrikindən bu işi görməsi tələb olunmaz. Avtomobil Elektriki işə alınarkən görülməli işlər arasında elektrik işləri yoxdur.

Bir obyektin baza obyektinin göstəricisinə mənimsədilərək baza obyektə xüsusiyyətləri göstərməsinə *dinamik əlaqələndirmə* deyilir. Ümumi məqsədli alqoritmlərin (sıralama, axtarma kimi) tətbiqində və ya eyni xüsusiyyətli müxtəlif xarakterli obyektlərin birgə istifadəsində istifadə olun bilər. Yalnız bu xüsusiyyət üçün törətmə əməliyyatının olmasının vacibliyinə diqqət edin.

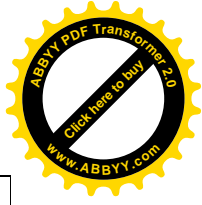
Törənmə xüsusiyyəti ilə əlaqədar olaraq daha bir xüsusiyyət obyektin bir obyektə deyil, bir neçə obyektə törənmiş olmasıdır. Buna çoxbazaılıq və ya çoxdan törənmə (**multi inheritance**) deyilir. Burada obyekt

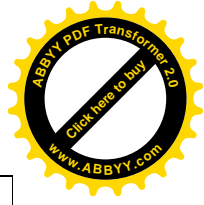
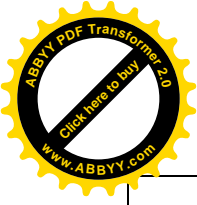


törəndiyi bütün obyektlərin xüsusiyyətlərini göstərir. Məsələn, İdarəçi və Avtomobil Təmirçisi obyektlərindən törənən Müdir obyekti həm İdarəçi, həm də Avtomobil Təmirçisi kimi davrana bilir. Yəni rəngləyə, karbürətor tənzimləyə, işçi alıb maaş verə bilir. Bu xüsusiyyətləri dəyişdirə və ya yeni xüsusiyyətləri tərkibinə ala bilir.

Çox bazalı obyektlərin əhəmiyyəti dinamik əlaqələndirmədə öz əksini tapır. Çoxbazalı obyekt törəndiyi bütün obyektlərin göstəricilərinə mənimsədilə bilər. Bu, mənimsətmə nəticəsində sadəcə mənimsədildiyi obyektin xüsusiyyətlərini göstərir. Məsələn, Müdir obyekti Avtomobil Təmirçisi göstəricisinə mənimsədilsə, Avtomobil Təmirçisi kimi davranır. Yox əgər İdarəçi göstəricisinə mənimsədilsə, İdarəçi kimi davranır.

Obyektyönlü proqramlaşdırmanın əsas anlayışlarından biri də proqramlar daxilində müəyyən qəliblərin (şablonların) hazırlanıb bir neçə dəfə istifadə oluna bilməsidir. Bu xüsusiyyətə şablonlama ([template](#)) deyilir. Şablonlar yazılmış bir proqram kodunun oxşar hallar üçün istifadə edilməsini ifadə edir. Məsələn, tam ədədlərdən ibarət olan bir massivə sıralamaq üçün proqram kodunun eyni zamanda həqiqi ədədləri, sətirləri və hətta yeni törədiləcək obyektləri də sıralaya biləcək bir şəkildə yazıla bilməsi şablonlama xüsusiyyətidir.





II FƏSİL

C-DƏ YENİLİKLƏR VƏ C-YƏ ƏLAVƏLƏR

2.1 Eyni Adlı Müxtəlif Arqumentli Funksiyalar

C və digər yüksək səviyyəli proqramlaşdırma dillərində arqument kimi daxil edilmiş eyni tipli iki qiymətdən ən böyüyünü geri qaytaran bir funksiya yazmaq üçün növbəti üsullardan istifadə etmək olar: Birinci üsul - istifadə olunan ən böyük aralıqlı tipə görə bir funksiya yazıb, alt tipləri bu tipə çevirməkdən ibarətdir. Məsələn,

```
long double Max(long double A, long double B)
{ return A < B ? B : A; }
```

və

```
long double LD;
double D;
int I;
char C;
```

təyin edildikdən sonra

```
LD = Max(12, 18);
D = (double)Max(12, 18);
I = (int)Max(12, 18);
C = (char)Max('C', 'H');
```

tip çevirmə operatorları ilə yerinə yetirilə bilər. Belə ki,

```
I = (int)Max(12e7, 18.36);
```

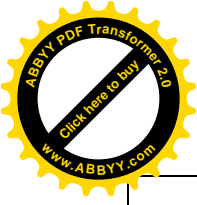
əməliyyatı heç bir səhv göstərmədən kompyasiya olunur. Lakin icra olunarkən müəyyən xətlər baş verə bilər. Sətirlərin ən böyüyünün təyin edilməsi üçün bu funksiyadan istifadə etmək olmaz. Bu halda kompyator səhvləri ola bilər. Bu cür səhvlər əhəmiyyətsiz sayılsa da, proqram səhv icra olunur.

İkinci üsul - hər tip üçün ayrı bir funksiya yazmaqdan ibarətdir. Bu halda **C** qaydalarına uyğun olaraq hər bir funksiya üçün ayrı addan istifadə etmək lazım gələcəkdir.

```
#include <string.h>
char Max_char(char A, char B)
{ return A < B ? B : A; }
```

```
int Max_int(int A, int B)
{ return A < B ? B : A; }
```

```
double Max_double(double A, double B)
{ return A < B ? B : A; }
```



```
char *Max_string(char* A, char* B)
{ return strcmp(A, B) < 0 ? B : A; }
```

Bu cür təyinlər bütün tiplər üçün yazıla bilər. Bu dəfə yuxarıda göstərilən təyini uyğun olaraq

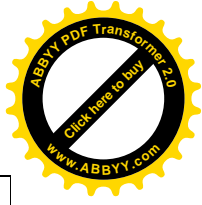
```
D = Max_double(12, 18);
I = Max_int(12, 18);
C = Max_char(12, 18);
```

şəklində olmalıdır. Bu halda tip çevirmə əməliyyatının aradan qalxmış olduğuna diqqət edin. Bunun əvəzində yalnız funksiya adı dəyişmişdir. Bu üsulla

```
I = Max_int(12e7, 18.36);
```

kimi əməliyyatları əvvəlcədən yerinə yetirmək də mümkündür. Böyük funksiyaların makro səviyyədə təyin edilməsinin proqramın böyüklüyünü artırdığını nəzərə alıqda, bu ən yaxşı üsul sayıla bilər. Bu zaman hansı tip üçün hansı funksiyadan istifadə ediləcəyini çox yaxşı bilmək lazımdır.

Lakin C++-da bir funksiya eyni adla bir neçə dəfə təyin oluna bilər. Hər təyində funksiya digərindən fərqli olaraq müstəqil bir funksiya kimi işləyir. Bu hadisəyə üzərinə yükləmə və ya üst-üstə qoyma (**overloading**) adı verilir. Bir ad üzərinə iki və ya daha artıq ad yüklənəcəksə, bu



overload funksiya_adı;

şəklində göstərilir. Bu şəkildə tanımlanmış hər funksiyanın argument siyahısındakı tip ardıcılığı daha əvvəl təyin edilmiş eyni adlı funksiyaların argument siyahısındakı tip ardıcılığı ilə eyni olmalıdır.

Buna görə **Max** funksiyasını aşağıdakı kimi təyin etmək olar:

```
// MAXDEC.CPP

#include <string.h>
#include <stdio.h>

overload Max;

int Max(int A, int B)
{ return A < B ? B : A; }

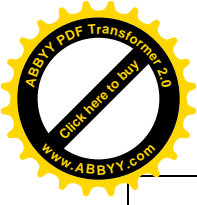
char Max(char A, char B)
{ return A < B ? B : A; }

double Max(double A, double B)
{ return A < B ? B : A; }

char *Max(char* A, char* B)
{ return strcmp(A, B) < 0 ? B : A; }
```

Bu təyindən sonra

```
C = Max('C', 'H');           // char C
I = Max(12, 18);              // int I
```



```
D = Max(12.4, 18.9);    // double D
S = Max("String", "Massiv"); // char *S
```

mənimsətmə əməliyyatlarını yerinə yetirmək olar. Bu mənimsətmələrin yerinə yetirilməsindən sonra hər **Max** funksiyası digərlərindən fərqlənir.

Belə ki,

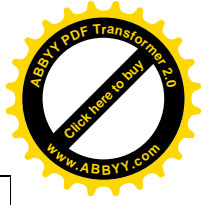
<pre>char C = Max('C', 'H'); int I = Max(12, 18); double D = Max(12.4, 18.9); char *S = Max("String", "Massiv");</pre>	<pre>char Max(char, char); int Max(int, int); double Max(double, double); char Max(char*, char*);</pre>
--	---

prototipli funksiyalar çağırılır. Hansı funksiyanın çağırılacağından təyin edilməsində funksiyanın adından başqa, argument siyahısındakı tiplər və tiplərin ardıcılığı əsas olmalıdır. Funksiyanın qaytardığı tip isə bir nəticə olaraq ortaya çıxır. Məsələn,

```
int    funksiya(char);
char   funksiya(int);
int    funksiya(int, int);
```

parametr siyahısındakı müxtəlifliyə görə bir-birindən fərqli olan üç funksiyayı təyin edərkən,

```
int    funksiya(char);
char   funksiya(char);
```



parametr siyahısındakı tip ardıcılığı eyni olduğu üçün eyni funksiyayı təyin edir. Qaytarılan tiplər müxtəlif olduğuna görə səhv aşkar edilir və bu cür təyin qəbul edilmir.

Üst-üstə yükləmə ilə əlaqədar təyin etmə əməliyyatları yerinə yetirilərkən tip təyinedicilərində müxtəlif tiplərin olduğuna diqqət edilməlidir. Yəni

```
int    funksiya(int);
int    funksiya(unsigned int);
```

kimi iki tip təyin edilərkən

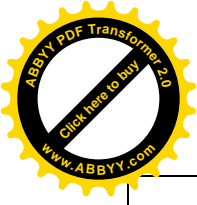
```
int    funksiya(int);
int    funksiya(signed int);
```

eyni funksiyayı iki dəfə təyin etmə mənasına gəldiyi üçün səhv verir. Eyni şəkildə

```
int    funksiya(int);
int    funksiya(const int);
```

bir-birindən fərqli iki təyin olmasına baxmayaraq, **const** yeni tip təşkil etmədiyi üçün birlikdə istifadə edilə bilməzlər.

Yeni təyin edilmiş tiplərdən istifadə edərkən funksiyalar üzərinə yükləmə aparmaq olar. Məsələn,



```
struct Tarix {  
    int gun, ay, il;  
};
```

təyinindən sonra

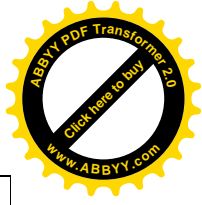
```
struct Tarix Max(struct Tarix A, struct Tarix B)  
{ if( A.il == B.il )  
    if(A.ay == B.ay )  
        return A.gun >= B.gun ? A : B;  
    else return A.ay >= B.ay ? A : B;  
    else return A.il >= B.il ? A : B;  
}
```

təyin edilə bilər.

Borland C++-da **overload** göstərilmədən də bütün funksiyaların üzərinə yükləmə aparıla bilər. Kompilyator Sizə bununla əlaqədar xəbərdarlıq edərsə, buna əhəmiyyət verməyin və ya **#pragma warn -ovl** direktivindən istifadə edərək bu tipli xəbərdarlıq məlumatlarının qarşısını ala bilərsiniz.

2.2 Operatorların Təyini

C++-da operatorları da funksiya kimi qəbul etmək olar.



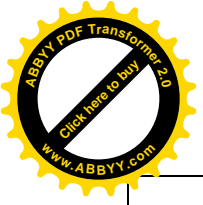
c = a + b;	yerinə	c = operator+ (a, b);
c = a - b;	yerinə	c = operator- (a, b);
c = a + b - d * 4;	yerinə	c = operator- (operator + (a, b), operator* (d, 4));

ifadələrini yazmaq olar. Burada **operator+**, **operator-** və **operator*** hər biri bir funksiyaadır. C++-da **operator** sözü funksiya adıdır. Bu addan istifadə edərkən ondan sonra bir operator işarəsi yazmaq lazımdır. C və C++-da təyin olunmuş operator işarələri Cədvəl 2.1-də göstərilmişdir.

Cədvəl 2.1 Operatorlar

Operator İşarəsi	Operator Adı
+ - * / %	Riyazi operator
= += -= *= /= %= ^= &= = <<= >>=	Mənimsetmə operatorları
^ &	Bit səviyyə operatorları
<< >>	Sürüşdürmə operatorları
< > == != <= >= ! &&	Məntiqi operatorlar
++ -- + -	Artırma/azaltma operatorları
()	Cevirmə operatoru
[]	Massiv operatoru
&	Ünvan operatoru
sizeof	Uzunluq operatoru
new, delete	Yaddaş operatoru

Operatorlara funksiya kimi baxıldığı üçün funksiyalar üçün nəzərdə tutulmuş olan üst-üstə yükləmə əməliyyatı operatorlar üçün də istifadə edilə bilər. Bununla da yeni təyin olunan tiplər üçün operator funksiyalarının üzərinə yükləmək olar.



```
struct vector { double x, y, z; }
```

```
double operator*(struct vector A, struct vector B)  
{ return A.x * B.x + A.y * B.y + A.z * B.z; }
```

Bu misalda təyin olunmuş **vector** tipindən asılı olaraq iki vektorun skalyar hasilini hesablayan vurma operatoru da təyin edilmişdir.

```
struct vector V1 = {1, 2, 3};  
struct vector V2 = {7, 8, 2};  
double skalyarhasil = V1 * V2;
```

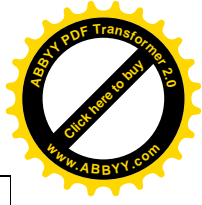
sətirləri program sətirləridir. Lakin hələ tanımlanmadığı üçün

```
struct vector V3 = V1 + V2;
```

kimi bir sətir istifadə edilə bilməz. Ancaq **vector** strukturu üçün toplama operatoru təyin edildikdən sonra istifadə oluna bilər.

2.3 Aktiv Qiymət Vermək

Müstəvi üzərində qövs, mərkəzi (**Cx**, **Cy**), radiusu (**R**), başlanğıc (**Sa**) və son (**Ea**) bucağı ilə təyin olunur. Belə bir qövsün uzunluğu $2\pi R(Ea-Sa)/360$ düsturu ilə hesablanır.



```
#include <math.h>
```

```
double Qovs(double Cx, double Cy, double R, double Sa,  
            double Ea)  
{ return 2*M_PI*R*(Ea-Sa)/360; }
```

Bu cür bir **Qovs** alt proqramı yazıla bilər. Əgər bu proqram hissəsi tam bir çevrə üçün istifadə olunarsa, **Sa** yerinə **0**, **Ea** yerinə isə **360** yazılmalıdır.

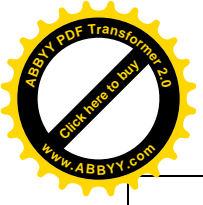
```
double Tam = Qovs(Cx, Cy, R, 0, 360);  
double Yarim = Qovs(Cx, Cy, R, 0, 90);
```

Belə ki, çevrə çox istifadə olunan olduğu üçün **0** və **360** qiymətləri əvvəlcədən məlum parametrlərdir. Hər dəfə bu qiymətləri göstərməyə ehtiyac yoxdur. Çevrə üçün

```
double Tam = Qovs(Cx, Cy, R);
```

şəklində istifadə olunması daha qısa yoldur. Bunu funksiya üzərinə yükləmə üsulundan istifadə edərək yenidən yazmaq mümkündür. Lakin bunun yerinə funksiya prototipi təyin edilərkən,

```
double Qovs(double Cx, double Cy, double R, double Sa = 0.0,  
            double Ea = 360.0);
```



şəkilində parametrlərə *aktiv qiymət* vermək də mümkündür. Bu təyindən sonra radiusu 50.0 olan bir çevrə təyin olunarkən

```
double Tam = Qovs(Cx, Cy, 50.0);
```

istifadə olunarsa, kompilyator əvvəlcə

```
Qovs(double, double, double)
```

kimi təyin olunmuş bir funksiya axtaracaqdır. Tapmadığı zaman da

```
Qovs(double, double, double, ...)
```

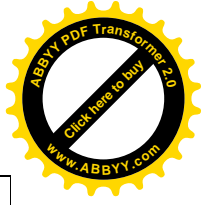
kimi və ilk üç qiymətdən sonrakılara aktiv qiymət verilmiş bir funksiya axtaracaqdır. Tapdığı zaman ilk qiymətləri də yazaraq bu funksiyanı çağıracaqdır. Yəni kompilyator

```
double Tam = Qovs(Cx, Cy, 50.0);
```

əmrini

```
double Tam = Qovs(Cx, Cy, 50.0, 0.0, 360.0);
```

kimi istifadə edəcəkdir.



Bu funksiyanın bütün parametrlərinə aktiv qiymət vermək mümkündür. Ya da misalda olduğu kimi, sadəcə müəyyən parametrlərə aktiv qiymət mənimsədilə bilər. Lakin aktiv qiymət verilmiş hər parametrin sağındakı parametrə də aktiv qiymət verilməlidir. Məsələn,

```
void misal(double x = 0, double y = 0);           //doğru  
void misal(double x, double y = 0);              //doğru  
void misal(double x = 0, double y);               //səhv
```

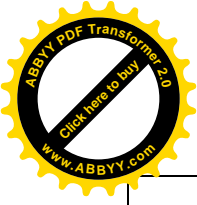
Üçüncü misal, aktiv qiymət verilmiş *x* parametrinin sağındakı bir parametrə (*y* parametrinə) aktiv qiymət verilmədiyi üçün istifadə oluna bilməz.

Daha bir halı nəzərə almaq lazımdır ki, aktiv qiymət verilərək əldə edilən prototiplərlə, üzərinə yükləmə aparılan funksiyalardan əldə edilən prototiplər eyni deyillər.

```
double Qovs(double Cx, double Cy, double R, double Sa =  
0.0, double Ea = 360.0);
```

təyini ilə bərabər

```
double Qovs(double Cx, double Cy, double R);  
double Qovs(double Cx, double Cy, double R, double Sa);  
double Qovs(double Cx, double Cy, double R, double Sa,  
double Ea);
```



funksiyaları birlikdə təyin edilə bilməzlər, çünki, anlaşılmazlıq yarana bilər.

```
Qovs(100.0, 120.0, 40.0);
```

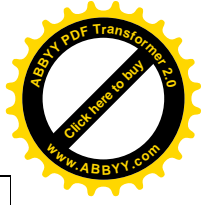
funksiyasının çağırılması zamanı hansı funksiya müraciət ediləcəyi bəlli olmur.

2.4 Təqdimat (Referans) Tip Təyinedicisi

C-də bir dəyişkənin başqa bir dəyişkəni göstərməsi üçün göstərici (**pointer**), məlumatların emalı üçün də göstərici əməliyyatları (**pointer arithmetics**) anlayışından istifadə olunur.

```
int X = 10, Y = 20;
int *P;
P = &X;
*P = 12;           /* X = 12; */
X = 14;            /* *P = 14; */
P = &Y;
*P = 12;           /* Y = 12; */
X = 15;
```

C++-da bu anlayış yerinə təqdimat tipi təyin edilir. Bu tipdli təyinlərdə dəyişkən başqa bir dəyişkənlə əlaqələndirilir. Lakin göstəricilərdən fərqli olaraq bu əlaqə pozularaq başqa bir əlaqə qurula bilməz.



int X;	
int &R = X;	/* & işarəsi R-in təqdimat dəyişkəni olduğunu göstərir. X mənimsədilməsi ilə də R və X bir-biri ilə əlaqələndirilir. */
X = 20;	/* R = 20; */
R = 40;	/* X = 40; */
R++;	/* X++; ++ operatorunun R-in qiymətini bir vahid artırdığına, sonrakı qiymətini göstərmədiyinə diqqət edin. */

Təqdimat dəyişkənlərinin təyini zamanı hansı dəyişkən ilə bağlı olduqlarının bildirilməsi məcburidir. Təqdimat dəyişkəni sabitlə də əlaqələndirilə bilər.

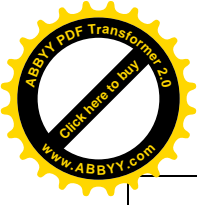
```
int &RR = 2;
```

Həqiqətdə isə kompilyator müvəqqəti bir dəyişkən təyin edərək ona 2 qiymətini mənimsədir və sonra bu dəyişkənin təqdimat dəyişkəni tərəfindən göstərilməsini təmin edir. Belə ki,

```
int Muveqqeti = 2;
int &RR = Muveqqeti;
```

Lakin proqramçının bu müvəqqəti dəyişkəni istifadə etmə haqqı yoxdur.

Bu çür təyin etmə müxtəlif vaxtlarda eyni məqsəd üçün istifadə edilmiş dəyişkənləri birləşdirməyə kömək edir.



```
/* REF.CPP */

#include <conio.h>
#include <stdio.h>
double Zaman = 0;

void ZamanYaz()
{ printf("Zaman = %lf\n", Zaman); }

/* ***** */

double &Z = Zaman;

void ZamanArtir()
{ Z += 6; }

int main()
{ clrscr();
  ZamanYaz();

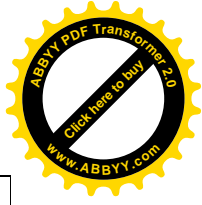
  ZamanArtir();
  ZamanYaz();

  ZamanArtir();
  ZamanYaz();

  return 0;
}
```

Program çıxışı

```
Zaman = 0.000000
Zaman = 6.000000
Zaman = 12.000000
```



Bu, funksiyalardan istifadə edərkən daha əhəmiyyətlidir. İndi **C**-də parametr kimi daxil edilən dəyişkənin qiymətini bir vahid artıran **INC** adlı funksiya və onu çağıran bir program yazaq.

```
/* INC.C */

void INC(double *D)
{ (*D)++; }

int main()
{ double X, *Y, Z;
  X = 6; Z = 8; Y = &Z;

  INC(&X);      /* X = 7; */
  INC(Y);       /* *Y = 9 ve ya Z = 9 */

  /* INC(*Y);      Sehvidir */

  return 0;
}
```

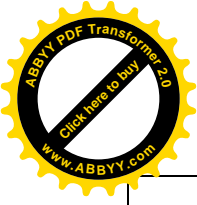
İndi də **C++**-də təqdimat təyin edicisi ilə bu programı yazaq.

```
/* INC.CPP */

void INC(double &D)
{ D++; }

int main()
{ double X, *Y, Z;
  X = 6; Z = 8; Y = &Z;

  INC(X);      /* X = 7; */
```



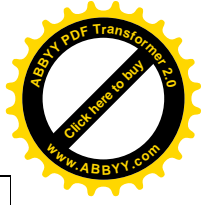
```
INC(*Y);      /* *Y = 9 və ya Z = 9 */  
  
/* INC(Y);      Sehvdır */  
  
return 0;  
}
```

2.5 Gizlənmiş Dəyişənləri Görmək

C-dən bildiyimiz kimi bir blok daxilində təyin olunmuş dəyişənin adı daha əvvəl təyin olunmuş dəyişən adı ilə üst-üstə düşərsə, son təyin olunmuş dəyişən blokun sonuna qədər öz funksiyasını yerinə yetirərək digər dəyişənə müraciətin qarşısını alır (ümumi və lokal dəyişən anlayışlarını xatırlayın).

```
/* SCOPE.C */  
  
#include <stdio.h>  
  
int X = 5;          /* X int tipində və qiyməti 5 */  
  
void f()  
{ double X = 27.5e30; /* X double tipində və qiyməti 2.75e31 */  
  
  X = 71;           /* X double tipində və qiyməti 71 */  
}  
  
/* X int tipində */
```

C++-da `X = 71` mənimsədilməsinin lokal deyil, ümumi təyin olunmuş `X`-ə mənimsədilməsini



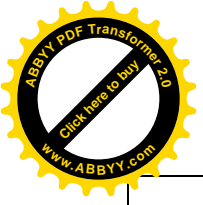
istəyirsinizsə, `::` təyinedicisindən, yəni görmə (`scope`) operatorundan istifadə etməlisiniz. Belə ki,

```
/* SCOPE.CPP */  
  
#include <iostream.h>  
  
static int X = 5;      /* X int tipində və qiyməti 5 */  
  
void f()  
{ double X = 27.5e30; /* X double tipində və qiyməti 2.75e31 */  
  
  cout<<X<<endl;  
  
  X = 71e12;           /* X double tipində və qiyməti 7.1e13 */  
  
  ::X = 71;            /* X int tipində və qiyməti 71 (qlobal X-dir) */  
}  
  
/* X int tipində qiyməti 71 */
```

Uyğun olaraq ifadələrdə ümumi təyin olunmuş dəyişənlərin

`Y = X * ::X + X + ::X;`

şəklində istifadəsi də mümkündür.



2.6 C++-da Prototiplərin Təyin Edilməsi

C++-da C-də istifadə olunan klassik stildəki prototip təyinlərinə icazə verilmir. Bunların yerinə daha müasir prototip təyinlərindən istifadə edilir.

```
double f(a, b, c);
int a;
double b;
float *c;
{
    ...
    ...
    ...
}
```

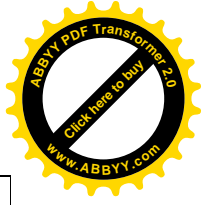
təyini səhvdir.

```
double f(int a, double b, float *c);
{
    ...
    ...
    ...
}
```

təyini doğrudur. Sadəcə prototip təyində də

```
double f(int a, double b, float *c);      /* ve ya */
double f(int, double, float*);
```

təyini doğrudur.



2.7 Struktur Tiplər

struct, union və enum strukturları ilə typedef təyinedicisi istifadə edilmədən tip təyinlərinin istifadə edilməsi zamanı struct, union və enum sözlərinin dəyişkənin adından əvvəl göstərilməsinin vacibliyi C++-da aradan qaldırılmışdır.

Tip təyini

```
struct Telebe
{ char Adi[20];
  int No;
  char Qiymeti;
};
```

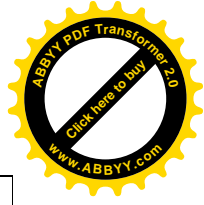
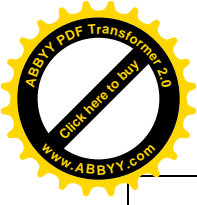
olarsa, C-də dəyişkənin təyini

```
struct Telebe A, B, C;
```

C++-da isə sadəcə

```
Telebe A, B, C;
```

şəklində yazıla bilər.



2.8 Şərh Operatoru

Proqram daxilində proqramçının verdiyi şərhlər növbəti mərhələlərdə düzəlişlər, dəyişikliklər və digər proqramçıların proqrama müdaxiləsi üçün çox əhəmiyyətlidir. Yaxşı yazılmış bir proqram daxilində proqram sətirlərindən daha çox şərh sətirləri olur. Bütün proqramlaşdırma mühitlərində olduğu kimi C-də də şərh operatoru vardır. Bu `/*` ilə başlayıb `*/` bitən sətirlərdir. Proqramın istənilən bir yerində qoyula bilər. Bu operator C++-da da istifadə oluna bilər.

C++-da bundan başqa `//` işarəsi ilə başlayan şərh operatorundan da istifadə olunur. Bu işarə ilə başlayan şərh sətirləri növbəti sətirdən davam edə bilməz.

//SERH.CPP

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

main()                // Proqrama baslama.
{ clrscr();           // Ekranın temizlenmesi.
  double PI = 3.14;    // Pi-nin qiymeti menimsedilir.
  double X;            // Bucaq qiymeti ucun teyin olunmus deyisken.
  int Err;             // Sehlere nezaret ucun teyin olunmus deyisken.
  printf("Bucagin qiymetini derece olaraq daxil ediniz.\n");
  Err = scanf("%lf", &X); // X-in qiymeti bucaqla daxil edilmeli.
                        // Yalniz bir qiymet daxil edilmelidir.
                        // Err-in qiymeti 1 olmalidir.
  if (Err != 1) // Err-in qiymeti 1-den ferqli olarsa, sehf oxuma bas verir.
  { fprintf(stderr, "Yalniz heqiqi eded daxil edin.\n");
```

```
    return 1;          // Xetali proqram cixisi.
  }
  fflush(stdin); // Giris yaddasindaki artiq melumatlari silir.

  X *= PI / 180.0;     // X radyana cevrilir.
  printf("%lf radyan\n", X);
  return 0;            // Proqramdan sehv olmadan cixilir.
}                      // Proqramin sonu.
```

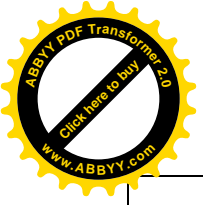
2.9 new və delete Opratorları

Dinamik yaddaşdan istifadə edərkən icra olunan iki əsas əməliyyat yaddaşda yer ayrılması (`malloc`, `calloc`) və ayrılan bu sahənin istifadə edildikdən sonra sərbəst (`free`) buraxılmasıdır. Bu əməliyyatlar üçün C-də prototipləri `stdlib.h` və `alloc.h` başlıq fayllarında göstərilən funksiyalardan istifadə edilir.

```
double *DP = (double*)malloc(sizeof(double));
int *IP = (int*)malloc(sizeof(int));
struct Date {int Gun, Ay, Il;};
struct Date *SDP = (struct Date*)malloc(sizeof(struct Date));
```

ayrılan bu sahələri sərbəst buraxmaq üçün də

```
free((void*)DP);
free((void*)IP);
free((void*)SDP);
```



əmləri istifadə edilir. C++-da bu funksiyalardan başqa eyni zamanda

```
double *DP = new double;  
int *IP = new int;  
struct Date {int Gun, Ay, Il};  
struct Date *SDP = new struct Date;
```

istifadə etmək və ayrılan bu sahələri sərbəst buraxmaq üçün də

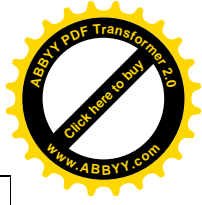
```
delete DP;  
delete IP;  
delete SDP;
```

ifadələrindən istifadə etmək olar.

Eyni tipdə iki və daha artıq yer ayırmaq üçün (məsələn, 100 double tipli elementi olan A massivi üçün)

```
double *A = new double[100];
```

əmrindən istifadə etmək olar. Bu əməliyyat nəticəsində 100 double tipli ədədin yerləşdirilməsi üçün sahə ayrılaraq ilk ünvanı A-ya mənimsədiləcəkdir. Bu mənimsətmə nəticəsində $*(A+3) = 81$; və ya $A[3] = 81$; əməliyyatları ilə massivin 4-cü elementinə qiymət mənimsədilməsi mümkündür. Burada massivin hər elementinin double tipində olmasına diqqət etmək lazımdır.



Belə bir sahəni sərbəst buraxmaq üçün

```
delete A;
```

əmrindən istifadə etmək kifayətdir. (Bəzi C++ proqramlarında delete əmri ilə yanaşı massivin ölçüsünü də göstərmək lazım gələ bilər (delete [100]A; kimi).

Əgər ayrılacaq sahəyə double tipli qiymətlərin göstəriciləri yerləşdiriləcəksə,

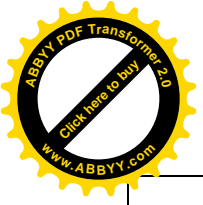
```
double **A;  
A = new double*;
```

kimi istifadə olunmalıdır. Əgər double göstəricilərinin massivi istifadə ediləcəksə,

```
A = new double *[100];
```

kimi istifadə olunmalıdır.

Bundan başqa new və delete operatorları üçün massiv kimi yer ayırmalarında istifadə edilən elementlərin sayının sabit olması vacib deyildir. Elementlərin sayı dəyişkən ola bildiyi kimi, əməliyyat nəticəsində də əldə edilə bilər.



2.10 inline Makroları

`inline` makroları parametrik makrolara oxşar şəkildə icra olunmasına baxmayaraq funksiyalar kimi təyin edilə bilər. Məsələn, funksiya təyini

```
int max(int a, int b)
{ return a > b ? a : b; }
```

makro təyini

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

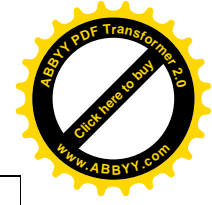
`inline` təyini aşağıdakı kimidir:

```
inline int max(int a, int b)
{ return a > b ? a : b; }
```

`inline` makrolarının funksiyalara oxşadığına diqqət edin. `inline` makrolarını təyin edərkən sanki, bir funksiya təyin etmiş olursunuz. Sadəcə təyin `inline` ifadəsi ilə başlayır.

Digər makrolardan fərqli olaraq `inline` makrolarında tip nəzarəti aparılır. Məsələn, yuxarıda təyin edilmiş `#define` makrosunun istifadə edilməsi zamanı

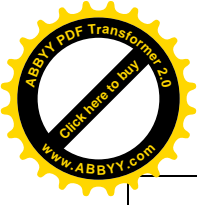
```
double d = max(3.8, 3.1);
```



heç bir səhv olmadan icra olunur. `inline` makrosundan istifadə edilərsə, bunun səhvsiz icra olunacağını gözləmək çətindir. Belə ki, `max(double, double)` kimi təyin olunmuş bir funksiya və ya makro yoxdur. Buna görə də `double` tipli ədədlər `int` tipinə çevrilərək istifadə olunacaq və nəticə də `3` olacaqdır.

`inline` makrolarının yazılacağı yer onların makro olması nəzərə alınaraq təyin edilməlidir. Yəni, program kodu hissəsində deyil, təyin etmə hissəsində yerləşdirilməlidir. Çünki `inline` makroları kitabxanalarda saxlanılır.

`inline` makroları ilə sürətli icra olunan sadə mənimsətmə və nəzarət əməliyyatlarından ibarət funksiyalar yazıla bilər. `inline` makroları daxilində `goto`, `for`, `do-while`, `while`, `break`, `continue`, `switch`, `case` əməllərinin istifadə edilməsi strukturun böyüməsinə səbəb olduğu üçün, bunların istifadə edilməsi məqsədsə uyğun deyildir.



III FƏSİL

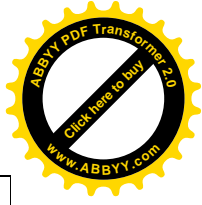
OBYEKT LƏR

3.1 Obyekt Nədir?

Obyekt (**Object**), yaddaşın dəyişdirilə bilən qiymətlər və ya müəyyən funksiyaları yerinə yetirən adlandırılmış sahəsidir. Bu baxımdan bütün dəyişkənlər bir obyektədir. Obyektlərin davranışlarına görə təsnifləndirilməsi də sinif (**class**) anlayışını meydana gətirir. Bu baxımdan da verilənlərin tipləri bir sinfi ifadə edir. Bəzi mənbələrdə sinif yerinə obyekt, obyekt yerinə isə nümunə (**instance**) anlayışından istifadə edilir.

Obyektlər dəyişkən və funksiyalardan ibarət olan struktur dəyişkənləridir. Obyektə daxil olan dəyişkənlərə üzv dəyişkənləri (**member variables**), funksiyalara da üzv funksiyaları (**member functions**) adı verilir.

C++-da obyektlər iki cür təyin olunur: **struct** və **class** sözü ilə başlayan təyin. Bunlar arasındakı yeganə fərq, əgər əksi göstərilməzsə, **struct** ilə təyin olunan obyektlərin bütün üzv dəyişkən və funksiyalarına digər obyekt və funksiyalar müraciət edərək istifadə edə bilirlər. **class** ilə təyin olunan obyektlərdə isə əksi



göstərilmədiyi halda, yalnız üzvlər bir-birlərini çağıraraq bilirlər. Digər obyekt və funksiyalar üçün isə bağlıdırlar.

İndi biz yalnız dairə və halqaları tanıyan, bunların sahə və çevrələri ilə əlaqədar olan obyektləri təyin etdikdən sonra istifadə edən proqramı tərtib edək.

```
// DAIRE.CPP
#include <math.h>

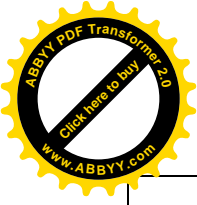
struct DAIRE {
    double Diametr;
    double Cevre();
    double Sahe();
};

double DAIRE::Cevre()
{ return Diametr * M_PI; }

double DAIRE::Sahe()
{ return Diametr * Diametr * M_PI / 4; }
```

Burada:

- **struct** strukturu daxilində funksiyaların da təyin edilməsinə;
- funksiyaların kodlaşdırılması zamanı heç bir təyin olmadan **Diametr** dəyişkəninin istifadə edilməsinə;



- kodlaşdırılma aparılarkən funksiya adının əvvəlinə funksiyanın aid olduğu sinif adının **DAIRE::** şəklində əlavə edilməsinə diqqət edin.

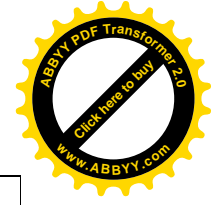
Funksiya adlarının əvvəlinə aid olduqları sinfin adının yazılmasının səbəbi digər siniflərin də eyni adlı üzv funksiyalarının ola biləcəyi ehtimalıdır. Müxtəlif siniflərə aid eyni adlı funksiyaları ayırmağın yeganə yolu sinif adı ilə bərabər görmə operatorunun istifadə edilməsidir.

Nəticə olaraq obyektin kodlaşdırılması ilə əlaqədar bu qaydaları qeyd etmək olar:

1. Funksiyalar da strukturun bir hissəsiymiş kimi dəyişkənlərlə birlikdə təyin oluna bilərlər;
2. Funksiya ilə eyni struktur daxilində təyin olunmuş dəyişkənlər heç bir təyin olmadan funksiya tərəfindən istifadə oluna bilərlər;
3. Struktura aid funksiyalar yazılarkən, aid olduqları struktur göstərmək üçün funksiyanın adının əvvəlinə strukturun adı, aralarına isə görmə operatoru yazılmalıdır.

```
//KVADRAT.CPP
```

```
struct KVADRAT {  
    double Hundurluk;  
    double Cevre();  
    double Sahe();  
};
```

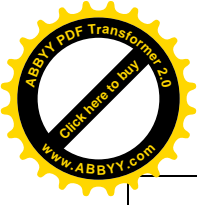


```
double KVADRAT::Cevre()  
{ return Hundurluk * 4; }  
  
double KVADRAT::Sahe()  
{ return Hundurluk * Hundurluk; }
```

Cevre və **Sahe** funksiyalarının burada **KVADRAT** üçün yenidən təyin edildiyinə diqqət edin.

```
//OBJECT1.CPP
```

```
#include <stdio.h>  
#include <conio.h>  
  
#include "daire.cpp"  
#include "kvadrat.cpp"  
  
KVADRAT K1, K2;  
  
DAIRE D;  
  
main()  
{ clrscr();  
  
    K1.Hundurluk = D.Diametr = 10.0;  
    K2.Hundurluk = 6;  
  
    printf("\nOlculer:\nKvadrat\tHundurluk1 = %f\t"  
        "Hundurluk2 = %f\nDaire\tDiametr = %f\n",  
        K1.Hundurluk, K2.Hundurluk, D.Diametr);  
  
    printf("\nSaheler:\nKvadrat\tSahe1 = %f\t"  
        "Sahe2 = %f\nDaire\tSahe = %f\n",  
        K1.Sahe(), K2.Sahe(), D.Sahe());  
  
    printf("\nCevreler:\nKvadrat\tCevre = %f\t"
```



```
"Cevre2 = %f\nDaire\tCevre = %f\n",
K1.Cevre(), K2.Cevre(), D.Cevre());

printf("\nKvadrat halqanın sahəsi = %f\n", K1.Sahe() - K2.Sahe());

return 0;
}
```

3.2 Layihələndirici

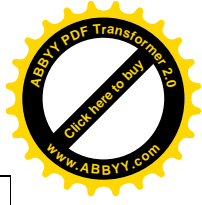
Dəyişənlər kimi, obyektləri də təyin edərkən onlara başlanğıc qiymət vermək olar. Bunun üçün obyekt sinfi təyin edilərkən, obyektin yaradılması zamanı istifadə ediləcək xüsusi bir funksiya obyekt sturukturu ilə bərabər təyin edilir. Bu funksiyanı digər funksiyalardan fərqləndirən əsas xüsusiyyəti adının təyin olunan sinif adı ilə eyni olmasıdır. Bu funksiya heç bir qiyməti geri qaytarmır. Bu funksiya layihələndirici (**constructor**) deyilir.

```
//DAIRECON.CPP

#include <math.h>

struct DAIRE
{ double Diametr;

    DAIRE(double);           // Layihelendircinin teyin edilmesi
    double Cevre();
    double Sahe();
};
```



```
DAIRE::DAIRE(double C)           // Layihelendircinin
yazılması
{ Diametr = C > 0 ? C : -C; }

double DAIRE::Cevre()
{ return Diametr * M_PI; }

double DAIRE::Sahe()
{ return Diametr * Diametr * M_PI / 4; }
```

Bu təyində bundan əvvəlki misala əlavə olaraq **DAIRE** sinfinin layihələndiricisi təyin edilmişdir.

```
//OBJECT2.CPP

#include <conio.h>
#include <stdio.h>
#include "dairecon.cpp"

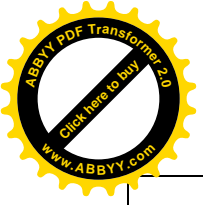
main()
{ clrscr();

    DAIRE Xarici(30);         // Layihelendircinin istifadə edilməsi
    DAIRE Daxili(20);

    printf("\nDaire diametrləri\nXarici -> %f\tDaxili -> %f\n",
        Xarici.Diametr, Daxili.Diametr);

    printf("\nDaire halqasının sahəsi = %f\n",
        Xarici.Sahe() - Daxili.Sahe());

    return 0;
}
```



Program çıxışı

Daire diametrləri
Xarici -> 30.000000 Daxili -> 20.000000
Daire halqasının sahəsi = 392.699082

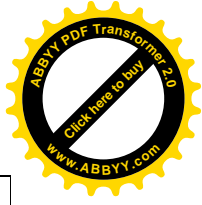
Bu cür istifadə ilə **Diametr** üzv dəyişkəninə müraciət sadələşir. Digər tərəfdən layihələndiricinin işə qoşulması avtomatik olaraq həyata keçirilir. Bunun üçün programçının əlavə cəhdlər etməsinə ehtiyac qalmır.

3.3 Müraciət Haqqı

Bir obyektin dəyişkən və funksiyalardan ibarət üzvlərinin digər obyektlər tərəfindən birbaşa istifadə edilməməsi üçün, bu obyektlər digər obyektlərə qarşı qoruna bilirlər.

Bu qorunma 4 müxtəlif halda ola bilər:

1. Bir üzv dəyişkən və ya üzv funksiyasının yalnız üzvü olduğu obyekt daxilində istifadə edilməsi - xüsusi (**private**);
2. Bir üzvün obyekt xaricində yalnız o obyektədən törənən obyektlər tərəfindən istifadə edilə bilməsi - qorunmuş (**protected**);
3. Bir üzvün bütün obyektlər tərəfindən ortaq istifadə edilə bilməsi - ümumi (**public**);

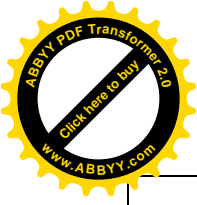


4. Bir obyektin başqa bir obyektə "dostu" elan edərək üzvlərinin hamısının bu obyekt tərəfindən istifadə edilməsinə icazə verməsi - dost (**friend**).

Bunlardan ilk üçü çox istifadə edilir.

```
//ADSTRC.H  
  
struct Adlar  
{ private:  
    char Ad[20];  
    int Yas;  
  
    void Boyuk();  
  
public:  
    Adlar(char*, int);  
  
    int Yaz();  
    int NormalYaz();  
};
```

struct ilə obyekt təyinlərində istifadə haqqı başlanğıcda **public** olur. Əksi göstərilmədikdə bu belə də qalır. **class** ilə obyekt təyinlərində isə başlanğıcda **private** olur. Bundan başqa **struct** və **class** vasitəsilə obyekt (sınıf) təyinləri arasında fərq yoxdur. Yuxarıdakı misaldan görüldüyü kimi hansı üzvün hansı istifadəçi səviyyəsində təyin ediləcəyini, təyindən əvvəl **private**;, **protected**; və ya **public**; kimi ifadələrdən istifadə edərək müəyyənləşdirmək mümkündür.



Yuxarıdakı misalda **Ad** və **Yas** dəyişənləri ilə **Boyuk** funksiyası **private**, digər üzv funksiyalar isə **public** kimi təyin olunmuşdur.

Eyni obyekti **class** açar sözündən istifadə edərək aşağıdakı kimi təyin etmək olar:

```
//ADCLASS.H
```

```
class Adlar
{
    char Ad[20];
    int Yas;

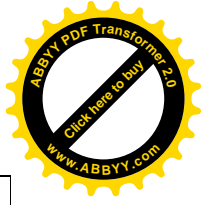
    void Boyuk();

public:
    Adlar(char*, int);

    int Yaz();
    int NormalYaz();
};
```

struct və **class** təyinlərinin müqayisəsini aşağıdakı kimi göstərmək olar:

<pre>struct ad { };</pre>	<pre>class ad { public: };</pre>
--	---



<pre>class ad { };</pre>	<pre>struct ad { private: };</pre>
---	---

Üzv funksiyalarının **struct** və ya **class** ilə kodlaşdırılması arasında elə bir fərq yoxdur.

Üzv funksiyalarının digər üzvləri, xüsusilə də üzv dəyişənlərini sanki, lokal dəyişənlər kimi istifadə etdiklərinə diqqət edin.

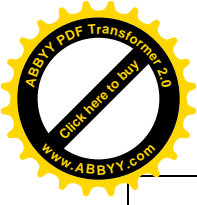
```
//ADDEC.CPP
```

```
#define __STRUCT

#ifdef __STRUCT
#include "adstrc.h"
#else
#include "adclass.h"
#endif

#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

Adlar::Adlar(char* N, int Y)
{ strcpy(Ad, N);
  Yas = Y;
}
```



```
void Adlar::Boyuk()
{ char *p;
  for(p = Ad; *p; ++p)
    if(islower(*p))
      *p = toupper(*p);
}

int Adlar::Yaz()
{ char t[20];
  int i;
  strcpy(t, Ad);
  Boyuk();
  i = NormalYaz();
  strcpy(Ad, t);
  return i;
}

int Adlar::NormalYaz()
{ return puts(Ad); }

Adlar A("Kenan Seyidzade", 7);
Adlar B("KAMRAN Resulov", 8);

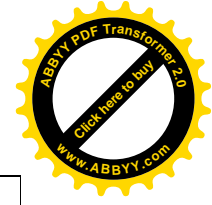
main()
{ clrscr();
  A.Yaz();
  B.Yaz();

  A.NormalYaz();
  B.NormalYaz();

  return 0;
}
```

Program çıxışı

KENAN SEYIDZADE
KAMRAN RESULOV



Kenan Seyidzade
KAMRAN Resulov

Belə bir proqram daxilində [A.Boyuk\(\)](#) kimi müraciət səhv qəbul ediləcəkdir. Bu funksiyanın mövcud olmasına baxmayaraq, [main\(\)](#) funksiyanının ona müraciət haqqı yoxdur.

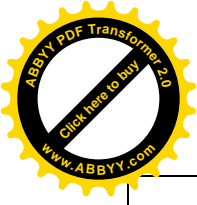
3.4 Yoxedici (Destructor)

Layihələndirici necə ki, obyektin mövcud olması halında onu proqrama hazırlayır, yoxedici funksiyası da obyekt proqram xaricində qaldığı zaman onun məhdudlaşdırdığı və ya dəyişdirdiyi kompüter mühitlərinin yenidən nizamlanmasını təmin edir. Bu funksiya, xüsusilə obyektin dinamik yaddaşdan istifadə etdiyi zaman ehtiyac olur.

Hər hansı bir yoxedicinin adı təyin olunduğu obyektin tipi (sinfi) ilə eynidir. Yalnız layihələndiricinin adı ilə qarışdırmamaq üçün əvvəlinə “~” (tildə) işarəsi qoyulur. Bu funksiya hər hansı bir qiymət hasil etmədiyi kimi, heç bir parametri də yoxdur.

// MASSIV.CPP

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



```
class Massiv
{ int *p;           //Massivin ilk elementinin gostericisi
  int Olcu;         //Massivin olcusu

public:
  Massiv(int s); //Layihelendirici s elementli bir massiv yaradir

  ~Massiv();     //Yoxedici

  int Getir(int i); //Massivin i-ci elementinin qiymetini verir

  void Menimset(int i, int d);
                        //Massivin i-ci elementine d-ni menimsedir

  void Sehv(char* msg); //Sehv mesajlarini gosterir

  double Orta();        //Massivin orta qiymetini hesablayir
};

Massiv::Massiv(int s)
{ printf("Layihelendirici islemeye basladi.\n");

  if (s <= 0) Sehv("Menfi ededler olcu ola bilmez!");
  if ((p = new int [Olcu = s]) == NULL)
    Sehv("Massive yer ayrila bilmedi!");

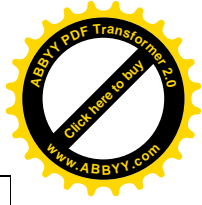
  printf("Layihelendirici isini dayandirdi.\n\n");
}

Massiv::~Massiv()
{ printf("Yoxedici islemeye basladi.\n");

  if (p != NULL) delete p;

  printf("Yoxedici isini dayandirdi.\n");
}

int Massiv::Getir(int i)
{ if (i < 0 || i > Olcu)
```



```
    Sehv("Massivin olcusu xaricinde!");
    return p[i];
}

void Massiv::Menimset(int i, int d)
{ if (i < 0 || i > Olcu)
    Sehv("Massivin olcusu xaricinde!");
  else p[i] = d;
}

void Massiv::Sehv(char* msg)
{ fprintf(stderr, "%s\n", msg);
  exit(4);
}

double Massiv::Orta()
{ int i;
  double t;

  for (i = 0, t = 0; i < Olcu; i++)
    t += p[i];
  return t / Olcu;
}

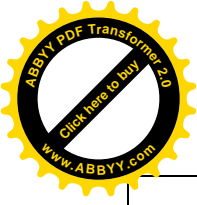
// ***** Misallar *****

Massiv A(3);

void Misal1()
{ printf("\n1-ci Misal\tLokal teyin etme\n\n");
  Massiv B(5);
}

#pragma warn -aus // 'C' is assigned a value that is never used
                  // mesajinin qarsisini alir

void Misal2()
{ printf("\n2-ci Misal\tLokal gosterici teyin etme\n\n");
  Massiv *C = new Massiv(5);
}
```

```
#pragma warn +aus

void Misal3()
{ printf("\n3-cu Misal\tLokal gosterici teyin etme ve silme\n\n");
  Massiv *D = new Massiv(5);
  delete D;
}

main()
{ clrscr();
  printf("main funksiyasi icra olunmaga basladi.\n");

  Misal1();
  Misal2();
  Misal3();

  printf("\nUzvlarin istifade olunmasi\n\n");

  Massiv E(4);

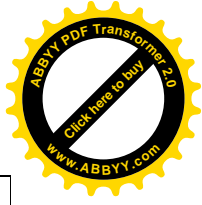
  int i;

  for (i = 0; i < 3; i++)
    A.Menimset(i, rand());
  for (i = 0; i < 4; i++)
    E.Menimset(i, rand());
  for (i = 0; i < 3; i++)
    printf("A[%d] = %d\n", i, A.Getir(i));

    printf("\nOrta qiymetler\n\n A = %lf, E = %lf\n\n",
      A.Orta(), E.Orta());

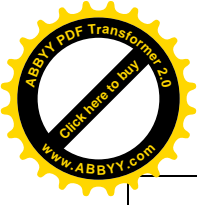
  printf("main funksiyasi icrasini tamamladi.\n\n");

  return 0;
}
```



Program çıxışı

main funksiyası icra olunmaga basladi.	A üçün
1-ci Misal Lokal teyin etme	
Layihelendirici islemeye basladi. Layihelendirici isini dayandirdi.	B üçün
Yoxedici islemeye basladi. Yoxedici isini dayandirdi.	B üçün
2-ci Misal Lokal gosterici teyin etme	
Layihelendirici islemeye basladi. Layihelendirici isini dayandirdi.	C üçün
3-cu Misal Lokal gosterici teyin etme ve silme	
Layihelendirici islemeye basladi. Layihelendirici isini dayandirdi.	D üçün
Yoxedici islemeye basladi. Yoxedici isini dayandirdi.	D üçün
Uzvlarin istifade olunmasi	
Layihelendirici islemeye basladi. Layihelendirici isini dayandirdi.	E üçün
A[0] = 346 A[1] = 130 A[2] = 10982	
Orta qiymetler	
A = 3819.333333, E = 9364.500000	
main funksiyası icrasını tamamladı.	



Yoxedici islemeye basladi.
Yoxedici isini dayandirdi.
Yoxedici islemeye basladi.
Yoxedici isini dayandirdi.

E üçün

A üçün

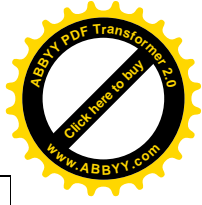
3.5 Standart Obyekt Tipləri

C-də `int`, `char`, `double` kimi tanınan standart tiplərin hər biri C++-da bir sinif kimi istifadə oluna bilər. Bunlara aşağıdakı kimi qiymətlər mənimləmək olar. Bu cür obyektlərə mənimləniləcək qiymət, siniflərə uyğun olmalıdır. Əgər uyğun deyilsə, onları uyğunlaşdırmaq lazımdır. Bu da bəzi səhvlərə yol açır.

<code>int x = 5;</code>	yerinə <code>int x(5);</code>
<code>double pi = 3.14;</code>	yerinə <code>double pi(3.14);</code>
<code>int Y = X;</code>	yerinə <code>int Y(X);</code>
<code>float *f;</code>	yerinə <code>float *f = new float(7.0/2.0);</code>
<code>f = (float*)malloc(sizeof(float));</code>	
<code>*f = 7.0/2.0;</code>	

3.6 Layihələndirici Üzərinə Yükləmə

Bir layihələndirici, üzvü olduğu obyektin başlanğıc vəziyyətini nizamlayarkən programçının istəkləri müxtəlif ola bilər. Bunun üçün də müxtəlif layihələndiricilərin istifadə edilməsi lazım ola bilər. Bunu yerinə yetirmək üçün Fəsil 2-də şərh edilən üzərinə



yükləmə üsulu layihələndiricilər üçün də istifadə oluna bilər.

Məsələn, bundan əvvəlki misalda baxdığımız massiv sinfinin layihələndiricisi verilən miqdarda yer ayırırdı. Əgər massivin ölçüsü göstərilməzsə, 100 olduğu qəbul edilsin və ölçüsündən asılı olmayaraq massivin hər bir elementinə -1000 ədədi mənimlədilsin. Yenə lazım gələrsə, başlanğıc qiymət də verilə bilər.

Bunun üçün üç layihələndiriciyə ehtiyac vardır:

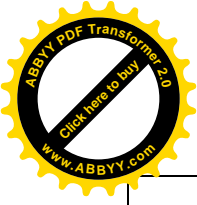
1. Heç bir qiymətin verilməməsi halı;
2. Ölçünün verilməsi halı;
3. Hər ikisinin verilməsi halı.

```
//MAS3CON.CPP

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

class Massiv
{ int *p;           //Massivin ilk elementinin gostericisi
  int Olcu;         //Massivin olcusu

public:
  Massiv();          //Layihelendirici ilk qiymeti -1000 olan 100
                    //elementli massiv yaradir.
  Massiv(int s);     //Layihelendirici ilk qiymeti -1000 olan s elementli
                    //massiv yaradir.
  Massiv(int s, int d); //Layihelendirici ilk qiymeti d olan s elementli
                    //massiv yaradir.
```



```
~Massiv();           //Yoxedici

int Getir(int i); //Massivin i-ci elementinin qiymetini verir

void Menimset(int i, int d); //Massivin i-ci elementine d-ni
                             //menimsedir

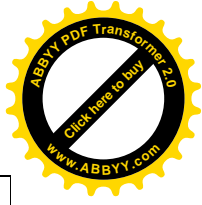
void Sehv(char* msg);      //Sehv mesajlarini gosterir

double Orta();             //Massivin orta qiymetini hesablayir
};

Massiv::Massiv()
{ int s = 100;              //Qebul edilen massivin olcusu 100;
  if ((p = new int [Olcu = s]) == NULL)
    Sehv("Massive yer ayrila bilmedi!");
  for (; s;)
    p[--s] = -1000;
}

Massiv::Massiv(int s)
{ if (s <= 0) Sehv("Menfi ededler olcu ola bilmez!");
  if ((p = new int [Olcu = s]) == NULL)
    Sehv("Massive yer ayrila bilmedi!");
  for (; s;)
    p[--s] = -1000;
  //s ededini Olcu uzv deyiskenine yazdigi ucun saygac olaraq
  //istifade edilmisdir.
}

Massiv::Massiv(int s, int d)
{ if (s <= 0) Sehv("Menfi ededler olcu ola bilmez!");
  if ((p = new int [Olcu = s]) == NULL)
    Sehv("Massive yer ayrila bilmedi!");
  for (; s;)
    p[--s] = d;
  //s yene saygac olaraq istifade edilmisdir.
}
```



```
Massiv::~~Massiv()
{ printf("Yoxedici islemeye basladi.\n");

  if (p != NULL) delete p;

  printf("Yoxedici isini dayandirdi.\n");
}

int Massiv::Getir(int i)
{ if (i < 0 || i > Olcu)
  { Sehv("Massivin olcusu xaricinde!");
    return p[i];
  }

void Massiv::Menimset(int i, int d)
{ if (i < 0 || i > Olcu)
  { Sehv("Massivin olcusu xaricinde!");
    else p[i] = d;
  }

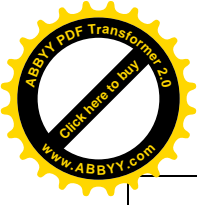
void Massiv::Sehv(char* msg)
{ fprintf(stderr, "%a%s\n", msg);
  exit(4);
}

double Massiv::Orta()
{ int i;
  double t;

  for (i = 0, t = 0; i < Olcu; i++)
    t += p[i];
  return t / Olcu;
}

// ***** Misallar *****

Massiv A(3);
```



OBYEKT LƏR

```
void Misal1()
{ printf("\n1-ci Misal\tLokal teyin etme\n\n");
  Massiv B(5);
}

#pragma warn -aus      // 'C' is assigned a value that is never used
                        // mesajinin qarsini alir

void Misal2()
{ printf("\n2-ci Misal\tLokal gosterici teyin etme\n\n");
  Massiv *C = new Massiv(5);
}

#pragma warn +aus

void Misal3()
{ printf("\n3-cu Misal\tLokal gosterici teyin etme ve silme\n\n");
  Massiv *D = new Massiv(5);
  delete D;
}

main()
{ clrscr();
  printf("main funksiyasi icra olunmaga basladi.\n");

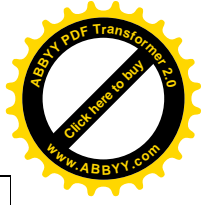
  Misal1();
  Misal2();
  Misal3();

  printf("\nUzvlarin istifade olunmasi\n\n");

  Massiv E(4);

  int i;

  for (i = 0; i < 3; i++)
    A.Menimset(i, rand());
  for (i = 0; i < 4; i++)
    E.Menimset(i, rand());
```



Etibar Seyidzadə

```
for (i = 0; i < 3; i++)
  printf("A[%d] = %d\n", i, A.Getir(i));

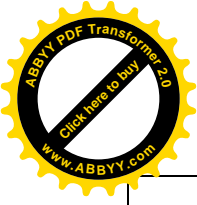
printf("\nOrta qiymetler\n\n A = %lf, E = %lf\n\n",
  A.Orta(), E.Orta());

printf("main funksiyasi icrasini tamamladi.\n\n");

return 0;
}
```

Proqram çıxışı

main funksiyasi icra olunmaga basladi.	A üçün
1-ci Misal Lokal teyin etme	
Yoxedici islemeye basladi. Yoxedici isini dayandirdi.	B üçün
2-ci Misal Lokal gosterici teyin etme	
3-cu Misal Lokal gosterici teyin etme ve silme	
Yoxedici islemeye basladi. Yoxedici isini dayandirdi.	D üçün
Uzvlarin istifade olunmasi	
A[0] = 346 A[1] = 130 A[2] = 10982	
Orta qiymetler	
A = 3819.333333, E = 9364.500000	
main funksiyasi icrasini tamamladi.	



Yoxedici islemeye basladi.
Yoxedici isini dayandirdi.
Yoxedici islemeye basladi.
Yoxedici isini dayandirdi.

E üçün

A üçün

Bu təyindən sonra **Massiv** sinfinin müxtəlif halları üçün aşağıdakıları qeyd etmək olar:

```
Massiv X(10, 5);
```

10 elementli bir massiv obyektı yaddaşda yerləşdirilir və hər elementə **5** qiyməti mənimsədir.

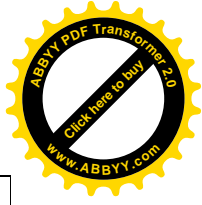
```
Massiv Y(20);
```

20 elementli massiv obyektı yaradılır və bütün elementlərə başlanğıc qiymət olaraq **-1000** mənimsədir.

```
Massiv Z;
```

Heç bir başlanğıc şərt verilmədiyi üçün **Z** massivi üçün **100** elementlik yer ayrılır və hər elementə **-1000** qiyməti mənimsədir.

Layihələndirici üzərinə yükləmələr başlanğıc şərtlər müxtəlif olmasına baxmayaraq, eyni davranışlı hadisələri təyin etmək üçün çox istifadə edilən bir yoldur. Layihələndirici xaricində yoxedici funksiyasından başqa



bütün üzv funksiyalar üzərinə yükləmə əməliyyatını aparmaq mümkündür.

Bununla bərabər üzv funksiyalarının və layihələndiricinin parametrlərinə aktiv qiymət vermək mümkündür. Belə ki, yuxarıda göstərilən misalda olduğu kimi üç layihələndirici təyin etməkdənsə, sadəcə

```
Massiv(int s, int d);
```

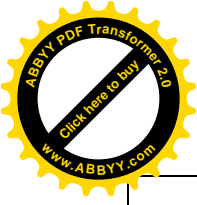
layihələndiricisini təyin edib, **s** və **d**-yə aktiv qiymət verilsə idi,

```
Massiv(int s = 100, int d = -1000);
```

kimi təyin olunmuş kod eyni qalmaqla eyni işi görə bilərdi.

3.7 Obyektlərə Mənimsətmə

Bir qayda olaraq obyektlərə mənimsətməyə ehtiyac yoxdur. Obyektlər üzv funksiyalar tərəfindən nəzarət olunmalıdır. Əgər obyekt üçün mənimsətmə operatoru **"="** yenidən təyin edilməzsə, mənimsətmələr layihələndiricinin çağırılması şəklinə çevrilərək tətbiq edilir. Bu da yalnız obyekt ilk dəfə yaradılarkən yerinə yetirilir.



Məsələn, `Massiv X = 18;` mənimşədilməsi `Massiv X(18);` kimi istifadə edilir. Bu da `X`-ə `18` elementlik yer ayrılması və hər elementə `-1000` mənimşədilməsi deməkdir.

Əgər `Massiv X = "Setir";` mənimşədilsə, `Massiv X("Setir");` müraciəti yaranacaqdır ki, bu da səhv olacaqdır.

Əgər bir obyektə eyni sinfə daxil olan başqa bir obyekt mənimşətmək tələb olunarsa, əvvəlcə belə bir mənimşətmənin olub olmayacağına nəzarət edilir. Təyin olunmuşsa, operator funksiya icra olunur. Təyin olunmamışsa, layihələndirici siyahısında belə bir layihələndiricinin olub olmamasına baxılır. Əgər orada da yoxdursa, mənimşədiləcək obyektin üzv dəyişkənləri digərinə bir-bir köçürülür.

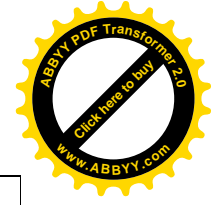
```
//AD2.CPP
```

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

class Adlar
{ char Ad[30];
  int Yas;

public:
  Adlar(char*, int);

  void Deyisdir(char*);
  void Yaz();
```



```
};

Adlar::Adlar(char* ad, int yas)
{ strcpy(Ad, ad);
  Yas = yas;
}

void Adlar::Deyisdir(char* YeniAd)
{ strcpy(Ad, YeniAd); }

void Adlar::Yaz()
{ printf("\nObyekt\n\nAdi : %s\nYasi : %d\n", Ad, Yas); }

main()
{ clrscr();
  Adlar A("Etibar Seyidzade", 35);
  Adlar B = A;

  A.Yaz();
  B.Yaz();

  printf("\nA-nin adi deyisdirildi.\n\n");
  A.Deyisdir("Seyidzade Etibar Vaqif oglu");
  A.Yaz();
  B.Yaz();

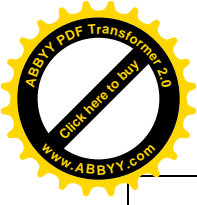
  return 0;
}
```

Proqram çıxışı

Obyekt

Adi : Etibar Seyidzade
Yasi : 35

Obyekt



OBYEKTlər

Adi : Etibar Seyidzade
Yasi : 35

A-nin adi deyisdirildi.

Obyekt

Adi : Seyidzade Etibar Vaqif oglu
Yasi : 35

Obyekt

Adi : Etibar Seyidzade
Yasi : 35

Misalda $B = A$ mənimsədilməsi ilə A -nın saxladığı qiymətlər B -yə mənimsədilmiş olur.

Proqram aşağıdakı kimi yazılırsa,

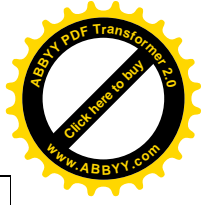
```
//AD3.CPP

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

class Adlar
{ char *Ad;
  int Yas;

public:
  Adlar(char*, int);
  ~Adlar();

  void Deyisdir(char*);
  void Yaz();
};
```



Etibar Seyidzade

```
Adlar::Adlar(char* ad, int yas)
{ Ad = new char[strlen(ad)+1];
  if(!Ad) // if(Ad == NULL) menasinda
    abort();
  strcpy(Ad, ad);
  Yas = yas;
}

Adlar::~~Adlar()
{ if(Ad) // if(Ad != NULL) menasinda
  { delete Ad;
    Ad = NULL;
  }
}

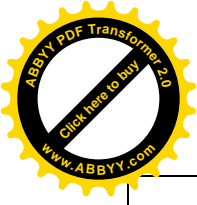
void Adlar::Deyisdir(char* YeniAd)
{ if(Ad) delete Ad;
  Ad = new char[strlen(YeniAd)+1];
  if(!Ad) abort();
  strcpy(Ad, YeniAd);
}

void Adlar::Yaz()
{ printf("\nObyekt\n\nAdi : %s\nYasi : %d\n", Ad, Yas); }

main()
{ clrscr();
  Adlar A("Etibar Seyidzade", 35);
  Adlar B = A;

  A.Yaz();
  B.Yaz();

  printf("\nA-nin adi deyisdirildi.\n\n");
  A.Deyisdir("Seyidzade Etibar Vaqif oglu");
  A.Yaz();
  B.Yaz();
}
```



OBYEKT LƏR

```
return 0;  
}
```

Proqram çıxışı aşağıdakı kimi olacaqdır:

Obyekt

Adi : Etibar Seyidzade
Yasi : 35

Obyekt

Adi : Etibar Seyidzade
Yasi : 35

A-nin adi deyisdirildi.

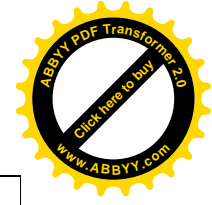
Obyekt

Adi : Seyidzade Etibar Vaqif oglu
Yasi : 35

Obyekt

Adi : Seyidzade Etibar Vaqif oglu
Yasi : 35

Göründüyü kimi heç bir problem yoxdur. Hətta **A** ilə **B** üzvləri bir-biri ilə əlaqəli olduqları üçün misalda olduğu kimi birinin ad dəyişikliyindən digərinin də xəbəri olacaqdır. Yox edilərkən **Ad** göstəricisinin göstərdiyi yaddaş sahəsi əvvəlcə obyektlərin biri tərəfindən, sonra da eyni sahə ikinci obyekt tərəfindən sərbəst buraxılır. Eyni sahənin iki dəfə sərbəst



Etibar Seyidzadə

buraxılması isə çox güman ki, növbəti mərhələlərdə böyük səhvlərə yol açacaqdır. Belə halların meydana gəlməməsi üçün mənimsətmə operatoru təyin edilməli və ya layihələndiricilərin daxilinə başqa bir layihələndirici də əlavə edilməlidir.

//AD4.CPP

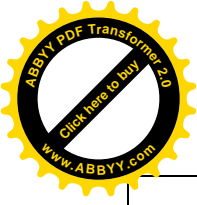
```
#include <conio.h>  
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>
```

```
class Adlar  
{ char *Ad;  
  int Yas;
```

```
public:  
    Adlar(char*, int);  
    Adlar(Adlar&);  
    ~Adlar();  
  
    void Deyisdir(char*);  
    void Yaz();  
};
```

```
Adlar::Adlar(char* ad, int yas)  
{ Ad = new char[strlen(ad)+1];  
  if(!Ad) // if(Ad == NULL) menasinda  
    abort();  
  strcpy(Ad, ad);  
  Yas = yas;  
}
```

```
Adlar::Adlar(Adlar&Obyekt)  
{ Ad = new char[strlen(Obyekt.Ad)+1];
```

OBJEKTlər

```
if(!Ad) abort();
strcpy(Ad, Obyekt.Ad);
Yas = Obyekt.Yas;
}

Adlar::~Adlar()
{ if(Ad) // if(Ad != NULL) menasinda
  { delete Ad;
    Ad = NULL;
  }
}

void Adlar::Deyisdir(char* YeniAd)
{ if(Ad) delete Ad;
  Ad = new char[strlen(YeniAd)+1];
  if(!Ad) abort();
  strcpy(Ad, YeniAd);
}

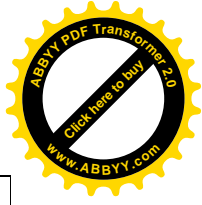
void Adlar::Yaz()
{ printf("\nObyekt\n\nAdi : %s\nYasi : %d\n", Ad, Yas); }

main()
{ clrscr();
  Adlar A("Etibar Seyidzade", 35);
  Adlar B = A;

  A.Yaz();
  B.Yaz();

  printf("\nA-nin adi deyisdirildi.\n\n");
  A.Deyisdir("Seyidzade Etibar Vaqif oglu");
  A.Yaz();
  B.Yaz();

  return 0;
}
```



Etibar Seyidzade

Program Çıxışı

Obyekt

Adi : Etibar Seyidzade
Yasi : 35

Obyekt

Adi : Etibar Seyidzade
Yasi : 35

A-nin adi deyisdirildi.

Obyekt

Adi : Seyidzade Etibar Vaqif oglu
Yasi : 35

Obyekt

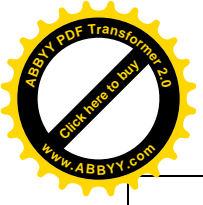
Adi : Etibar Seyidov
Yasi : 34

Mənimsətmə ilə əlaqədar bu problem mənimsətmə olmasa belə, obyektlərin funksiyalara parametr kimi göndərilməsi zamanı meydana gəlir.

long Axtar(Adlar X);

prototipli funksiyaya

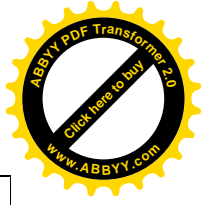
adlar A("Seyidzade Kenan", 8);

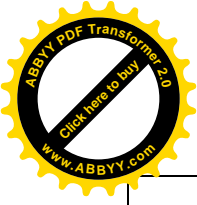


kimi bir parametr ötürüldüyü zaman program sanki, $X = A$; mənimsədilməsi baş vermişdir kimi davranacaq və haqqında söhbət açılan problemlər meydana gələcəkdir. Buna görə də **AD4.CPP** programında olduğu kimi problemin qarşısını almaq lazım gələcəkdir.

Obyektlərin təyin edilməsi üçün bu səbəbdən lazım gələn digər layihələndiricilərlə bərabər daha iki layihələndirici də olur:

1. Standart layihələndirici (**default constructor**) - **Sinif_adı()**; şəklində təyin edilən bu layihələndirici heç bir ilkin şərt verilmədiyi halda istifadə olunur;
2. Köçürmə layihələndiricisi (**copy constructor**) - **Sinif_adı(Sinif_adı&)**; şəklində təyin edilən bu layihələndirici isə əvvəlcədən təyin edilmiş bir hadisənin qiymətlərini yeni yaradılmaqda olan hadisəyə ötürür. Əvvəlki obyektin bir nüsxəsini çıxarır.





IV FƏSİL

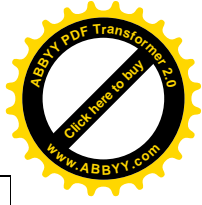
OBYEKT LƏRİN XÜSUSİYYƏTLƏRİ

4.1 Obyekt Üzvləri Olan Obyektlər

Bir obyektin üzv dəyişənləri arasında digər obyektlər də ola bilər. İstifadə edilib edilməməsindən asılı olmayaraq bu obyektlərin digər dəyişənlərdən fərqi yoxdur.

```
class A
{ int I;
  ...
  public:
  A();
  A(A&);
  A(int);
  ...
  ...
};

class B
{ A a, b;
  ...
  public:
  B();
  B(B&);
  B(int, int);
```



```
...
};
```

kimi təyin edilmiş iki sinifdən birincisinin kodlaşdırılmasında indiyə qədər şərh etdiklərimizə əlavə ediləcək yeni bir şey yoxdur. İkinci sinfin kodlaşdırılmasında isə

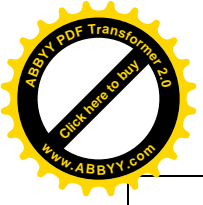
```
B::B()
{ ...
  ...
}
```

kimi bir layihələndiricinin kodlaşdırılmasında **a** və **b** obyektləri üçün standart layihələndiricilər avtomatik olaraq çağırılır. Əgər bunun yerinə başqa bir layihələndiricinin istifadə olunması tələb olunarsa,

```
B::B()
:a(33),
b(12) { ... }

B::B(int u, int v)
:a(u),
b(v) { ... }
```

şəklində layihələndirici başlığının yazılmasından sonra ":" işarəsi, üzv obyektlərin istənilən layihələndiriciləri arasına isə "," qoyularaq yazıla bilər. Layihələndiricisi



təyin olunmayan obyektlər üçün isə standart layihələndirici avtomatik olaraq çağırılacaqdır.

```
//DOST1.CPP

#include <conio.h>
#include <stdio.h>
#include <math.h>

class Noqte
{ float X, Y, Z;

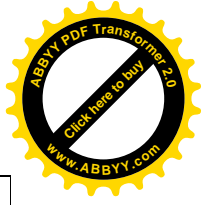
public:
    Noqte(float = 0, float = 0, float = 0);
    Noqte(Noqte&);

    int Dasi(float, float, float);
    int Yaz();
    float Mesafe(Noqte&);
};

Noqte::Noqte(float _x, float _y, float _z)
{
    X = _x;
    Y = _y;
    Z = _z;
}

Noqte::Noqte(Noqte& N)
{
    X = N.X;
    Y = N.Y;
    Z = N.Z;
}

int Noqte::Dasi(float dx, float dy, float dz)
{ X += dx;
```



```
Y += dy;
Z += dz;
return 0;
}

int Noqte::Yaz()
{ printf("%f, %f, %f", X, Y, Z);
  return 0;
}

#define DIFSQR(p) ((p-lkinci.##p)*(p-lkinci.##p))
float Noqte::Mesafe(Noqte& lkinci)
{ return sqrt(DIFSQR(X) + DIFSQR(Y) + DIFSQR(Z)); }
#undef DIFSQR

class Duxzett
{ Noqte Baslangic, Bitis;

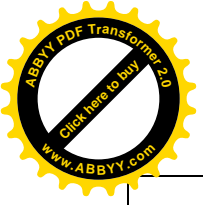
public:
    Duxzett(Noqte&, Noqte&);
    Duxzett(Duxzett&);

    float Uzunluq();
    int Yaz();
};

Duxzett::Duxzett(Noqte& A, Noqte& B)
:Baslangic(A),
Bitis(B)
{}

Duxzett::Duxzett(Duxzett& D)
:Baslangic(D.Baslangic),
Bitis(D.Bitis)
{}

float Duxzett::Uzunluq()
{ return Baslangic.Mesafe(Bitis); }
```



```
int Duzxett::Yaz()
{ printf("[");
  Baslangic.Yaz();
  printf("-");
  Bitis.Yaz();
  printf("]");
  return 0;
}

main()
{ clrscr();

  Noqte A;
  Noqte B(30, 40, 50);

  A.Yaz();
  printf("\n");
  B.Yaz();
  printf("\n\niki noqte arasindaki mesafe = %f\n\n", A.Mesafe(B));

  B.Dasi(10, -10, 60);
  Duzxett D(Noqte(10, 20, 30), B);

  D.Yaz();
  printf("\n\nduz xettin uzunlugu = %f\n\n", D.Uzunluq());

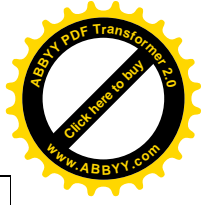
  return 0;
}
```

Program çıxışı

```
(0.000000, 0.000000, 0.000000)
(30.00000, 40.00000, 50.00000)

iki noqte arasindaki mesafe = 70.710678

[(10.00000, 20.00000, 30.00000)-(40.00000, 30.00000, 110.00000)]
```



```
duz xettin uzunluđu = 86.023253
```

4.2 Friend (Dost) Təyinedicisi

Bir sinfin üzvlərinin digər sinif və funksiyalardan qorunması bəzən mənfi hallara gətirib çıxarır. Bu üzvlərin bütün istifadələrə açılması da bəzən problemlər çıxara bilər. Bu baxımdan bir sinfin üzvləri qorunarkən bəzi sinif və funksiyalardan qorunmayıb, xüsusi (**private** təyinli) üzvləri, bu sinif və funksiyalar tərəfindən istifadə oluna bilər. Qadağan olunmasına baxmayaraq, bütün xüsusi üzvlərə müraciət hüququ verilən funksiyalara dost-funksiya (**friend-function**), siniflərə isə dost-sinif (**friend-class**) deyilir.

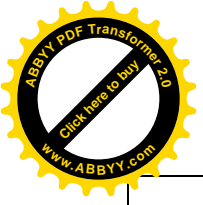
```
//DOST2.CPP

#include <conio.h>
#include <stdio.h>
#include <math.h>

class Noqte
{ float X, Y, Z;

public:
  Noqte(float = 0, float = 0, float = 0);
  Noqte(Noqte&);

  int Dasi(float, float, float);
```



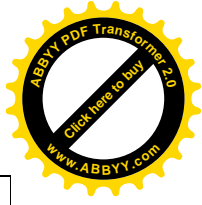
OBYEKTŁƏRİN XÜSUSİYYƏTLƏRİ

```
friend int Yaz(Noqte&);  
friend float Mesafe(Noqte&, Noqte&);  
};
```

Yaz və **Mesafe** üzv funksiyası deyildir. Normal olaraq prototipləri **Noqte** sinfinin xaricində təyin olunmalıdır. Bu iki funksiya vəzifələrinə görə **Noqte** sinfinin xüsusi dəyişənlərinə müraciət etməli olduqlarından bu sinif tərəfindən **friend** olaraq təyin edilmişdir.

DOST2.CPP davamı

```
Noqte::Noqte(float _x, float _y, float _z)  
{  
    X = _x;  
    Y = _y;  
    Z = _z;  
}  
  
Noqte::Noqte(Noqte& N)  
{  
    X = N.X;  
    Y = N.Y;  
    Z = N.Z;  
}  
  
int Noqte::Dasi(float dx, float dy, float dz)  
{  
    X += dx;  
    Y += dy;  
    Z += dz;  
    return 0;  
}  
  
int Yaz(Noqte& N)
```



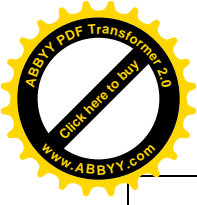
Etibar Seyidzadə

```
{  
    printf("(%f, %f, %f)", N.X, N.Y, N.Z);  
    return 0;  
}  
  
float Mesafe(Noqte& Birinci, Noqte& Ikinci)  
{  
    return sqrt((Birinci.X - Ikinci.X) * (Birinci.X - Ikinci.X) +  
                (Birinci.Y - Ikinci.Y) * (Birinci.Y - Ikinci.Y) +  
                (Birinci.Z - Ikinci.Z) * (Birinci.Z - Ikinci.Z));  
}
```

Yaz və **Mesafe** funksiyaları üzv funksiyası olmadıqları üçün kodlaşdırılarkən "**Noqte::**" ifadəsi başlıq daxilində olmamalıdır. Olarsa, bu səhv qəbul ediləcəkdir.

DOST2.CPP davamı

```
class Duzxett  
{  
    Noqte Baslangic, Son;  
  
public:  
    Duzxett(Noqte&, Noqte&);  
    Duzxett(Duzxett&);  
  
    float Uzunluq();  
  
    friend int Yaz(Duzxett&);  
};  
  
Duzxett::Duzxett(Noqte& A, Noqte& B)  
:Baslangic(A),  
Son(B)  
{  
}  
  
Duzxett::Duzxett(Duzxett& D)
```



OBYEKTŁƏRİN XÜSUSİYYƏTLƏRİ

```
.Baslangic(D.Baslangic),
Bitis(D.Son)
{}

float Duzxett::Uzunluq()
{ return Mesafe(Baslangic, Son); }

int Yaz(Duzxett& D)
{ printf("[");
  Yaz(D.Baslangic);
  printf("-");
  Yaz(D.Son);
  printf("]");
  return 0;
}
```

Yaz və **Mesafe** üzv funksiyası olmadıqları üçün istifadə edilərkən **Baslangic.Mesafe(Son);** və ya **Baslangic.Yaz(Son);** şəklində istifadə edilmədiyinə diqqət edin.

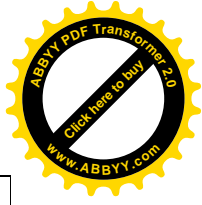
DOST2.CPP davamı

```
main()
{ clrscr();

  Noqte A, B(30, 40, 50);

  Yaz(A);
  printf("\n");
  Yaz(B);
  printf("\n\niki noqte arasindaki mesafe = %f\n\n", Mesafe(A, B));

  B.Dasi(10, -10, 60);
  Duzxett D(Noqte(10, 20, 30), B);
```



Etibar Seyidzadə

```
Yaz(D);
printf("\nnduz xettin uzunluq = %f\n\n", D.Uzunluq());

return 0;
}
```

Bu proqram icra olunması baxımından əvvəlkindən fərqlənir. Sadəcə proqramın yazılmasında istifadə olunan funksiyaların, istifadə olunma məntiqi müxtəlifdir. Bu cür təyinlər kitabxana yaratmaq üçün daha çox istifadə oluna bilən funksiyaların yazılmasına kömək edir.

Proqram çıxışı

```
(0.000000, 0.000000, 0.000000)
(30.00000, 40.00000, 50.00000)

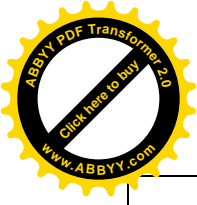
iki noqte arasindaki mesafe = 70.710678

[(10.00000, 20.00000, 30.00000)-(40.00000, 30.00000, 110.00000)]

duz xettin uzunluq = 86.023253
```

Bir sinfin başqa bir sinfi dost elan etməsi isə aşağıdakı kimi həyata keçirilir:

```
class A
{ friend class B;
  ...
  ...
  ...
```



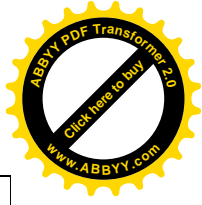
```
};  
  
class B  
{ ....  
  ....  
  ....  
};
```

Bir sinfin iki və daha artıq dost-sınıf və dost-funksiyası ola bilər. Dost-sınıf, dost-funksiya təyinlərinin **private**, **public** və ya **protected** səviyyələrində olmasının elə bir əhəmiyyəti yoxdur.

Dost-sınıf təyinlərinin məqsədi dost elan edilən sinfin (məsələn, **B** sinfi), dost elan edən sinfin (məsələn, **A** sinfi) bütün xüsusi üzvlərinə müraciət haqqını təmin etməkdir. Beləliklə, aralarında heç bir oxşarlığın olmamasına baxmayaraq bir sinfin digər sinifdən istifadə etməsi təmin olunmuş olur. Bu haqq sadəcə **friend** ilə təyin olunmuş siniflərə verilir. Bu sinifdən törənən siniflərin və ya bu sinfin dost elan etdiyi digər dost-sınıf və dost-funksiyaları eyni hüquqdan faydalana bilməzlər.

Dost-sınıf təyini birtərəfli təyindir. Yəni, **A B**-ni dost elan etmişdir, **B A**-nın xüsusiyyətlərini istifadə edə bilər. Lakin **A B**-nin xüsusiyyətlərini istifadə edə bilməz. Çünki, **B A**-nı dost elan etməmişdir.

Dost-funksiya təyini layihələndirici (**constructor**) və yoxedici (**destructor**) funksiyası ilə mənimsətmə "="



operatoru qarşısında təsirsizdir. Bu funksiyalar üzv funksiyalar olmaq məcburiyyətindədirlər.

Digər bir fərq də, dost funksiyaların dost olduqları bir sinfin üzvlərinə birbaşa müraciət etməmələridir.

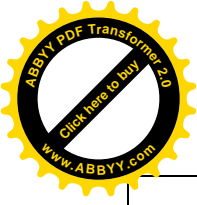
```
class Noqte  
{ float X, Y, Z;  
  
public:  
    Noqte(float, float, float);  
  
friend void Yaz(Noqte&);  
};
```

təyindən sonra

```
#include <stdio.h>  
  
void Yaz(Noqte&)  
{ printf("(%d, %d, %d)", X, Y, Z); }
```

kimi kodlaşdırıla bilməz. Ona görə ki, **Yaz** funksiyasının istifadə etdiyi **X**, **Y**, **Z** dəyişənlərinin nə olduqları bəlli deyildir. (Bizə görə bu dəyişənlər **Noqte** sinfinin dəyişənləridir. Lakin **Yaz** funksiyası bu dəyişənləri birbaşa istifadə edə bilməz. Əgər aşağıdakı kimi kodlaşdırma aparılırsa,

```
#include <stdio.h>  
  
void Yaz(Noqte& _noqte)
```

```
{ printf("(%d, %d, %d)", _noqte.X, _noqte.Y, _noqte.Z); }
```

X, Y, Z-in `_noqte` obyektinin üzvləri olduğu başa düşülməkdir; Bu üzvlərin `private` təyininin qarşılığı olaraq `Yaz` funksiyasının dost təyini bu xüsusi üzvlərə müraciəti təmin edir.

4.3 Obyektlərin Operatorlara Yüklənməsi

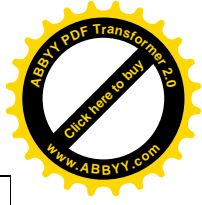
Təyin olunmuş yeni siniflərin operatorlara yüklənməsi mümkündür.

```
class Vector
{ public:
    float X, Y, Z;

    Vector(float = 0, float = 0, float = 0);
    Vector(Vector&);
};

Vector::Vector(float a, float b, float c)
{ X = a;
  Y = b;
  Z = c;
}

Vector::Vector(Vector& _vector)
{ X = _vector.X;
  Y = _vector.Y;
  Z = _vector.Z;
}
```



`Vector` sinfini təyin etdikdən sonra indi də iki vektorun cəmi və fərqi üçün `+` və `-` operatorlarını təyin edək.

```
Vector& operator+(Vector& A, Vector& B)
{ return Vector(A.X + B.X, A.Y + B.Y, A.Z + B.Z); }

Vector& operator-(Vector& A, Vector& B)
{ return Vector(A.X - B.X, A.Y - B.Y, A.Z - B.Z); }
```

Bu təyinlərdən sonra

```
Vector A(3, 4);
Vector B(8, 6.5 / 3, 1);
Vector C = A + B;
Vector D = A - B;
float f = 67;
Vector E = A - Vector(16, 1, -10) + Vector(f * 3, f / 4);
```

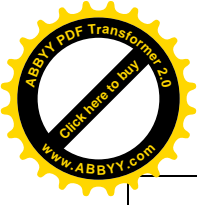
ifadələrini yazmaq olar.

Bir vektoru həqiqi ədədlə genişlədən vurma operatorunu aşağıdakı kimi yazı bilərik:

```
Vector& operator*(Vector& A, float R);
{ return Vector(A.X * R, A.Y * R, A.Z * R); }
```

Bu cür təyindən sonra

```
Vector A(4, 5, 3);
```



```
Vector B = A * 2;
```

doğru

```
Vector C = 2 * A;
```

isə səhvdir. Çünki, `Vector& operator*(Vector&, float);` ilə `Vector` ilə `float` qiymətinin hasili təyin edilmişdir. `float` ilə `Vector` qiymətlərinin hasili ayrılıqda təyin edilməlidir.

```
inline Vector& operator*(float R, Vector& A)
{ return A * R; }
```

Bir vektorun istiqamətini tərs çevirən `unary` operatorunu təyin etmək üçün

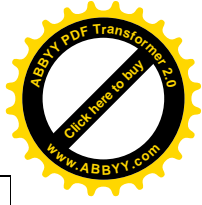
```
Vector& operator-(Vector& A)
{ return Vector(-A.X, -A.Y, -A.Z); }
```

yazmaq olar.

Bir sinfin xüsusi (`private`) üzvlərinin də operator funksiyaları daxilində istifadə olunması tələb olunarsa, bu operator funksiyalarının dost (`friend`) kimi təyin olunması lazımdır.

```
class Vector
{ float X, Y, Z;

public:
```



```
Vector(float = 0, float = 0, float = 0);
Vector(Vector&);
```

```
friend Vector& operator+(Vector&, Vector&);
friend Vector& operator-(Vector&, Vector&);
friend Vector& operator*(Vector&, Vector&);
friend Vector& operator*(Vector&, float);
friend Vector& operator*(float, Vector&);
friend Vector& operator-(Vector&);
};
```

```
Vector& operator+(Vector& A, Vector& B)
{ return Vector(A.X + B.X, A.Y + B.Y, A.Z + B.Z); }
```

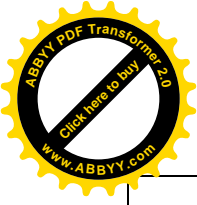
Operatorların ilk parametrlərinin sinfin özünün olması halında onları üzv funksiya kimi təyin edib istifadə etmək də mümkündür.

```
class Vector
{ float X, Y, Z;

public:
    Vector(float = 0, float = 0, float = 0);
    Vector(Vector&);

    Vector& operator+(Vector&);
    Vector& operator-(Vector&);
    Vector& operator*(Vector&);
    Vector& operator*(float);
    friend Vector& operator*(float, Vector&);
    Vector& operator-();
};

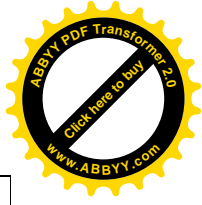
Vector& Vector::operator+(Vector& B)
{ return Vector(X + B.X, Y + B.Y, Z + B.Z); }
```



```
Vector& Vector::operator-()  
{ return Vector(-X, -Y, -Z); }
```

4.4 this Lokal Dəyişkəni

```
//THISNKT.CPP  
#include <stdio.h>  
  
class Nöqte  
{ float X, Y, Z;  
public:  
    Nöqte(float, float, float);  
  
    friend void Yaz(Nöqte&);  
    void Hesab();  
};  
  
Nöqte::Nöqte(float x, float y, float z)  
{ X = x;  
  Y = y;  
  Z = z;  
}  
  
void Nöqte::Hesab()  
{ printf("Bu nöqte ");  
  Yaz(Nöqte(X, Y, Z));  
  printf(" mövqeyindedir\n");  
}  
  
void Yaz(Nöqte& _nöqte)  
{ printf("(%f, %f, %f)", _nöqte.X, _nöqte.Y, _nöqte.Z); }  
  
main()  
{ Nöqte A(1, 2, 3);  
  A.Hesab();
```



```
return 0;  
}
```

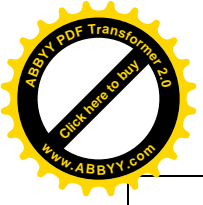
Bu misalda bundan əvvəlki misala əlavə olaraq **Hesab** üzv funksiyasından istifadə edilmişdir. **Yaz** funksiyası yalnız üzvün koordinatlarını, **Hesab** funksiyası isə "**Bu nöqte (x, y, z) mövqeyindedir**" məlumatını ekrana çıxaracaqdır. **Hesab** funksiyası nöqtənin koordinatlarını ekrana çıxararkən **Yaz** funksiyasını çağırmalı, çağırarkən də yazılacaq nöqtəni parametr olaraq göndərməlidir.

Burada **Yaz** funksiyasına parametr vermək üçün oxşar bir obyekt yaradılır. Bu əməliyyat vaxt və əlavə yaddaş tələb edir. Buna baxmayaraq layihələndiricilər hər zaman oxşar obyekt yarada bilmirlər.

Lakin **Hesab** və buna oxşar bütün funksiyalar hansı obyektə aid olduqları haqqında məlumata malikdirlər. Bu məlumat üzv funksiyasının aid olduğu obyektin göstəricisi məlumatından ibarətdir. Bu göstəricinin adı hər bir obyekt üçün **this** qəbul olunur. Bu dəyişkən üzv funksiyaları üçün avtomatik olaraq təyin olunur. Baxdığımız misalda **Hesab** funksiyası üçün bu təyin **Nöqte *const this** şəklindədir.

İstifadəsi aşağıdakı kimidir:

```
void Nöqte::Hesab()  
{ printf("Bu nöqte ");
```



```
Yaz(*this);  
printf(" movqeyindedir\n");  
}
```

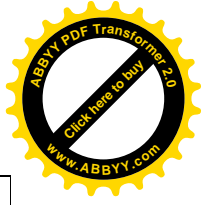
this göstəricisindən istifadə edərək obyektin üzv funksiya və dəyişənlərinə müraciət etmək də mümkündür. Hətta, bəzən məcburidir. Üzv funksiya daxilində obyektinlə eyni adda olan dəyişənlər varsa, bu funksiya obyektin eyni adlı dəyişənlərinə müraciət edə bilməz. Bu halda görmə (**scope::**) operatorundan da istifadə etmək olmaz. Belə hallarda **this** dəyişənindən istifadə edilir.

```
Noqe::Noqe(float X, float Y, float Z)  
{ this->X = X;  
  this->Y = Y;  
  this->Z = Z;  
}
```

4.5 Ümumi Ortaq Dəyişənlər

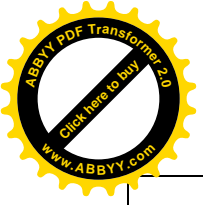
Hər obyekt malik olduğu məlumatları bütün obyektlərdən qoruyur. Əgər bəzi məlumatların bütün proqramlaşdırma üzvləri tərəfindən sərbəst istifadə edilməsi tələb olunarsa, bu dəyişənlər global dəyişənlər kimi təyin edilir.

Əgər bir dəyişənin müəyyən bir sinfə mənsub olan obyektlər tərəfindən istifadə edilməsi tələb olunarsa, bu



istək sinif təyini daxilində bildirilir. Bu dəyişənin təyini zamanı tipdən əvvəl **static** sözü əlavə edilərək yerinə yetirilir.

```
//STATIC1.CPP  
  
#include <conio.h>  
#include <stdio.h>  
  
class Misal  
{ int a;  
  static int b;  
  
public:  
  Misal(int a, int b)  
  { this->a = a;  
    this->b = b;  
  }  
  
  int De_a()  
  { return a; }  
  int De_b()  
  { return b; }  
};  
  
int Misal::b;  
main()  
{ clrscr();  
  Misal A(5, 10), B(8, 15);  
  printf("A a = %d, b = %d\n", A.De_a(), A.De_b());  
  printf("B a = %d, b = %d\n", B.De_a(), B.De_b());  
  printf("A a = %d, b = %d\n", A.De_a(), A.De_b());  
  return 0;  
}
```



Proqram çıxışı

```
A a = 5, b = 15;  
B a = 8, b = 15;  
A a = 5, b = 15;
```

Bu təyin ilə **b** dəyişkəninin bütün **Misal** obyektlərində ortaq istifadə ediləcəyi, digər obyektlərin isə bu dəyişkənə birbaşa müraciət edə bilməyəcəyi bildirilir. Lakin bu təyin kifayət deyildir. **b** dəyişkəni bundan başqa sinif xaricində də

```
int Misal::b;
```

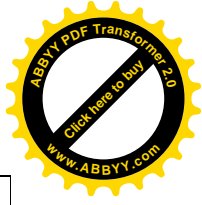
şəklində təyin edilməlidir.

Bir qayda olaraq dəyişkənlərin təyini aşağıdakı kimidir:

```
class sinif_adı  
{ static tip1 dəyişkən1;  
  static tip2 dəyişkən2;  
  ...  
};  
  
tip1 sinif_adı::dəyişkən1;  
tip2 sinif_adı::dəyişkən2;  
...
```

Əgər lazım gələrsə, bu dəyişkənlərə başlanğıc qiymət də mənimlətmək olar.

```
int Misal::b = 30;
```

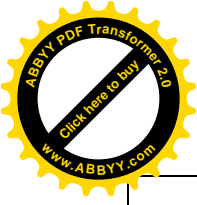


Obyektlər üçün istifadə olunan **static** açar sözü **C**-dəki lokal dəyişkənlər ilə eyni məntiqə malikdir. Yəni obyekt, proqram icra olunmağa başladığı zaman varlığını ortaya qoyur: yaddaşdan yer istəyir və ilk qiymətini alır. Proqram icrasını tamamladıqdan sonra varlığına son qoyulur. Lokal təyinlər kimi funksiya icra olunmağa başladığı anda varlığını göstərib, icrasını tamamladıqdan sonra varlığına son vermir. Eyni formada üzv dəyişkənləri də obyekt mövcud olduğu anda (layihələndirici icra olunduqdan sonra) varlığını göstərib, obyekt yox olduğu anda (yoxedici icra olunduqdan sonra) varlıqlarına son qoyulur. **static** ilə təyin edilən bu üzvlər də eyni şəkildə obyektlərin ömür müddətindən asılı qalmayaraq, proqramın ömür müddətindən asılı hala gəlirlər.

static dəyişkənlər ola bildiyi kimi, **static** obyektlərin olaması da mümkündür.

```
int Mesafe(float a)  
{ static Nöqtə A(3, 7, -12);  
  ...  
}
```

Yalnız burada diqqət edilməsi vacib olan hal, hər hansı bir obyekt mövcud olmazdan əvvəl, **static** obyektlər mövcud olacağı üçün bu obyektlər yaradılarkən mövcud olmayan dəyişkənlərdən istifadə olunmamalıdır.



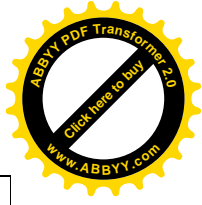
```
int Mesafe(float a)
{ static Nöqte A(a, 7, -a); //Sehv. a-nin qiymeti yoxdur.
  ...

  Nöqte B(a, 7, -a);          //Doğru. B static deyildir.
  ...
}
```

4.6 Statik (Static) Funksiyalar

Üzv dəyişənlər kimi üzv funksiyalar da **static** olaraq təyin edilə bilərlər. Bu təyin baxımından üzv funksiyalarını iki müxtəlif qrupa ayırmaq olar: statik (**static**) və avtomatik (**auto**) funksiyalar. Normal olaraq hər hansı bir təyin olmazsa, təyin olunmayan bütün üzv funksiyaları avtomatik funksiya kimi qəbul edilir. Avtomatik funksiyaların əsas xüsusiyyəti, üzv olduqları sinfin bütün üzv dəyişən və funksiyalarını birbaşa istifadə edə bilmələridir. Virtual funksiyalar da daxil olmaqla, indiyə kimi təyin edib istifadə etdiyimiz bütün üzv funksiyalarının avtomatik funksiya olduğunu xatırlayaraq, yazılarkən üzv dəyişən və funksiyalarını necə istifadə etdiklərinə diqqət edin.

Statik funksiyalar da üzv funksiyası kimi təyin edilib istifadə olunmalarına baxmayaraq avtomatik funksiyalardan iki əlamətinə görə fərqlənirlər. Birincisi, statik funksiyalar üzvü olduqları sinfin üzv dəyişən və



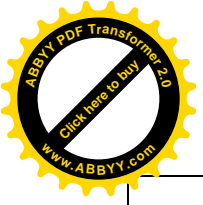
funksiyalarından avtomatik olanları birbaşa istifadə edə bilməzlər. İkincisi, statik funksiyaları çağırmaq üçün bir obyektə ehtiyac vardır.

Statik funksiyalar sinif daxilə təyin edilərkən funksiya prototipinin əvvəlinə **static** açar sözü əlavə edilir. Funksiyanın gövdəsinin yazılması isə digər funksiyalar kimidir. Yalnız sinfin üzv dəyişən və funksiyalarından statik kimi təyin edilməyənlər birbaşa istifadə oluna bilməzlər. Bundan başqa **this** lokal dəyişəni də funksiyaların daxilində istifadə oluna bilməz.

```
class sinif_adı
{ ...
  static funksiya_tipi funksiya_adı(parametr_siyahısı);
  ...
};

{ ...
  funksiya_tipi sinif_adı::funksiya_adı(parametr_siyahısı);
  ...
  ...
}
```

Statik funksiyalar iki formada çağırıla bilər: Əgər statik funksiyanın aid olduğu sinfin bir obyekt mövcuddursa, bu obyektə əsaslanaraq bir statik funksiya sanki, avtomatik bir funksiya kimi çağırıla bilər. Və ya statik funksiyanın adından əvvəl aid olduğu sinfin adını



və :: görmə (**scope**) operatorunu yazaraq, statik funksiyanı çağırmaq mümkündür.

```
//STATIC.CPP

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

class Static_Misal
{ int X;
  static int Y;

  public:
    Static_Misal(int);

    int X_Qiymeti();

    void Auto_Yaz();
    static void Static_Yaz();
    static void Static_Message();
};

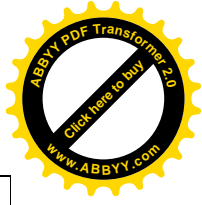
int Static_Misal::Y = 300;

Static_Misal::Static_Misal(int x)
{ X = x; }

int Static_Misal::X_Qiymeti()
{ return X; }

//avtomatik funksiyanın yazılması

void Static_Misal::Auto_Yaz()
{ printf("x = %d\t", X);
  printf("*** %d **\n", X_Qiymeti());
  printf("\ny = %d\n", Y);
```



```
//Static funksiya avtomatik funksiyaadan cagrilir

  Static_Message();
}

//statik funksiyanın yazılması

void Static_Misal::Static_Yaz()
{ /*

  printf("x = %d\t", X);

  //Error: Member X cannot be used without an object
  //Sehv: X üzvü obyekt olmadan istifadə oluna bilməz

  printf("*** %d **\n", X_Misal());

  //Error: Use . or -> to call 'Static_Misal::X_Qiymeti()'
  //Sehv: 'Static_Misal::X_Qiymeti()' çağırılarkən
  //. və ya -> istifadə edin
  //Qeyd: Bunun üçün də yeni bir obyektdə ehtiyac vardır.

  */

  printf("\ny = %d\n", Y);

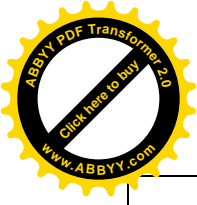
  //Static funksiyası static funksiyasından çağırılır

  Static_Message();
}

void Static_Misal::Static_Message()
{ printf("\nStatic_Misal, Static_Message funksiyaları\n"); }

main()
{ clrscr();

  Static_Misal A(8);
```



```
A.Auto_Yaz();
A.Static_Yaz();

printf("\n");

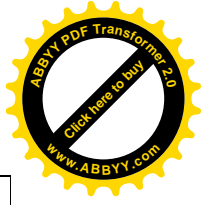
//Static_Misal::Auto_Yaz();
//Error: Use . or -> to call 'Static_Misal::Auto_Yaz()'
//Sehv: 'Static_Misal::Auto_Yaz()' cagrilarken
//. ve ya -> istifade edin

Static_Misal::Static_Yaz();

return 0;
}
```

Program çıxışı

```
x = 8  ** 8 **
y = 300
Static_Misal, Static_Message funksiyasi
y = 300
Static_Misal, Static_Message funksiyasi
y = 300
Static_Misal, Static_Message funksiyasi
```



4.7 const Funksiyaları

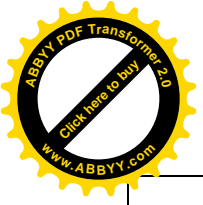
Bir obyektdən bir məlumatı almaq üçün istifadə olunan üzv funksiyalarının, yazılarkən səhvən də olsa, üzv dəyişkənlərini dəyişdirməsi arzuolunmazdır. Daxilində bu cür funksiyaların təyin edilməsi zamanı bu istək parametr siyahısından sonra **const** (sabit) açar sözü yazılaraq bildirilir. Təyin edilərkən sonunda **const** (sabit) açar sözü olan funksiyalar *məlumat funksiyası* adlandırılır. Bu funksiyalar üzv dəyişkənlərinin qiymətlərini dəyişdirə bilmədikləri kimi, funksiyası olmayan digər funksiyaları da çağıra bilməzlər.

```
class sinif_adı
{
    ...
    int const_funksiyası(parametr_siyahısı) const;
    ...
}

int sinif_adı::const_funksiyası(parametr_siyahısı) const
{
    ...
    return qiymət;
}
```

4.8 İç-içə Təyinlər

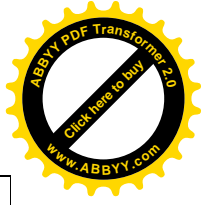
Sinfin təyin edilməsi zamanı yeni bir sinif və ya tip də təyin etmək olar. Təyin edilən bu yeni tip, təyin olunduğu sinfin adı ilə bərabər istifadə edilir. Bu da təyin



olunduğu sinfin adından sonra görmə (:: scope) operatoru və adın yazılması ilə yerinə yetirilir.

```
sinif_adı::tip_adı  
sinif_adı::enum_qiyməti
```

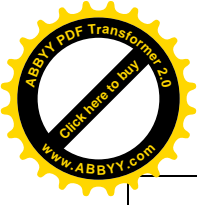
```
//ARRAYNST.CPP  
  
#include <conio.h>  
  
#include <stdio.h>  
  
struct Uzv  
{ int X;  
  char *Ad;  
  
  Uzv(char* ad = "XxXxX", int a = 0)  
  { Ad = ad; X = a; }  
  
  void Yaz()  
  { printf("::Uzv %s %d\n", Ad, X); }  
};  
  
class Massiv  
{ public:  
  
  struct Uzv  
  { int X, Y;  
  
    Uzv(int a = 0, int b = 0)  
    { X = a; Y = b; }  
    void Yaz()  
    { printf("Massiv::Uzv %d %d\n", X, Y); }  
  };  
  
  Uzv A[10]; //Massiv daxilinde teyin olunan Uzv obyektı  
  
  ::Uzv B[10];
```



```
//Massiv xaricinde teyin olunan Uzv obyektı  
//Her iki teyinetmeye diqqet edin  
public:  
  Massiv(char *, int, int);  
};  
  
Massiv::Massiv(char* n, int t, int s)  
{ for(int i = 0; i < 10; i++)  
  { A[i].X = t;  
    A[i].Y = s;  
    B[i].Ad = n;  
    B[i].X = s;  
  }  
}  
  
main()  
{ clrscr();  
  Massiv::Uzv K;  
  K.Yaz();  
  //K Massiv daxilinde teyin edilen obyektıdır.  
  
  Uzv H;  
  H.Yaz();  
  //H Massiv xaricinde teyin edilen obyektıdır.  
  //Burada ::Uzv H seklinde istifade edile bilər.  
  
  return 0;  
}
```

Proqram çıxışı

```
Massiv::Uzv 0 0  
::Uzv XxXxX 0
```



4.9 Obyekt Göstəriciləri

Bir obyektin göstərici kimi təyin olunması C-də olduğu kimidir:

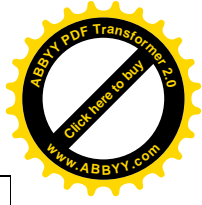
```
Massiv *A;  
Noqte *C;  
Noqte **D;
```

Bu cür təyinlərdə obyekt deyil, sadəcə, obyekt göstərən bir göstərici vardır. Buna görə də təyin zamanı layihələndiricinin, proqram bloku tamamlandıqdan sonra da yoxedicinin icra olunacağını gözləmək olmaz. Bu istək proqramçı tərəfindən təyin olunur.

```
main()  
{ Massiv massiv(20, 10);  
  Massiv *massivPtr = &massiv;  
  ...  
  ...  
  
  return 0;  
} //massiv obyektı özü yox olur.
```

Əgər obyekt göstəriciləri üçün yaddaşda dinamik olaraq bir yer ayrılırsa, bunun üçün **new** və ayrılmış bu obyektin silinməsi üçün də **delete** operatorunun istifadə edilməsi lazımdır.

```
main()  
{ Massiv *massiv = Massiv(10, 20);
```

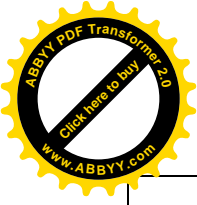


```
Massiv *massivPtr;  
massivPtr = new Massiv(30, 40);  
...  
delete massivPtr;  
...  
...  
int PaketSayi = 30;  
int PaketUzunlugu = 6*;  
massivPtr = new Massiv(PaketSatisi *PaketUzunlugu, 0);  
...  
...  
delete massivPtr;  
delete massiv;  
  
return 0;  
}
```

new operatoru yer ayırdığı obyektin layihələndiricisini, **delete** operatoru da bu obyektin yoxedicisini avtomatik çalışdırır. Buna görə də **new** operatoru istifadə edilərkən obyektin layihələndiricilərindən biri **new** operatorundan sonra yazılır. **delete** operatorunun istifadə edilməsində isə sadəcə obyekt göstəricisi verilir. Burada **new** ilə yer ayrılmamış obyektləri **delete** ilə, **new** ilə yer ayrılmış olsalar belə, eyni obyektı bir dəfədən artıq silmək olmaz.

Massiv massiv(20, 30);	Massiv* massivPtr = new Massiv(20, 30);
...	...
delete massiv;	delete massivPtr;
	delete massivPtr;

Yuxarıda göstərilənlərin hər biri səhvə yol açar.



4.10 Obyekt Massivi

Standart (default) layihələndiriciyə malik olmaq şərti ilə bir sinfin obyektlərindən ibarət olan massiv əldə etmək mümkündür.

```
//MASARRAY.CPP

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

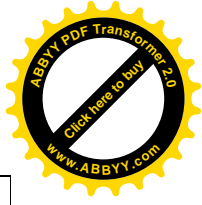
class Massiv
{ public:
    int Olcu;
    float *f;

    Massiv(int n = 50);
    ~Massiv();
};

Massiv::Massiv(int n)
{ static say = 0;
  f = new float(Olcu = n);
  printf("%d\tolcu %d\n", ++say, Olcu);
}

Massiv::~~Massiv()
{ printf("Yoxedici\tolcu = %d\n", Olcu);
  delete f;
}

main()
{ clrscr();
  Massiv *A = new Massiv(20);
    //20 elementlik bir Massiv gostericisi
  Massiv *B = new Massiv;
    //50 elementlik bir Massiv gostericisi
```



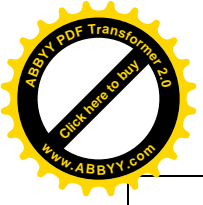
```
Massiv *C = new Massiv[10];
    //50 elementden ibarət Massivlərin massivi

*(A -> f + 5) = 1.00;
    //A-nin gosterdiyi Massivin 5-ci elementi
*(B -> f + 5) = 2.00;
    //B-nin gosterdiyi Massivin 5-ci elementi
*((C + 8) -> f + 5) = 3.00;
    //C-nin gosterdiyi massivlərin 8-ci sirasındakı
    //Massivin 5-ci elementi
*(C[8].f + 5) = 4.00;
    //C-nin gosterdiyi massivlərin 8-ci sirasındakı
    //Massivin 5-ci elementi
delete A;
    //Bir Massiv obyektini silinir. (20 elementli)
delete B;
    //Bir Massiv obyektini silinir. (50 elementli)
delete []C;
    //10 Massiv obyektini silinir. (Her biri 50 elementli)

return 0;
}
```

Burada `delete []C;` yerinə `delete [10]C;` kimi `C` massivinin ölçüsü də yazıla bilər. Lakin bu əmr massivi tamamilə sildiği üçün mötərizələrin daxilində verilən massivin ölçüsünü nəzərə almayacaqdır. Və kompilyator bunu Sizə “Array size for ‘delete’ ignored” məlumatı ilə bildirəcəkdir. Bu səhvdən qurtulmaq üçün “`#pragma warn dsz-`” direktivindən istifadə edə bilərsiniz.

Burada `delete []C;` yerinə `delete C;` əmri istifadə edilərsə, `C`-nin ilk elementi silinəcəkdir ki, bu da xoşagəlməz halların yaranmasına səbəb olacaqdır. Buna



görə də **delete** əmrindən bu şəkildə istifadə etmək məqsədəuyğun deyildir.

Əgər istifadə etmək istədiyiniz obyektin standart layihələndiricisi yoxdursa və ya yaradılacaq obyektlər bir-birindən fərqli xüsusiyyətlər daşıyaqlarsa, bu halda obyekt göstəriciləri massivindən istifadə etmək məqsədəuyğundur.

```
//MASPTRAR.CPP

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

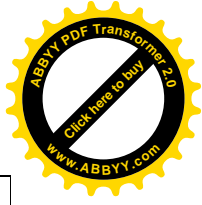
class Massiv
{ public:
    int Olcu;
    float *f;

    Massiv(int n = 50);
    ~Massiv();
};

Massiv::Massiv(int n)
{ static say = 0;
  f = new float(Olcu = n);
  printf("%d\tolcu %d\n", ++say, Olcu);
}

Massiv::~~Massiv()
{ printf("Yoxedicil\tolcu = %d\n", Olcu);
  delete f;
}

#define MasOlcu 10
```



```
main()
{ clrscr();
  int i;
  //Gostericiler ucun yer ayirma
  Massiv **A = new Massiv*[MasOlcu];

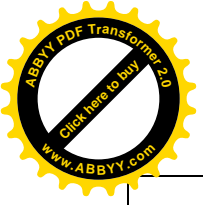
  //Her bir gostericinin gosterdiyi obyekt yaratmaq ve menimsetmek
  for(i = 0; i < MasOlcu; i++)
    A[i] = new Massiv(i * 2 + 10);

  //Istifade edilmesi
  *(A[2] -> f + 5) = 1.0;

  //Her bir obyektin bir-bir silme emeliyyati
  for(i = MasOlcu - 1; i >= 0; i--) delete A[i];

  //Gosterici sahelerini silme emeliyyati
  delete []A;

  return 0;
}
```



V FƏSİL

OBYEKT TÖRƏTMƏK

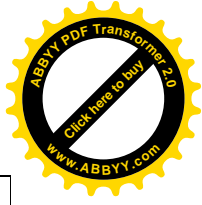
5.1. Törətmə Əməliyyatı

Bir sinfə əsaslanaraq onun yerinə yetirdiyi işin mahiyyətini dəyişdirmək və ya inkişaf etdirmək kimi icra olunan əməliyyatlara törətmə (**derivation**) deyilir. Törətmə əməliyyatının yerinə yetirilməsi üçün təyin olunmuş bir sinif mövcud olmalıdır. Törətmə əməliyyatında istifadə olunan bu mövcud sinfə baza sinfi (**base class**) deyilir. Törətmə nəticəsində meydana gələn obyektə isə törənmiş sinif (**derivated class**) adı verilir. Başqa sözlə baza sinfinə valideyn (**parent**), törənmiş sinfə isə uşaq (**child**) adı verilir.

Aşağıda göstərilmiş iki metodla sinif törətmək olar:

1. Bir sinfin üzv funksiyalarından birinin vəzifələrini başqa bir formada yerinə yetirəcək şəkildə dəyişdirməklə;
2. Obyektin vəzifələrini artıraraq yeni üzv funksiyaları əlavə etmək, yəni inkişaf etdirməklə.

Obyekt törədərkən bu iki metodun hər ikisi ayrı-ayrılıqda və ya bərabər istifadə oluna bilər.



5.2 Siniflərin Törədilməsi

```
//MESAJ.CPP

#include <stdio.h>
#include <string.h>

class Mesaj
{ char *mesaj[5];

public:
    Mesaj();
    ~Mesaj();

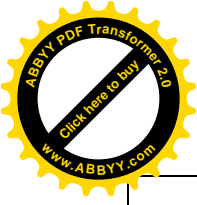
    int Deyisdir(int, char* = "");
    int Xeber(int);
};

Mesaj::Mesaj()
{ int i;

    for(i = 0; i < 5; i++)
    { mesaj[i] = new char[1];
      strcpy(mesaj[i], "");
    }
}

Mesaj::~Mesaj()
{ int i;
  for(i = 0; i < 5; i++) delete mesaj[i];
}

int Mesaj::Deyisdir(int i, char *mes)
{ if(i < 0 || i > 5)
  return -1;
  delete mesaj[--i];
  mesaj[i] = new char[strlen(mes) + 1];
```



```
strcpy(mesaj[i], mes);
return 0;
}

int Mesaj::Xeber(int n)
{ if(n < 0 || n > 5)
  { printf("\nDaxili sehv: Xeberdarliq mesajinin nomresi sehvdır\n");
    return -1;
  }

  printf("%s\n", mesaj[n-1]);
  return 0;
}
```

Bu siniflərin tətbiqi nəticəsində aşağıdakı proqramı yazaraq növbəti nəticəni əldə etmək olar:

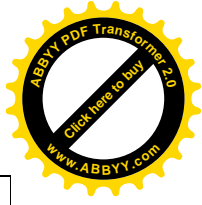
```
//MMESAJ.CPP

#include <conio.h>
#include "mesaj.cpp"

Mesaj Xeberd;

main()
{ clrscr();
  Xeberd.Deyisdir(1, "\nYoxlama xeberdarligi");
  Xeberd.Deyisdir(2, "\nCixis xeberdarligi");
  Xeberd.Xeber(1);
  Xeberd.Xeber(21);
  Xeberd.Xeber(2);
  Xeberd.Xeber(1);
  return 0;
}
```

Proqram çıxışı



Yoxlama xeberdarligi

Daxili sehv: Xeberdarliq mesajinin nomresi sehvdır

Cixis xeberdarligi

Yoxlama xeberdarligi

İndi isə xəbərdarlıq mesajının nömrəsi tək isə, proqram **Enter** düyməsini sıxana qədər gözləsin, cüt isə gözləmədən icrasını tamamlasın. **Xeber** hissəsi istisna olmaqla bu strukturun digər funksiyaları eyni qalsın. Bu halda **Mesaj** sinfini baza kimi qəbul edib **Sehv** sinfini törətmək daha məqsədəuyğundur.

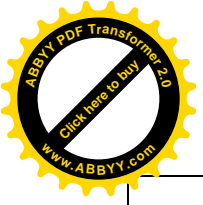
```
//SEHV.CPP

#include "mesaj.cpp"
#include <conio.h>
#include <stdlib.h>

struct Sehv : Mesaj
{ Sehv() { }
  int Xeber(int);
};

int Sehv::Xeber(int N)
{ int Err = Mesaj::Xeber(N);

  if(N % 2)
  { printf("\nDavam etmək ucun ENTER duymesini sixin.");
    while(getch() != 13);
    printf("\n");
  }
  else
```



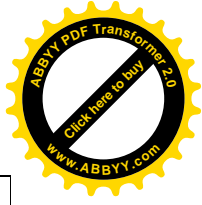
```
{ printf("\nProqram icrasini davam etdire bilmir.\n");
  exit(N);
}

return Err;
}
```

Yeni sinfin təyin edilməsi zamanı baza sinfinin əsas sinif adından sonra yazıldığına diqqət edin. `class Sehv : Mesaj` təyini `Sehv` sinfinin `Mesaj` sinfindən törəndiyini göstərir. Digər bir halda əgər lazım gələrsə, yeni sinfin layihələndirici və yoxedici funksiyaları digər üzv funksiyalarından fərqli olaraq törənmiş olduqları sinfin layihələndirici və ya yoxedici funksiyalarını avtomatik olaraq istifadəyə verirlər.

`Sehv` sinfi üçün `Xeber` funksiyası yenidən yazılarkən `Mesaj` sinfinin `Xeber` funksiyasına ehtiyacı olduğu zaman, bu ehtiyac `Mesaj` sinfinin `Xeber` funksiyasının `Mesaj::Xeber` şəkildə çağırılması ilə ödənilmiş olur. Bu çağırışı yerinə yetirmək vacib deyildir. Eyni formada “uşaq” sinfinin “valideyn” sinfindən olan bir funksiyayı çağırması əsasında çağırılan funksiyanın adı öz funksiyalarının adı ilə üst-üstə düşürsə, bu funksiyanı əsas sinfin “valideyn”lərindən olmaq şərti ilə `::` (görmə) operatorundan istifadə edərək çağırmaq mümkündür. `Mesaj::Deyisdir()` kimi.

İndi də bundan əvvəlki misalı `Sehv` sinfi ilə yenidən yazaq:



```
//MSEHV.CPP

#include <conio.h>
#include "sehv.cpp"

Sehv Xeberd;

main()
{ clrscr();
  Xeberd.Deyisdir(1, "\nYaxlama xeberdarligi");
  Xeberd.Deyisdir(2, "\nCixis xeberdarligi");
  Xeberd.Xeber(1);
  Xeberd.Xeber(21);
  Xeberd.Xeber(2);
  Xeberd.Xeber(1);
  return 0;
}
```

Proqram çıxışı

```
Yaxlama xeberdarligi

Davam etmek ucun ENTER duymesini sixin.

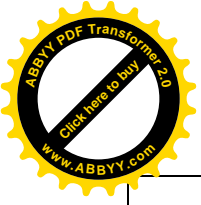
Daxili sehv: Xeberdarliq mesajinin nomresi sehvdır

Davam etmek ucun ENTER duymesini sixin.

Cixis xeberdarligi

Proqramin icrasi tamamlandı.
```

Bu misalda `Sehv` sinfi üçün `Deyisdir` funksiyasının varmış kimi istifadə edilməsinə diqqət edin. Həqiqətdə isə `Sehv` sinfi `Deyisdir` vəzifəsini `Mesaj` sinfinin tətbiq



etdiyi metoddan istifadə edərək yerinə yetirir. **Xeber** vəzifəsi olduğu zaman isə **Sehv** bunu öz metodlarına görə yerinə yetirir. **Sehv** vəzifəsini yerinə yetirərkən **Mesaj**-dan da faydalana bilir.

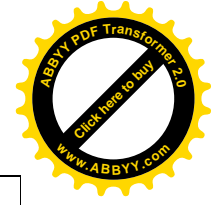
5.3 Müraciət Haqları və Nüfuz Etmə

Bir sinfin üzvlərinə (dəyişkən və metodlarına) müraciət edə bilən xarici agentlər üç sinfə bölünür. Bir törədilmiş sinif, baza sinfi daxilindəki qorunmuş (**protected**) və təbii ki, ümumi (**public**) kimi təyin edilmiş üzvlərə müraciət hüququna malikdir.

Törədilmiş sinif baza qəbul edilərək yeni bir obyektin törədilməsi tələb olunduğunda, son törədilən sinif ilk törədilən sinfə bu sinfin müəyyən etdiyi hüquqlar ilə müraciət edə biləcəkdir. Lakin bu halda əsas bazanın üzvlərinə müraciət hüququ necə olacaqdır?

Burada törədilən sinfin, özündən törənən siniflərə əvvəlki siniflərdən qalan mirası necə təhvil verəcəyini müəyyən etmək lazımdır. Bu o sinfin nüfuz etmə qabiliyyətini müəyyən edir.

```
class A { ... };
class B : public A { ... };
class C : private A { ... };
class D : A { ... };
class E : A { ... };
```



Burada **A** təyin edilmiş baza sinfidir. **B**, **C**, **D** və **E** isə **A**-dan törənmiş siniflərdir. **B**-nin təyin edilməsi zamanı **A**-nın əvvəlindəki **public** ifadəsi **B**-nin **A**-dan təhvil aldığı mirası eyni hüquqlarla (**A**-nın **B**-yə verdiyi hüquqlarla) özündən sonrakılara təhvil verməsi mənasına gəlir.

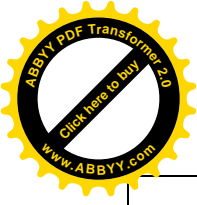
C də **A**-dan törənməsinə rəğmən onun təyin edilməsi zamanı **private** ifadəsindən istifadə edildiyi üçün **C** **A**-dan aldığı bütün xüsusiyyətləri özündən törənən siniflərə **private** kimi təhvil verəcəkdir. Burada **A**-nın **public** xüsusiyyətlərinin belə **B**-dən törənənlər üçün istifadə edilməsinin qadağan olmasına diqqət edin.

Nüfuz etmə public	
public	public
protected	protected
private	private
Nüfuz etmə private	
public	private
protected	private
private	private

D və **E** siniflərinin təyin edilməsi aşağıdakı kimidir:

```
class D : public A { ... };
class E : private A { ... };
```

Buna səbəb, əvvəlcədən qeyd edildiyi kimi **class** vasitəsilə edilən təyintlərdə müraciət hüququ müəyyən edilmədiyi zaman **private**, **struct** vasitəsilə edilən



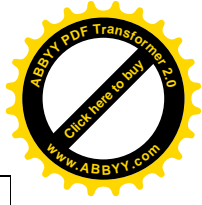
təyinlərdə isə **public** olduğunun avtomatik qəbul edilməsidir.

5.4 Dinamik Yükləmə

Baza sinfi və törədilmiş sinif arasındakı əhəmiyyətli olan əlaqələrdən biri də baza sinfinin göstəricisinə törədilmiş sinfin ünvanının mənimsədilə bilməsidir. Bu dinamik yükləmə (**dynamic binding**) qaydasının meydana gəlməsini təmin edən bir hadisədir. Lakin bu mənimsətmə dinamik yükləmənin meydana gəlməsi üçün kifayət deyildir. Bunu bir misal ilə izah edək:

```
//CLASSPOI.H  
  
#include <stdio.h>  
#include <conio.h>  
  
class Baza  
{ protected:  
    int X, Y;  
  
public:  
    Baza(int C)  
    { X = C; Y = 2 * C; }  
  
    void Goster()  
    { printf("Baza X = %d\tY = %d\n", X, Y); }  
};  
  
class Toremis:public Baza  
{ public:  
    int Z;  

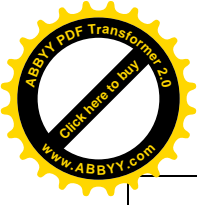
```



```
Toremis(int e) : Baza(e)  
{ Z = X + Y; }  
  
void Goster()  
{ printf("Toremis X = %d\tY = %d\tZ = %d\n", X, Y, Z); }  
};  
  
main()  
{ clrscr();  
  
    Baza A(1);  
    Toremis B(2);  
    Baza *C;  
  
    printf("A.Goster\n\t");  
    A.Goster();  
    printf("B.Goster\n\t");  
    B.Goster();  
  
    printf("C = &A\nC->Goster\n\t");  
    C = &A;  
    C -> Goster();  
  
    printf("C = &B\nC->Goster\n\t");  
    C = &B;  
    C -> Goster();  
  
    return 0;  
}
```

Program çıxışı

```
A.Goster  
    Baza X = 1          Y = 2  
B.Goster  
    Toremis X = 2      Y = 4      Z = 6  
C = &A  
C->Goster
```

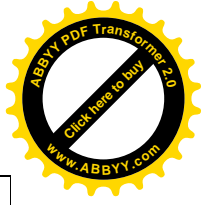


```
Baza X = 1      Y = 2
C = &B
C->Goster
Baza X = 2      Y = 4
```

Bu misalda törədilən sinfin daxilində baza sinfinin üzvü olan `Goster()` funksiyası yenidən təyin olunmuşdur. Bu təyin doğrudur (`B` obyektinin `Goster()` xüsusiyyətinin olmasına diqqət edin). Belə ki, baza sinfinin göstəricisinə törədilmiş sinif mənimsədildikdən sonra göstəricidən `Goster()` xüsusiyyətini verməsi tələb olunarsa, bu ancaq törədilmiş sinfin obyektinin məlumatları ilə baza sinfinin imkanlarının ortaya qoyulması ola bilər. Bu hal kompilyatorun kompilyasiya zamanı davranışlarından qaynaqlanır.

Bunun qarşısını almağın ən yaxşı yolu baza sinfi təyin edilərkən törədilən siniflərin dəyişdirəcəkləri metodları xəyali (`virtual`) təyin etməkdir. Bu səbəbdən də bu cür metodların prototip təyin etmələrinə `virtual` açarsözü ilə başlamaq lazımdır.

Buna görə də lazım olan səmərəliliyi əldə etmək üçün yuxarıdakı misalda `Baza` sinfinin üzv funksiyası olan `void Goster()` funksiyası `virtual` sözü ilə başlayaraq `virtual void Goster()` kimi təyin edilməlidir. Bu cür təyin etmə yerinə yetirildikdən sonra törədilən siniflər üçün `virtual` təyin etmənin həyata keçirilməsinin və ya keçirilməməsinin heç bir əhəmiyyəti yoxdur. Xəyali üzvə



sahib olan sinifdən törənmiş siniflərin eyni adlı üzvləri də xəyali xüsusiyyət daşıyırlar.

```
//CLASSPOL.H duzeldilmis hissəsi

class Baza
{ protected:
    int X, Y;

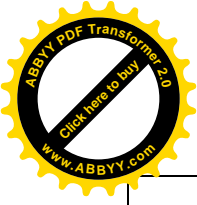
public:
    Baza(int C)
    { X = C; Y = 2 * C; }

    virtual void Goster()
    { printf("Baza X = %d\tY = %d\n", X, Y); }
};
```

Program çıxışı (düzəlişdən sonra)

```
A.Goster
    Baza X = 1      Y = 2
B.Goster
    Toremis X = 2    Y = 4      Z = 6
C = &A
C->Goster
    Baza X = 1      Y = 2
C = &B
C->Goster
    Baza X = 2      Y = 4
```

Bu iş prinsipini bu cür şərh etmək olar: `virtual` açarsözü olmadan edilən təyinlərdə obyektin yalnız məlumat sahələri (dəyişənləri) yaddaşda saxlanılır. Metodlar üçün isə kompilyator qərar verir. Lakin metod



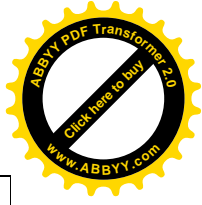
xəyali təyin edilərsə, bu, metodun yaddaşdakı yerinə aid məlumatda (başlanğıc ünvanı) obyektin digər məlumat sahələri ilə birlikdə yaddaşda saxlanılır. Belə bir metod çağırıldığı zaman da yaddaşdakı bu yer ilə əlaqədar olan məlumat istifadə edilərək haqqında söhbət açılan obyektin metoduna müraciət etməsi təmin olunur. Xüsusilə də bir sinfin metodları xaricində yoxedicilərinin də xəyali olaraq təyin edilməsinə imkan verilir. Layihələndiricilər isə xəyali olaraq təyin edilə bilməz.

5.5 Qaydalı Funksiyalar

Obyektlərlə proqramlaşdırma zamanı əsasən bütün obyektlər ümumi şəkildə deyil, eyni əməliyyatlar üçün nəzərdə tutulanlar təsnifləndirilərək baza obyektindən törədilir. Ümumi xüsusiyyətlər də mümkün olmadıqca baza obyekti daxilində cəmləşərək yenə bu obyekt üçün yazılır. Bundan sonra əsas obyektlər yazılaraq proqram ortaya çıxır.

Məsələn, əyrilərlə əlaqədar bir proqramda çevrə radiusu, ellips radiusu, qövs kimi əyrilərin olması və bu əyrilərin fəzadakı yerlərinin və uzunluqlarının hesablanması tələb oluna bilər.

Eyri		
Qovs	Cevre_Radiusu	Ellips_Radiusu



Bu halda bütün obyektləri bir **Eyri** sinfindən törətmək mümkündür. Əməliyyatları yerinə yetirmək üçün **Eyri** sinfinin əyrilərin yeri və ölçüləri ilə əlaqədar üzv funksiyaları olmalıdır.

```
//EYRI.CPP

class Eyri
{ protected:
    int _X, _Y;
public:
    Eyri();
    Eyri(const Eyri&);
    Eyri(int, int);

    virtual char* Ad() const;
    int Yer_X() const;
    int Yer_Y() const;

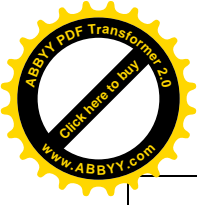
    virtual void Yer_X(int);
    virtual void Yer_Y(int);

    virtual int Uzunluq() const;
};

Eyri::Eyri()
{ _X = _Y = 0; }

Eyri::Eyri(const Eyri& eyri)
{ _X = eyri._X;
  _Y = eyri._Y;
}

Eyri::Eyri(int x, int y)
{ _X = x;
  _Y = y;
}
```



```
char* Eyri::Ad() const
{ return "EYRI"; }

int Eyri::Yer_X() const
{ return _X; }

int Eyri::Yer_Y() const
{ return _Y; }

void Eyri::Yer_X(int x)
{ _X = x; }

void Eyri::Yer_Y(int y)
{ _Y = y; }
```

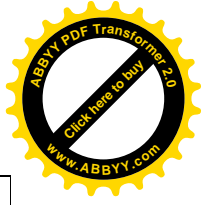
Lakin **Uzunluq** funksiyasını yazmaq lazım gəldiyi zaman **Eyri**-in uzunluğunun nə olduğu və ya necə hesablanacağı müəyyən olmadığı üçün **Uzunluq** funksiyası ancaq sonradan yazılmaq şərti ilə müvəqqəti bir funksiya kimi yazıla bilər.

EYRI.CPP davamı

```
int Eyri::Uzunluq() const
{ return 0; }
```

Bu cür təyinlərin üç əsas mənfəi cəhəti vardır:

1. Sonradan istifadə edilməsinə baxmayaraq təyin olunan bu funksiyalar, proqram daxilində istifadə olunmayacaqlarına baxmayaraq proqramın böyüməsinə səbəb olacaqlar;

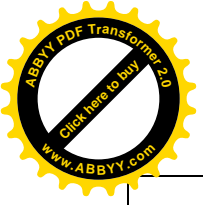


2. Proqramçılar istifadə etmədikləri bu proqram parçalarını yazmalı olacaqlar;
3. Gələcəkdə yazılması unudulduğu zaman səhv nəticələr verəcəkdir.

Bu hallardan qurtulmanın yolu isə, sinif daxilində bu cür sonradan yazılması tələb olunan funksiyaları təyin etdikdən sonra **"= 0"** ifadəsini yazmaqdır. Bu yazılış belə, bir funksiyanın olmasının vacibliyini, ancaq onun bu mərhələdə yazılmayacağını və sonrakı mərhələlərdə bu sinifdən törənən siniflərin həmin funksiyanı təyin edərək funksiyasında yazacaqlarını bildirir. Bu funksiyalar sonradan törənən siniflər daxilində yazılacaqları üçün onlar xəyali (**virtual**) təyin edilməlidirlər. Bu funksiyaları xəyali funksiyalardan fərqləndirmək üçün onlara xəyali kor funksiya (**pure virtual**) və ya qaydalı funksiya adı verilir.

Daxilində bu cür funksiyalar olan siniflərdən obyekt yaradıla bilməz. Bu siniflər ümumi məqsədlər üçün istifadə edilir. Bu cür siniflərə mücərrəd (**abstract**) sinif adı verilir. Yuxarıdakı misalda **Eyri** sinfi abstrakt sinifdir. Proqram daxilində **Eyri** adında bir obyekt ola bilməz. Ancaq **Eyri**-dən törənmiş həqiqi siniflərin (**Xett**, **Cevre_Qovsu** kimi) obyektləri ola bilər.

Proqramın strukturu ayrının müxtəlif əməliyyatları üçün bir obyektə ehtiyac duyulduğu zaman mücərrəd sinfin obyektlərinin yerinə göstəriciləri və ya təqdimatları



ola bilər. Təbii ki, bu halda göstərici və ya təqdimatlara həqiqi obyektlər mənimsədilməlidir.

Məsələn,

```
main()
{ Eyri A1;
  ...
  return 0;
}
```

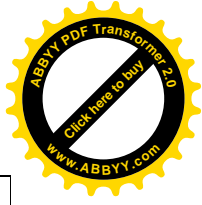
kimi bir proqram, mücərrəd sinif obyektindən istifadə etdiyi üçün yazıla bilmədiyi halda

```
main()
{ Eyri* Eptr = new Line(100, 100, 300, 600);
  ...
  return 0;
}
```

və ya

```
main()
{ Eyri*& Eptr = new Line(100, 100, 300, 600);
  ...
  delete Eref;
  return 0;
}
```

proqramları yazılaraq istifadə oluna bilərlər.



5.6 Misallar

5.6.1 Curve

Qaydalı funksiyalar üçün verilən misalı tamamlayaq.

```
//EYRI2.CPP

class Eyri
{ protected:
  int _X, _Y;
public:
  Eyri();
  Eyri(const Eyri&);
  Eyri(int, int);

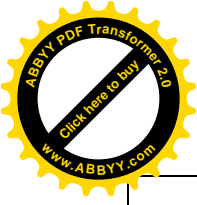
  virtual char* Ad() const = 0;
  int Yer_X() const;
  int Yer_Y() const;

  virtual void Yer_X(int);
  virtual void Yer_Y(int);

  virtual int Uzunluq() const = 0;
};

Eyri::Eyri()
{ _X = _Y = 0; }

Eyri::Eyri(const Eyri& eyri)
{ _X = eyri._X;
  _Y = eyri._Y;
}
```



```
Eyri::Eyri(int x, int y)
{ _X = x;
  _Y = y;
}

int Eyri::Yer_X() const
{ return _X; }

int Eyri::Yer_Y() const
{ return _Y; }

void Eyri::Yer_X(int x)
{ _X = x; }

void Eyri::Yer_Y(int y)
{ _Y = y; }

//*****

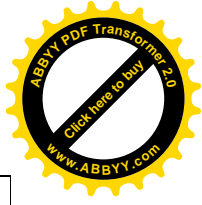
#include <math.h>
#include <stdlib.h>

class Cevre_Qovsu : public Eyri
{ protected:
    int _Radius;
    int _BaslangicBucagi;
    int _SonBucagi;

public:
    Cevre_Qovsu();
    Cevre_Qovsu(const Cevre_Qovsu&);
    Cevre_Qovsu(int, int, int = 1, int = 0, int = 360);
    Cevre_Qovsu(int);

    virtual char* Ad() const;
    virtual int Uzunluq() const;
};

Cevre_Qovsu::Cevre_Qovsu()
```



```
{ _Radius = 1;
  _BaslangicBucagi = 0;
  _SonBucagi = 360;
}

Cevre_Qovsu::Cevre_Qovsu(const Cevre_Qovsu& qovs) : Eyri(qovs)
{ _Radius = qovs._Radius;
  _BaslangicBucagi = qovs._BaslangicBucagi;
  _SonBucagi = qovs._SonBucagi;
}

Cevre_Qovsu::Cevre_Qovsu(int x, int y, int r, int a, int b) : Eyri(x, y)
{ _Radius = r;
  _BaslangicBucagi = a;
  _SonBucagi = b;
}

Cevre_Qovsu::Cevre_Qovsu(int r)
{ _Radius = r;
  _BaslangicBucagi = 0;
  _SonBucagi = 360;
}

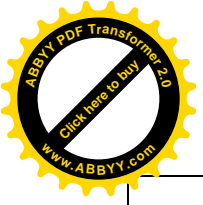
char* Cevre_Qovsu::Ad() const
{ return "Cevre Qovsu"; }

int Cevre_Qovsu::Uzunluq() const
{ return (int) 1.0 * _Radius * 2 * M_PI *
    abs(_SonBucagi - _BaslangicBucagi)/360.0;
}

#define SQR(a) ((float)(a) * (float)(a))

class Xett : public Eyri
{ protected:
    int _SonX;
    int _SonY;

public:
```



```
Xett();
Xett(const Xett&);
Xett(int, int, int, int);
Xett(int, int);

virtual char* Ad() const;
virtual int Uzunluq() const;
};

Xett::Xett()
{ _SonX = 1;
  _SonY = 1;
}

Xett::Xett(const Xett& xett) : Eyri(xett)
{ _SonX = xett._SonX;
  _SonY = xett._SonY;
}

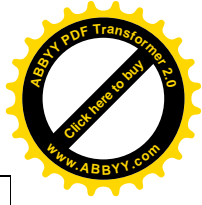
Xett::Xett(int x1, int y1, int x2, int y2) : Eyri(x1, y1)
{ _SonX = x2;
  _SonY = y2;
}

Xett::Xett(int x, int y)
{ _SonX = x;
  _SonY = y;
}

char* Xett::Ad() const
{ return "Xett"; }

int Xett::Uzunluq() const
{ return sqrt(SQR(_SonX - Yer_X()) + SQR(_SonY - Yer_Y())); }
//*****

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



```
Eyri *EyriMassivi[20];

void Hesab()
{ int i;
  printf("%3s %-22s %5s %5s %5s\n%30s %5s\n"
         "_____\n",
         "No", "Eyri Tipi", "Yer", "", "Olcu", "X", "Y");

  for(i = 0; i < 20; i++)
    if(EyriMassivi[i])
      { printf("%2d %-22s %5d %5d %5d\n",
               i, EyriMassivi[i]->Ad(), EyriMassivi[i]->Yer_X(),
               EyriMassivi[i]->Yer_Y(), EyriMassivi[i]->Uzunluq());
      }
}

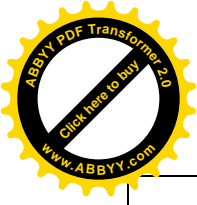
main()
{ clrscr();
  randomize();
  int i = 0;

  for(i = 0; i < 20; i++)
    if(random(2))
      EyriMassivi[i] = new Xett(random(100), random(100),
                                random(200), random(300));
    else
      { int baslangicbucagi = random(180);
        int sonbucagi = baslangicbucagi + random(180);

        EyriMassivi[i] = new Cevre_Qovsu(random(300), random(300),
                                           random(100), baslangicbucagi,
                                           sonbucagi);
      }

  Hesab();

  for(i = 0; i < 20; i++)
    if(EyriMassivi[i])
```

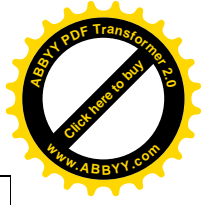


```
delete EyriMassivi[i];  
  
return 0;  
}
```

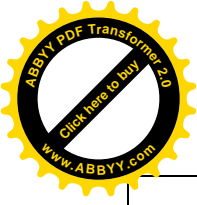
5.6.2 LineDemo

Xəttin qrafik ekranda təyin olunması və uc nöqtəsinin hərəkət etdirilərək canlandırılması (animasiyası) üçün yazılan bir proqrama edilən əlavələr ilə iki xətdən ibarət olan qrup təyini və eyni proqramın tətbiqi ilə əlaqədar misalları gözdən keçirək:

```
//LINE.CPP  
  
#include <graphics.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <conio.h>  
#include <dos.h>  
  
class Graphics  
{ public:  
    Graphics();  
    virtual ~Graphics();  
  
    virtual void Enter();  
    virtual void Leave();  
  
protected:  
    virtual char* getBGIPath();  
    virtual void MessageOnExit();  
};
```



```
Graphics::Graphics()  
{ }  
  
void Graphics::Enter()  
{ int grDrv;  
  int grMode;  
  char* grPath = "";  
  int grErr = grOk;  
  
  do  
  { grDrv = DETECT;  
    grMode = 0;  
  
    initgraph(&grDrv, &grMode, grPath);  
    if((grErr = graphresult()) != grOk)  
    { printf("Error %s\n", grapherrormsg(grErr));  
      grPath = getBGIPath();  
      if(grPath == NULL) break;  
    }  
  }  
  while(grErr != grOk);  
  
  if(grErr != grOk)  
    abort();  
}  
  
Graphics::~~Graphics()  
{ }  
  
void Graphics::Leave()  
{ MessageOnExit();  
  closegraph();  
}  
  
void Graphics::MessageOnExit()  
{ settextjustify(BOTTOM_TEXT, LEFT_TEXT);  
  outtextxy(0, getmaxy(), "Pres Any Key to EXIT Graph Mode");  
  getch();  
}
```

```
char* Graphics::getBGIPath()
{ static char Path[67];
  printf("Enter BGI path or '\\.'for end :");
  scanf("%s", Path);
  if(Path[0] == '.')
    return NULL;
  return Path;
}

//*****

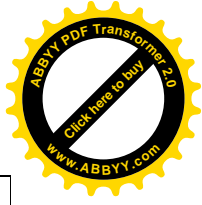
class Line
{ int X1, Y1, X2, Y2;
  int Status;

public:
  Line();
  Line(const Line&);
  Line(int, int);
  Line(int, int, int, int);

  int StartX() const;
  int EndX() const;
  int StartY() const;
  int EndY() const;

  virtual void MoveRel(int, int);
  virtual void Show();
  virtual void Hide();
  virtual int isVisible();
  virtual void StartPointRel(int, int);
  virtual void EndPointRel(int, int);

protected:
  virtual void Draw();
  virtual void Clear();
};
```



```
Line::Line()
{ X1 = Y1 = Y2 = 0;
  X2 = 1;
  Status = 0;
}

Line::Line(const Line& L)
{ X1 = L.X1;
  Y1 = L.Y1;
  X2 = L.X2;
  Y2 = L.Y2;
  Status = L.Status;
}

Line::Line(int x, int y)
{ X1 = Y1 = 0;
  X2 = x, Y2 = y;
  Status = 0;
}

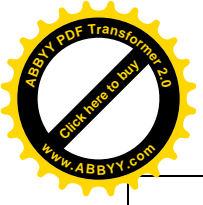
Line::Line(int a, int b, int c, int d)
{ X1 = a;
  Y1 = b;
  X2 = c;
  Y2 = d;
  Status = 0;
}

int Line::StartX() const
{ return X1; }

int Line::EndX() const
{ return X2; }

int Line::StartY() const
{ return Y1; }

int Line::EndY() const
```



```
{ return Y2; }

int Line::isVisible()
{ return Status; }

void Line::Show()
{ Status = 1;
  Draw();
}

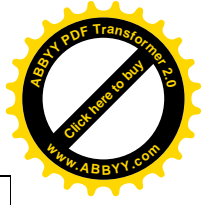
void Line::Hide()
{ Status = 0;
  Clear();
}

void Line::MoveRel(int Dx, int Dy)
{ StartPointRel(Dx, Dy);
  EndPointRel(Dx, Dy);
}

void Line::StartPointRel(int dx, int dy)
{ int S = isVisible();
  if(S) Hide();
  X1 += dx;
  Y1 += dy;
  if(S) Show();
}

void Line::EndPointRel(int dx, int dy)
{ int S = isVisible();
  if(S) Hide();
  X2 += dx;
  Y2 += dy;
  if(S) Show();
}

void Line::Draw()
{ int C = getcolor();
  setcolor(RED);
```



```
line(X1, Y1, X2, Y2);
setcolor(C);
}

void Line::Clear()
{ int C = getcolor();
  setcolor(getbkcolor());
  line(X1, Y1, X2, Y2);
  setcolor(C);
}

void DemoStartPoint(Line& L)
{ int i;
  for(i = 0; i < 23; i++)
  { L.StartPointRel(10, 0);
    delay(100);
  }
}

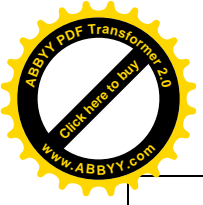
class LineGroup : public Line
{ Line& L1;
  Line& L2;

public:
  LineGroup(const LineGroup&);
  LineGroup(Line& a, Line& b);

  void MoveRel(int, int);
  void Show();
  void Hide();
  int isVisible();

  virtual void StartPointRel(int, int);
  virtual void EndPointRel(int, int);

private:
  int StartX() const;
  int EndX() const;
  int StartY() const;
```



```
int EndY() const;
};

LineGroup::LineGroup(const LineGroup& LG)
: L1(LG.L1),
  L2(LG.L2)
{}

LineGroup::LineGroup(Line& a, Line& b)
: L1(a),
  L2(b)
{}

void LineGroup::MoveRel(int dx, int dy)
{ L1.MoveRel(dx, dy);
  L2.MoveRel(dx, dy);
}

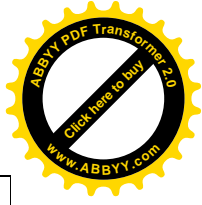
void LineGroup::Show()
{ L1.Show();
  L2.Show();
}

void LineGroup::Hide()
{ L1.Hide();
  L2.Hide();
}

int LineGroup::isVisible()
{ return L1.isVisible(); }

void LineGroup::StartPointRel(int dx, int dy)
{ L1.StartPointRel(dx, dy);
  L2.StartPointRel(dx, dy);
}

void LineGroup::EndPointRel(int dx, int dy)
{ L1.EndPointRel(dx, dy);
  L2.EndPointRel(dx, dy);
}
```



```
}

//*****

void LineDemo()
{ cleardevice();

  Line A(300, 300);
  A.Show();

  Line B(100, 200, 300, 300);
  B.Show();
  getch();

  A.Hide();
  getch();

  A.Show();
  getch();

  DemoStartPoint(A);
  DemoStartPoint(B);

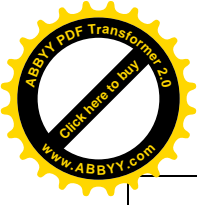
  /////////// LineGroup ///////////

  LineGroup Group(A, B);
  Group.Show();
  getch();

  Group.Hide();
  getch();

  Group.Show();
  getch();

  DemoStartPoint(Group);
  getch();
}
```



```
main()
{ clrscr();
  Graphics graphicsMedia;

  graphicsMedia.Enter();

  LineDemo();

  graphicsMedia.Leave();
  return 0;
}
```

İndi də **Line** sinfi əsasında **Box** (qutu) sinfini təyin edib eyni məqsədlə istifadə edək. Bunun üçün **LINE.CPP** proqramına qalın (**bold**) sitildə yazılmış sətirləri aşağıdakı kimi əlavə edək:

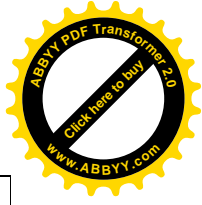
```
//BOX.CPP

#include <graphics.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>

class Graphics
{ public:
  Graphics();
  virtual ~Graphics();

  virtual void Enter();
  virtual void Leave();

protected:
  virtual char* getBGIPath();
  virtual void MessageOnExit();
```



```
};

Graphics::Graphics()
{}

void Graphics::Enter()
{ int grDrv;
  int grMode;
  char* grPath = "";
  int grErr = grOk;

  do
  { grDrv = DETECT;
    grMode = 0;

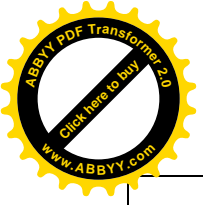
    initgraph(&grDrv, &grMode, grPath);
    if((grErr = graphresult()) != grOk)
    { printf("Error %s\n", grapherrormsg(grErr));
      grPath = getBGIPath();
      if(grPath == NULL) break;
    }
  }
  while(grErr != grOk);

  if(grErr != grOk)
    abort();
}

Graphics::~Graphics()
{}

void Graphics::Leave()
{ MessageOnExit();
  closegraph();
}

void Graphics::MessageOnExit()
{ settextjustify(BOTTOM_TEXT, LEFT_TEXT);
  outtextxy(0, getmaxy(), "Pres Any Key to EXIT Graph Mode");
```



```
    getch();
}

char* Graphics::getBGIPath()
{ static char Path[67];
  printf("Enter BGI path or '\\\\for end :");
  scanf("%s", Path);
  if(Path[0] == '\\')
    return NULL;
  return Path;
}

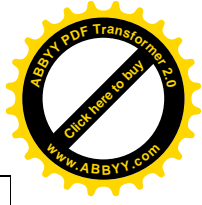
class Line
{ int X1, Y1, X2, Y2;
  int Status;

public:
  Line();
  Line(const Line&);
  Line(int, int);
  Line(int, int, int, int);

  int StartX() const;
  int EndX() const;
  int StartY() const;
  int EndY() const;

  virtual void MoveRel(int, int);
  virtual void Show();
  virtual void Hide();
  virtual int isVisible();
  virtual void StartPointRel(int, int);
  virtual void EndPointRel(int, int);

protected:
  virtual void Draw();
  virtual void Clear();
};
```



```
Line::Line()
{ X1 = Y1 = Y2 = 0;
  X2 = 1;
  Status = 0;
}

Line::Line(const Line& L)
{ X1 = L.X1;
  Y1 = L.Y1;
  X2 = L.X2;
  Y2 = L.Y2;
  Status = L.Status;
}

Line::Line(int x, int y)
{ X1 = Y1 = 0;
  X2 = x, Y2 = y;
  Status = 0;
}

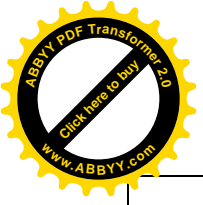
Line::Line(int a, int b, int c, int d)
{ X1 = a;
  Y1 = b;
  X2 = c;
  Y2 = d;
  Status = 0;
}

int Line::StartX() const
{ return X1; }

int Line::EndX() const
{ return X2; }

int Line::StartY() const
{ return Y1; }

int Line::EndY() const
{ return Y2; }
```



```
int Line::isVisible()
{ return Status; }

void Line::Show()
{ Status = 1;
  Draw();
}

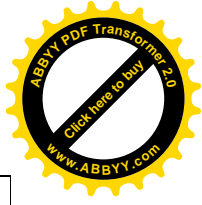
void Line::Hide()
{ Status = 0;
  Clear();
}

void Line::MoveRel(int Dx, int Dy)
{ StartPointRel(Dx, Dy);
  EndPointRel(Dx, Dy);
}

void Line::StartPointRel(int dx, int dy)
{ int S = isVisible();
  if(S) Hide();
  X1 += dx;
  Y1 += dy;
  if(S) Show();
}

void Line::EndPointRel(int dx, int dy)
{ int S = isVisible();
  if(S) Hide();
  X2 += dx;
  Y2 += dy;
  if(S) Show();
}

void Line::Draw()
{ int C = getcolor();
  setcolor(RED);
  line(X1, Y1, X2, Y2);
```



```
    setcolor(C);
}

void Line::Clear()
{ int C = getcolor();
  setcolor(getbkcolor());
  line(X1, Y1, X2, Y2);
  setcolor(C);
}

void DemoStartPoint(Line& L)
{ int i;
  for(i = 0; i < 23; i++)
  { L.StartPointRel(10, 0);
    delay(100);
  }
}

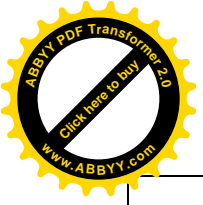
class LineGroup : public Line
{ Line& L1;
  Line& L2;

public:
  LineGroup(const LineGroup&);
  LineGroup(Line& a, Line& b);

  void MoveRel(int, int);
  void Show();
  void Hide();
  int isVisible();

  virtual void StartPointRel(int, int);
  virtual void EndPointRel(int, int);

private:
  int StartX() const;
  int EndX() const;
  int StartY() const;
  int EndY() const;
```



```
};

LineGroup::LineGroup(const LineGroup& LG)
: L1(LG.L1),
  L2(LG.L2)
{}

LineGroup::LineGroup(Line& a, Line& b)
: L1(a),
  L2(b)
{}

void LineGroup::MoveRel(int dx, int dy)
{ L1.MoveRel(dx, dy);
  L2.MoveRel(dx, dy);
}

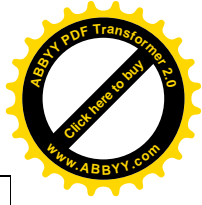
void LineGroup::Show()
{ L1.Show();
  L2.Show();
}

void LineGroup::Hide()
{ L1.Hide();
  L2.Hide();
}

int LineGroup::isVisible()
{ return L1.isVisible(); }

void LineGroup::StartPointRel(int dx, int dy)
{ L1.StartPointRel(dx, dy);
  L2.StartPointRel(dx, dy);
}

void LineGroup::EndPointRel(int dx, int dy)
{ L1.EndPointRel(dx, dy);
  L2.EndPointRel(dx, dy);
}
```



```
}

//*****

void LineDemo()
{ cleardevice();

  Line A(300, 300);
  A.Show();

  Line B(100, 200, 300, 300);
  B.Show();
  getch();

  A.Hide();
  getch();

  A.Show();
  getch();

  DemoStartPoint(A);
  DemoStartPoint(B);

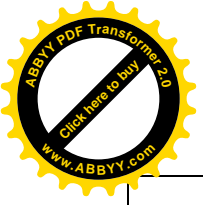
  //////////// LineGroup ////////////

  LineGroup Group(A, B);
  Group.Show();
  getch();

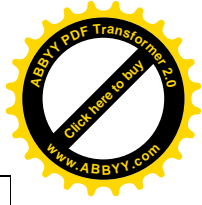
  Group.Hide();
  getch();

  Group.Show();
  getch();

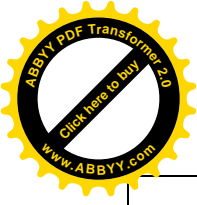
  DemoStartPoint(Group);
  getch();
}
```



```
//*****  
  
class Box : public Line  
{ public:  
    Box();  
    Box(const Box&);  
    Box(int, int);  
    Box(int, int, int, int);  
  
protected:  
    virtual void Draw();  
    virtual void Clear();  
};  
  
Box::Box()  
{ }  
  
Box::Box(const Box& box) : Line(box)  
{ }  
  
Box::Box(int x, int y) : Line(x, y)  
{ }  
  
Box::Box(int a, int b, int c, int d) : Line(a, b, c, d)  
{ }  
  
void Box::Draw()  
{ int C = getcolor();  
  setcolor(GREEN);  
  rectangle(StartX(), StartY(), EndX(), EndY());  
  setcolor(C);  
}  
  
void Box::Clear()  
{ int C = getcolor();  
  setcolor(getbkcolor());  
  rectangle(StartX(), StartY(), EndX(), EndY());  
  setcolor(C);  
}
```



```
void BoxDemo()  
{ cleardevice();  
  
    Box Abox(300, 300);  
    Abox.Show();  
  
    Box Bbox(100, 200, 300, 300);  
  
    Bbox.Show();  
    getch();  
  
    Abox.Hide();  
    getch();  
  
    Abox.Show();  
    getch();  
  
    DemoStartPoint(Abox);  
    DemoStartPoint(Bbox);  
  
    /////////// BoxGroup ///////////  
  
    LineGroup Group(Abox, Bbox);  
    Group.Show();  
    getch();  
  
    Group.Hide();  
    getch();  
  
    Group.Show();  
    getch();  
  
    DemoStartPoint(Group);  
    getch();  
}  
  
main()  
{ clrscr();
```

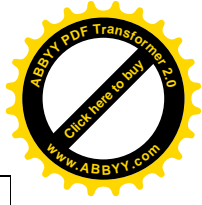



```
Graphics graphicsMedia;  
  
graphicsMedia.Enter();  
  
LineDemo();  
  
BoxDemo();  
  
graphicsMedia.Leave();  
return 0;  
}
```

5.7 C++ Metod Çağırış Sistemi

İndi də xəyali funksiyaların işləmə prinsipini şərh edək. Bunun üçün **C**-dən bildiyimiz kimi bir funksiya yazıldığı zaman, yaddaşda kompilyator tərəfindən müəyyən edilib çağırılarkən onun mövqeyinə avtomatik keçid baş verir. Lakin xəyali funksiyalar üçün bu mümkün deyildir.

Eyri misalında olduğu kimi təyin olunmuş baza sinfinin göstəricilərinə **Xett** mənimsədildiyi zaman çağırılan **Uzunluq** funksiyası xəttin uzunluğunu hesablayarkən, **Cevre_Qovsu** mənimsədilib uzunluğu soruşulduğu zaman, bu dəfə də çevrə qovsu üçün hesablamaları apararaq geri göndərir. Bu cür siniflərarası müqayisəni aparmaq üçün siniflərin hər biri üçün xəyali funksiya cədvəlləri (**virtual function table**) yaradılır.



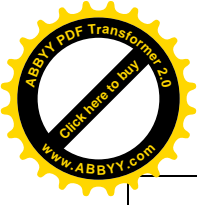
```
class Eyri  
{ protected:  
    int _X, _Y;  
  
    public:  
        Eyri();  
        Eyri(const Eyri&);  
        Eyri(int, int);  
  
    virtual char* Adlar() const = 0;  
  
    int Yer_X() const;  
    int Yer_Y() const;  
  
    virtual void Yer_X(int);  
    virtual void Yer_Y(int);  
  
    virtual int Uzunluq() const = 0;  
};
```

təyininə uyğun olaraq aşağıdakı cədvəl hazırlanır.

No	Adı	Ünvanı
1	Adlar	NULL
2	Yer_X	Eyri::Yer_X
3	Yer_Y	Eyri::Yer_Y
4	Uzunluq	NULL

Cədvəldəki **NULL** qiymətləri qaydalı funksiyaları göstərir.

Xett sinfi təyin edilərkən oxşar bir cədvəl onun üçün də hazırlanır. **Xett** sinfi **Eyri** sinfindən törəndiyi üçün bu cədvələ əvvəlcə **Eyri** sinfinin qiymətləri köçürülür. Sonra isə **Xett** sinfi üçün yazılan xəyali funksiyalar bu



cədvəldən tapılaraq ünvan qiymətləri dəyişdirilir. Törənən sinif üçün də yeni təyin edilən xəyali funksiyalar varsa, bunlar da cədvəlin sonuna sətir əlavə edilərək daxil edilir.

No	Adı	Unvanı
1	Ad	Xett::Ad
2	Yer_X	Eyri::Yer_X
3	Yer_Y	Eyri::Yer_Y
4	Uzunluq	Xett::Uzunluq

Buna oxşar əməliyyatlar törənən bütün siniflər üçün yerinə yetirilir. Bu siniflərdən birinin obyektı yaradıldığı zaman bu siniflər virtual funksiyaya malik olduqları üçün hansı cədvəlin istifadə ediləcəyi də daxil olmaqla eyni zamanda cədvəl göstəricisini də saxlayırlar. Məsələn, **Xett** və **Cevre_Qovsu** obyektlərini yaradaq:

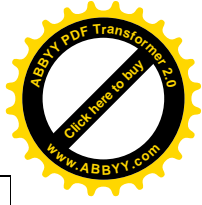
```
Xett D;  
Cevre_Qovsu C;
```

```
D⇒[VTP→Xett][_X][_Y][Son_X][Son_Y]
```

```
C⇒[VTP→Cevre_Qovsu][_X][_Y][_Radius][_BaslangicBucagi][  
_SonBucagi]
```

```
VTP ≡ VirtualTablePointer
```

Göründüyü kimi hər bir təyinə xəyali funksiyanın cədvəl göstəricisi (**VTP**) daxildir. Bu məlumat hər bir obyektin aid olduğu sinfin funksiya cədvəlini göstərir.



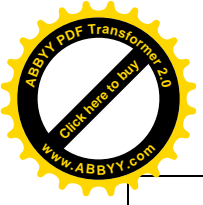
Xəyali funksiyalar çağırılarkən də elə bu cədvəldən faydalanaraq çağırma əməliyyatları yerinə yetirilir.

Məsələn, **D** obyektı üçün **Uzunluq** funksiyası çağırıldığı zaman, **D**-nin **VTP** qiyməti ilə **Xett** sinfinin cədvəlinə və bu cədvəlin 4-cü sətri vasitəsilə də **Xett::Uzunluq** funksiyasına müraciət ediləcəkdir. Bu cür birbaşa olmayan müraciətlər **C** proqramlarına nisbətən sürətin aşağı düşməsinə səbəb olur.

5.8 Mövcud Olandan Törənən Siniflər

Məlum olduğu kimi obyekt yönü proqramlaşdırmada törədilmiş sinfin yalnız bir baza sinfinə malik olma məcburiyyəti yoxdur. Bir sinif bir neçə sinifdən törəyə bilər. Bu halda törənən sinif törəndiyi sinfin bütün üzvlərinə müraciət edib onları istifadə edə bilər. Bu hal bütün siniflərin xüsusiyyətlərinin tək bir sinifdə cəmlənməsini təmin etdiyi üçün daha güclü siniflərin yaranmasına səbəb olur. Bu cür təyinlər çox bazalılıq adlandırılır.

C++-də bu cür təyinlərdə baza siniflərinin virtual, olub olmamasından asılı olmayaraq eyni metodlardan (eyni ad və parametr siyahısına malik olan funksiyalardan) ibarət olması zamanı törənən sinif daxilində bu metodların yenidən təyin edilməsinə ehtiyac



vardır. Bu təyin ilə baza siniflərindən biri və ya hamısı çağırılaraq biləcəyi kimi, metod yenidən də yazıla bilər.

Çox bazalılıq xüsusiyyətinin üstünlüklərindən biri də törənən sinfin obyektinin baza siniflərindən hər hansı birinin göstəricisinə mənimsədilə bilməsidir. Bu halda törənən sinfin obyektini mənimsəddiyi sinif kimi davranaraq özünə aid qabiliyyətlərini nümayiş etdirir.

```
//MULTINT2.CPP

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

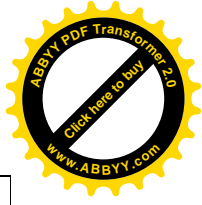
class A
{ public:
    A()
    {}
    void A_Xususi()
    { printf("Bu A sinfinin A_Xususiyyetidir.\n"); }

    void Yaz()
    { printf("Bu A sinfinin Yazilmasidir.\n"); }

    virtual void Goster()
    { printf("Bu A sinfinin Gosterilmesidir.\n"); }
};

class B
{ public:
    B()
    {}

    void B_Xususi()
    { printf("Bu B sinfinin B_Xususiyyetidir.\n"); }
```



```
virtual void Yaz()
{ printf("Bu B sinfinin Yazilmasidir.\n"); }

virtual void Goster()
{ printf("Bu B sinfinin Gosterilmesidir.\n"); }
};

class C : public A, public B
{ public:
    C()
    {}

    void C_Xususi()
    { printf("Bu C sinfinin C_Xususiyyetidir.\n"); }

    void Yaz()
    { printf("Bu C sinfinin Yazilmasidir.\n"); }

    void Goster()
    { printf("Bu C sinfinin Gosterilmesidir.\n"); }
};

main()
{ clrscr();

    C obyekt;

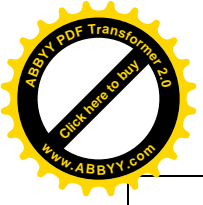
    obyekt.A_Xususi();
    obyekt.B_Xususi();
    obyekt.C_Xususi();

    obyekt.Yaz();
    obyekt.Goster();

    printf("\n");

    A *APtr;

    APtr = &obyekt;
```



```
APtr->A_Xususi();
APtr->Yaz();
APtr->Goster();

printf("\n");

B *BPtr;

BPtr = &obyekt;
BPtr->B_Xususi();
BPtr->Yaz();
BPtr->Goster();

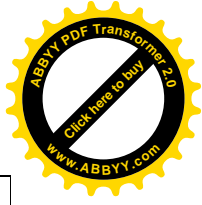
return 0;
}
```

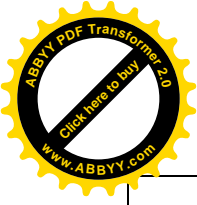
Program çıxışı

Bu A sinfinin A_Xususiyyetidir.
Bu B sinfinin B_Xususiyyetidir.
Bu C sinfinin C_Xususiyyetidir.
Bu C sinfinin Yazilmasidir.
Bu C sinfinin Gosterilmesidir.

Bu A sinfinin A_Xususiyyetidir.
Bu A sinfinin Yazilmasidir.
Bu C sinfinin Gosterilmesidir.

Bu B sinfinin B_Xususiyyetidir.
Bu C sinfinin Yazilmasidir.
Bu C sinfinin Gosterilmesidir.





VI FƏSİL

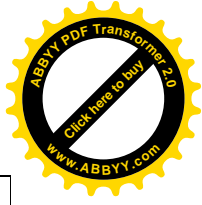
ŞABLONLAR HAZIRLAMAQ

6.1 Şablonlar

Bir proqram kodunun müxtəlif hallar üçün təkrar yazılması yerinə qəlib kimi hazırlanıb istifadə edilməsinə şablonlama (**template**) adı verilir. Məsələn, **float** tipli elementlərdən ibarət massiv ilə **double** tipli və hətta **int**, **char** və ya **long** tipli elementlərdən ibarət massivlərin kiçikdən böyüyə sıralanması alqoritması arasında heç bir fərq yoxdur. Bu halda hər bir tip üçün ayrı bir sıralama funksiyasının yazılması artıq bir işdir. Yalnız bir proqram kodu ilə proqram yazmaq həm proqramçının yükünü azaldır, həm də gələcəkdə ediləcək dəyişikliklərdə eyni məqsəd üçün bir çox proqram kodunun dəyişdirilməsi məcburiyyətini aradan qaldırır.

Bu baxımdan **C** dilində bu cür kodlaşdırmalar makroların köməyi ilə aparılır və bu hal ümumi (**generic**) təyinlər adlandırılır. Bir misalə baxaq:

```
//GENSORT.C
```



```
#define IMP_SORT(Tip)
void imp_sort_##Tip(Tip Mas[], unsigned int Olcu)
{
    int Nezaret = 1;
    int i;
    Tip c;

    while (Nezaret)
    {
        Nezaret = 0;
        for( i = 1; i < Olcu; i++ )
            if(Mas[i-1]>Mas[i])
            {
                c = Mas[i-1];
                Mas[i-1] = Mas[i];
                Mas[i] = c;
                Nezaret = 1;
            }
    }
}

IMP_SORT(float);
IMP_SORT(double);
IMP_SORT(int);
IMP_SORT(long);

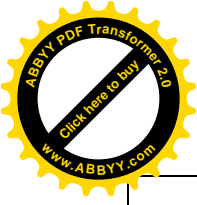
#define SORT(Tip, Mas, Olcu) imp_sort_##Tip(Mas, Olcu)

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

double D[5] = {3.4, 7, -2.5, 4.1, 0.5};
long L[5] = {6, 2, 90, 34, 45};
int i;

main()
{ clrscr();

    SORT(double, D, 5);
```



```
SORT(long, L, 5);

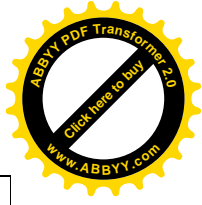
printf("\nlong\tdouble\n");
for(i = 0; i < 5; i++)
    printf("%ld\t\t%lf\n", L[i], D[i]);

return 0;
}
```

Misalda `IMP_SORT` makrosu ilə sıralama funksiyalarının ümumi bir şablonu çıxardılmış, təyin olunma sətrinin altında isə yeni bir makrodan istifadə edərək mövcud tiplər üçün sıralama funksiyaları əldə edilmişdir. Bu funksiyaların çağırılması üçün də `SORT` adlı digər bir makro yazılmışdır.

Ümumiləşdirilmiş bu makro təyirlərinin mənfə cəhətləri isə aşağıdakılardır:

- Sıralanacaq bir massiv elementlərinin tipi parametr siyahısına birbaşa yazılmalıdır. Əks halda səhv baş verir;
- Əgər massiv elementlərinin tipi dəyişdiriləcəksə, bu massivi sıralamaq üçün istifadə edilən `SORT` funksiyaları da dəyişdirilməlidir;
- Sıralanacaq hər massiv elementləri `IMP_SORT` ilə uyğun tipdə təyin edilməlidir. Məsələn, `char` tipli bir massiv sıralanmadan əvvəl bütün



funksiyaların xaricində (qlobal olaraq) `IMP_SORT(char);` sətri yazılmalıdır.

6.2 Şablon Funksiyalar

İndi də `C++`-da şablon xüsusiyyətə malik eyni proqramı yazaq.

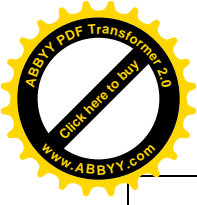
```
//TMPSORT.CPP

template <class Tip>
void SORT(Tip Mas[], unsigned int Olcu)
{
    int Nezaret = 1;
    int i;
    Tip c;

    while(Nezaret)
    {
        Nezaret = 0;
        for( i = 1; i < Olcu; i++ )
            if(Mas[i-1]>Mas[i])
            {
                c = Mas[i-1];
                Mas[i-1] = Mas[i];
                Mas[i] = c;
                Nezaret = 1;
            }
    }
}

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

main()
```



```
{
  clrscr();
  double D[5] = {3.4, 7, -2.5, 4.1, 0.5};
  long L[5] = {6, 2, 90, 34, 45};

  SORT(D, 5u);
  SORT(L, 5u);

  int i;
  printf("\nlong\\t\\tdouble\\n");
  for(i = 0; i < 5; i++)
    printf("%ld\\t\\t%lf\\n", L[i], D[i]);

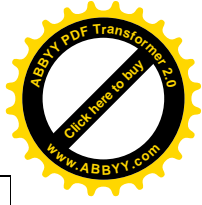
  return 0;
}
```

Proqram çıxışı

long	double
2	-2.500000
6	0.500000
34	3.400000
45	4.100000
90	7.000000

Bununla mövcud olan bütün tiplər üçün **SORT** funksiyası təyin edilir.

Burada **SORT** funksiyasının çağırılmasına nəzər yetirsəniz, sıralanacaq massiv elementlərinin tiplərinə aid heç bir məlumat görməyəcəksiniz. Çünki artıq kompilyator **SORT** funksiyasının çağırılması zamanı birinci parametrin tipini istifadə edərək hansı funksiyanı çağıracağına özü qərar verir.



template ifadəsi C++'ın acar sözüdür. **template** ilə təyin edilən funksiyalarda əvvəlcədən proqram kodunun dəyişməsinə səbəb olan tiplər təyin edilərək onlara simvolik bir ad verilir. Bu təyin **template** açar sözü ilə bərabər aşağıdakı simvolik adlar əlavə edilərək başlanır.

```
template<class simvolik_ad1 [, class simvolik_ad2 ...]>
```

Bu təyindən sonra funksiya, simvolik adlardan da istifadə edilərək C++ qaydalarına görə yazılır.

template ilə əldə edilən funksiyalarla istifadəçinin özünün təyin etdiyi obyektlərin istifadə edilməsi də mümkündür.

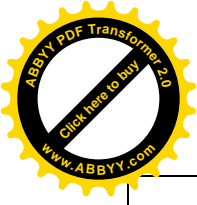
```
//TMPMAX.CPP

#include <conio.h>
#include <iostream.h>

template <class T>
T Max(T a, T b)
{ return a > b ? a: b; }

class A
{ int N, M;

  public:
  A()
  { N = 0; M = 1; }
  A(int a, int b)
  { N = a; M = b; }
  A(const A& K)
  { N = K.N; M = K.M; }
```



```
friend int operator>(const A& a, const A& b)
{ return a.N % a.M > b.N % b.M; }

friend ostream& operator<<(ostream& Stream, const A& a)
{ Stream<<a.N % a.M;
  Stream<<"("<<a.N<<" mod "<<a.M<<"");
  return Stream;
}
};

main()
{ clrscr();
  A Bir(34, 9);
  A Iki(456, 13);

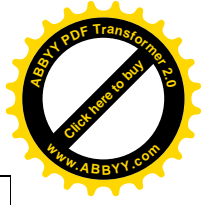
  cout<<"Bir = "<<Bir<<endl;
  cout<<"Iki = "<<Iki<<endl;
  cout<<"En boyuk qiymet ->"<<Max(Bir, Iki)<<endl;

  return 0;
}
```

Proqram çıxışı

```
Bir = 7(34 mod 9)
Iki = 1(456 mod 13)
En boyuk qiymet ->7(34 mod 9)
```

Burada **template** ilə təyin edilən funksiyanın simvolik adla verilən obyektədən nə gözlədiyinə diqqət etmək lazımdır. **Max** funksiyası obyektin öz sinfindən digər obyekt ilə müqayisə edilməsini və "böyükdürmü?" sualının cavablandırılmasını gözləyir. Buna görə də **A**



sinifi təyin edilərkən **operator >** funksiyası da təyin edilmişdir.

6.3 Şablon Obyektlər

Bəzi obyektlər icra olunduğu zaman istifadə etdikləri dəyişənlərin tipləri müxtəlif hallarda dəyişə bilər. Bu obyektlər üçün də funksiyalardakına oxşar şablonlar hazırlayıb istifadə etmək olar. Obyekt təyinləri iki və daha çox hissədən (üzv və dost funksiya təyinlərindən) təşkil olunduğu üçün funksiya təyininə nisbətən fərqlənir.

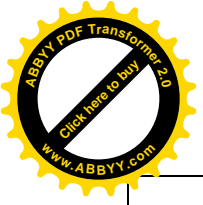
//TMPARRAY.CPP

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

```
template <class Tip>
class Array
{ Tip *Mas;
  unsigned int Olcu;
```

```
public:
  Array(unsigned int);
  Array(const Array&);
  ~Array()
  { if (Mas) delete Mas; }
  Tip& operator[](unsigned int);
```

```
virtual void PrintHeader(ostream& Stream) const
{ Stream<<'{';
```

```
virtual void PrintSeperator(ostream& Stream) const
{ Stream<<','; }

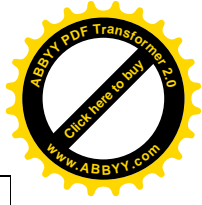
virtual void PrintTrailer(ostream& Stream) const
{ Stream<<'}'; }

virtual void Print(ostream&) const;
};
```

Burda sinif təyini funksiyalarda olduğu kimi **template** ifadəsi ilə başlamışdır. Simvolik tip adına əsaslanaraq təyinlər edilmişdir. Buradakı istifadə qaydası da funksiyalarda olduğu kimidir. Lakin **inline** ilə kodlaşdırılmayan funksiyalar üçün sonradan kodlaşdırılarkən **template** ifadəsini sinif təyininə olduğu kimi kodlaşdırılacaq funksiyaların əvvəlinə yazmaq lazımdır. Burada diqqət ediləcək hal simvolik tip adlarının eyni olması deyil, sinif təyininəki kimi tip adının bu təyin daxilində də eyni ardıcılıqla yazılmasıdır.

TMPARRAY.CPP davamı

```
template <class Tip>
Array<Tip>::Array(unsigned int B)
{ Mas = new Tip[Olcu = B];
  if(!Mas) abort();
}
```



Burada nəzəri cəlb edən, **template** ifadəsindən başqa, təyin edilən üzv funksiyasının hansı obyektə aid olduğunu göstərən görmə (**scope**) operatorunun yazılmasındakı dəyişiklikdir. Normal halda **Array::** şəklində olması lazım gələn təyin **Array::<Tip>::** şəklində edilmişdir. Buna səbəb **Array**-in sinif təyini deyil, müxtəlif **Array** sinfinin şablonu olmasıdır. Başqa sözlə təyin edilməkdə olan üzv funksiya **Array<Tip>** kimi təyin olunmuş bir şablonun üzvüdür.

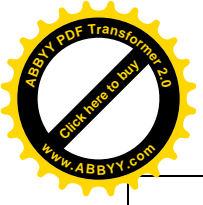
Digər tərəfdən təyin edilməkdə olan layihələndiricinin adının sadəcə **Array** olduğuna diqqət edin.

TMPARRAY.CPP davamı

```
template <class Tip>
Array<Tip>::Array(const Array<int> &A)
{ Mas = new Tip[Olcu = A.Olcu];
  if(!Mas) abort();

  int i;
  for(i = 0; i < Olcu; i++)
    Mas[i] = A.Mas[i];
}
```

Təyin edilməkdə olan üzv funksiyanın parametrləri köçürmə layihələndiricisində olduğu kimi şablondan əldə ediləcək bir obyektə göstərəcəksə, bu **Array** kimi deyil, **Array<Tip>** şəklində yazılmalıdır. Bu, şablondan əldə ediləcək sinfin üzv və ya dost funksiyasının daxilinə



eyni sinifdən bir obyektin parametr kimi girməsi deməkdir.

TMPARRAY.CPP davamı

```
template <class Tip>
Tip& Array<Tip>::operator[] (unsigned int I)
{ static Tip Komekci;
  if(I < Olcu) return Mas[I];
  return Komekci;
}

template <class Tip>
void Array<Tip>::Print(ostream& Stream) const
{ PrintHeader(Stream);

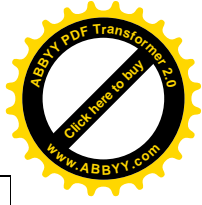
  int i;
  for(i = 0; i < Olcu - 1; i++)
  { Stream<<Mas[i];
    PrintSeperator(Stream);
  }

  Stream<<Mas[Olcu - 1];
  PrintTrailer(Stream);
}
```

Şablon təyin edildikdən sonra onun vasitəsilə obyektlər təyin edilərkən artıq simvolik tip adlarının əvəzinə həqiqi adlar təyin etmə ilə bərabər yazılmalıdır.

Məsələn, `double` və `unsigned long int` elementli massivlər üçün

```
Array<double> MassivD;
```



```
Array<unsigned long int> MassivULI;
```

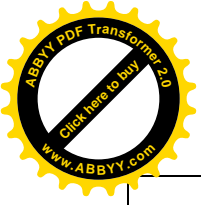
şəklində olmalıdır.

TMPARRAY.CPP davamı

```
main()
{ clrscr();
  Array<int>I(5);
  I[0] = 1; I[1] = 3; I[2] = 5; I[3] = 7; I[4] = 9;

  cout<<endl;
  I.Print(cout);
  cout<<endl;

  return 0;
}
```



VII FƏSİL

AXINLAR

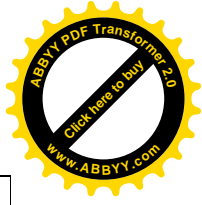
7.1 Axın Nədir?

Axın (**Stream**) – məlumatların ardıcıl formada axınını təmin edən, məlumatları istifadə etməzdən əvvəl və sonra onların saxlanması ardıcıl formada nizamlayan mexanizmdir (sınıf və ya obyekt). Axınlar giriş və ya çıxış məqsədilə istifadə edilir. Axınların əsas əhəmiyyəti istər standart tipdə olsun, istərsə də proqramçı tərəfindən təyin edilmiş olsun, hər tipdə olan dəyişkənin (obyektin) axına yazılıb oxuna bilməsidir. **C**-də yazılan proqramlarda olduğu kimi yazma və oxuma əməliyyatlarında format sətrinə ehtiyac yoxdur.

7.2 Standart Axınlar

C++-də təyin olunmuş standart axınlar aşağıdakılardır:

Axın	Simvolik fayl	DOS faylı	İstifadə məqsədi
cin	stdin	con	standart giriş
cout	stdout	con	standart çıxış



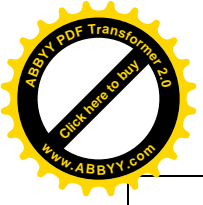
Axın	Simvolik fayl	DOS faylı	İstifadə məqsədi
cerr	stderr	con	səhv mesajı
clog		con	çap

Hər hansı bir axından verilənləri oxumaq üçün **>>** (sağa sürüşdürmə), axına verilənləri yazmaq üçün **<<** (sola sürüşdürmə) operatorundan istifadə edilir.

İndi bucağın qiymətini dərəcə ilə daxil edən və radyan ilə ekrana çıxaran bir **C** və **C++** proqramı yazaraq bunları müqayisə edək.

//BUCAQ.C	//BUCAQ.CPP
<pre>#include <stdio.h> #include <math.h> #include <conio.h> double d, r; main() { clrscr(); printf("Bucaq (derece) : "); scanf("%lg", &d); r = d / 180.0 * M_PI; printf("\n%lg%c = %lg rad\n", d, 248, r); return 0; }</pre>	<pre>#include <iostream.h> #include <math.h> #include <conio.h> double d, r; main() { clrscr(); cout << "Bucaq (derece) : "; cin >> d; r = d / 180.0 * M_PI; cout << d << (char)248 << " = " << r << " rad" << endl; return 0; }</pre>

Bu iki proqramın müqayisəsindən göründüyü kimi **printf("mesaj");** ifadəsi **cout<<"mesaj";** ifadəsinə çevrilmiş,



`cout` `printf` ilə eyni funksiyanı yerinə yetirmişdir. `printf("%d, %lf\n", 5, 1.4);` kimi bir ifadə də `cout<<5<<","<<1.4<<"\n";` şəklinə çevrilərək format təyinediciləri göstərilməmişdir. Beləliklə, format təyinedicisinin istədiyi tip ilə uyğun gələn qiymətin tiplərinin uyğunsuzluğundan meydana gələ biləcək səhvlər aradan qaldırılmışdır. Bununla bərabər növbəti paraqraflarda görəyimiz kimi formatlı çıxış üçün **C**-nin imkanlarından da istifadə oluna bilər.

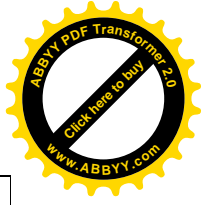
Giriş (oxuma) əməllərində isə `scanf("%lg", &d);` kimi bir ifadənin yerinə `cin>>d;` ifadəsi yazılaraq format ifadəsi və `&` ünvan operatorundan istifadə edilməmişdir.

Axınların istifadə edilməsi zamanı axına iki və daha artıq arqumentin daxil edilməsi, ya da oxunması lazım gələrsə, arqumentlər arasında uyğun istiqamətləndirmə işarələri qoyaraq əməliyyatları yerinə yetirmək mümkündür.

```
int X, Y;
double Z;
unsigned int U;
cin >>X>>Z>>U>>Y;
cout <<X<<'*'<<Y;
cout <<'='<<(X*Y)<<"\n";
```

və ya

```
cin >>X;
cin >> Z;
```



```
cin >> U;
cin >> Y;
cout <<X<<'*'<<Y<<'='<<(X*Y)<<"\n";
```

eyni əməliyyatları yerinə yetirən iki proqram hissəsidir.

`<<` və `>>` operatorlarını axınlarla istifadə edərkən ardıcılıqlarına diqqət etmək lazımdır. Belə ki,

```
cout <<X = Y<<"\n";
```

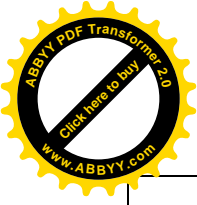
ifadəsində əməliyyatların prioritet sırası nəzərə alınarsa,

```
(cout <<X) = (Y<<"\n");
```

şəklində icra olunur. Bu da complyasiya səhvi verir. Çünki `(cout<<X)` əməliyyatı nəticəsində `X`-in qiyməti ekrana çıxarılaraq `cout` qiyməti alınır. İkinci halda ifadə `cout = (Y<<"\n");` halını alır. Bu ifadənin sağ tərəfi də hesablandıqdan sonra (əgər, təbii ki, hesablanırsa) `cout`-a bir qiymət mənimsədilməyə cəhd edilir. Bu cür mənimsətmə operatoru təyin olunmadığı üçün səhv baş verir. Belə hallarda ifadə mötərizələrin köməyi ilə açıq şəkildə yazılmalıdır.

```
cout <<(X = Y)<<"\n";
```

Yenə də sağa və ya sola sürüşdürmə əməliyyatlarının axınlarla birlikdə istifadə edilməsi



səhvlərə yol açma biləcəyi üçün mötərizələrdən istifadə etmək məqsədəuyğundur.

```
int X = 3;
cout <<X<<1<<'\n';
cout <<(X<<1)<<'\n';
```

Program nəticəsinin ekran görünüşü

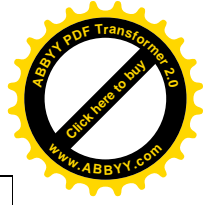
```
31
6
```

olacaqdır. İlk sətir 3 ilə 1-in ardıcıl yazılmasını, ikinci sətir isə 3-ün bir bit sola sürüldürülməsi və nəticənin ekrana çıxarılmasını təmin edir.

7.3 Axınlara Nizamlanmış Məlumat Yazılması

7.3.1 Genişlik Nəzarəti

Ekrana çıxarılaçaq məlumatın axın üzərində müəyyən sayda simvol uzunluğunu doldurması tələb olunarsa, bu axın üçün `width()` üzv funksiyasından istifadə edilir. Məsələn, ekrana çıxarılaçaq istənilən bir ədədin 12 simvol uzunluğunda olması üçün



```
cout.width(12);
cout<<56;
```

ifadəsindən istifadə edilir. `width` ifadəsi sadəcə özündən sonrakı məlumat sahəsinin yazılmasına təsir edir. Daha sonrakı məlumatların ekrana çıxarılmasında mənası olmaz. Əgər lazım gələrsə, digər məlumatları da ekrana çıxarmazdan əvvəl eyni ifadədən istifadə edilməlidir.

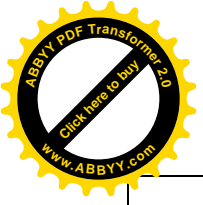
Lakin məlumat verilən genişliyə sığmazsa, bu genişlik nəzərə alınmır və ehtiyac olduğu qədər sahə istifadə edilərək ekrana çıxarılır.

Əvvəlcədən verilmiş genişlik qiymətini müəyyənləşdirmək üçün parametrsiz `width()` funksiyası istifadə edilərək bundan sorakı yazma (ekrana çıxarma) əməliyyatında məlumat üçün ayrılacaq sahənin genişliyini təyin etmək olar.

```
int Genislik = cout.width();
```

`width()` funksiyalarının hansının istifadə edilməsindən asılı olmayaraq nəticədə axın üçün daha əvvəlki addımlarda təyin olunan yazma genişliyinin qiyməti geri göndəriləcəkdir. Hər yazma əməliyyatından sonra yazma genişliyi sıfır olacaqdır. Yazma genişliyinin sıfır olması axına ehtiyac olduğu qədər yazma sahəsini istifadə etmə imkanını verir.

```
int width();
```



```
int width(int);
```

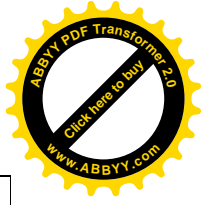
7.3.2 Yerləşmə Nəzarəti

Məlumat yazılarkən, onun genişliyi verilən genişlikdən az olarsa, məlumatlar sağa sıxışdırılaraq yazılır. Sol tərəfləri isə boş buraxılır.

Əgər məlumatları yalnız sağa sıxışdırılmış deyil, istəyə uyğun olaraq sağa və ya sola sıxışdırılaraq yazmaq tələb olunarsa, axınların mənsub olduqları `ios::adjustfield` parametri `ios::left` və ya `ios::right` kimi göstərilməlidir. Bu baxımdan `ios::adjustfield` parametrinə mənimsətmək üçün `setf()` funksiyasından istifadə etmək olar. Məsələn,

```
cout.setf(ios::left, ios::adjustfield);
cout.widht(14);
cout<<56;
cout.setf(ios::right, ios::adjustfield);
cout.width(14);
cout<<56;
```

Burada `ios::right` məlumatın verilən genişlik daxilində sağa, `ios::left` isə sola sıxışdırılmasını təmin edir.



7.3.3 Boşluq Nəzarəti

Məlumatların müəyyən bir genişlikdəki sahədə sağa və ya sola sıxışdırılaraq yazılması nəticəsində sol və ya sağ tərəfdə istifadə olunmamış sahələr qalır. Bu sahələr çox zaman boş saxlansa da, bəzən doldurula da bilər. Bunun üçün `fill()` üzv funksiyasından istifadə edilir. Məsələn, yazılacaq qiymətlərin sağa sıxışdırılmış olması və solda qalan boş sahələrin də sıfırla doldurulması tələb olunarsa,

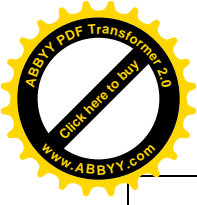
```
cout.setf(ios::right, ios::adjustfield);
cout.fill('0');
cout.width(14);
cout<<78<<"n";
```

kimi bir proqram hissəsi yazıla bilər.

`fill()` funksiyasının təsiri yeni bir `fill()` funksiyasının istifadə edilməsinə kimidir. Buna görə də doldurulma məcburiyyətinin aradan qaldırılması üçün `fill(32);` və ya `fill(' ');` sətirləri yazılmalıdır.

`fill()` funksiyası hər zaman əvvəlki halında istifadə edilməyən sahələrə doldurulacaq simvolu qaytarır. Əgər doldurma simvolu dəyişdirilməzsə və sadəcə hansı simvolum olduğunu müəyyənləşdirmək lazım gələrsə, parametrsiz `fill()` funksiyasından istifadə etmək olar.

```
char EwelkiSimvol = cout.fill();
```



```
cout.fill('!');  
...  
cout.fill(EvvelkiSimvol);
```

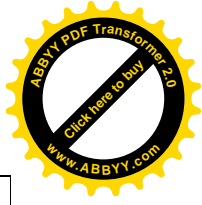
Hər iki `fill()` funksiyasının prototipi aşağıdakı kimidir:

```
char fill();  
char fill(char);
```

7.3.4 Tam Ədədlərin Əsaslarına Nəzarət

Tam ədədlər **3** formada: onluq (**decimal**), səkkizlik (**octal**) və onaltılıq (**hexadecimal**) say sistemlərində yazıla bilərlər. Tam ədədlər axınlara bu say sistemlərindən birində yazıla bilər. Əsasən onluq say sistemində, lazım gəldiyi zaman da digər say sistemlərində yazıla bilər. Bunun üçün `setf()` üzv funksiyasından istifadə edilir. `setf()` funksiyası ilə axınlar üçün təyin olunmuş `ios::basefield` parametrinə `ios::dec`, `ios::oct` və ya `ios::hex` qiymətlərindən biri əlavə edilməlidir.

```
int x = 36;  
  
cout.setf(ios::dec, ios::basefield);  
cout<<x<<"\n";  
cout.setf(ios::oct, ios::basefield);  
cout<<x<<"\n";  
cout.setf(ios::hex, ios::basefield);  
cout<<x<<"\n";
```



nəticədə

```
36  
44  
24
```

qiymətləri axında (`cout` olduğu üçün ekranda) görünəcəkdir.

7.3.5 Həqiqi Ədədə Nəzarət

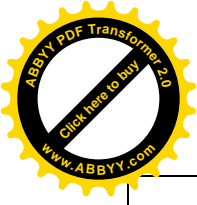
Həqiqi ədədlər yazılarkən onluq nöqtədən sonra neçə rəqəmin yazılacağı təyin olunması, lazımsız rəqəmlərin qarışıqlıq və nizamsız bir görünüş yaratmasının qarşısını alır. Bu `precision()` üzv funksiyası ilə təyin edilir. Bu funksiya ilə nöqtədən sonra görünəcək rəqəmlərin sayı təyin edilir. Geri qaytarma qiyməti kimi isə əvvəlki əsas qiymət qaytarılır.

```
double X = 7.7881881;  
cout.precision(3);  
cout<<X<<"\n"
```

nəticədə

```
7.788
```

və ya



```
double X = 7.7881881;  
cout.precision(2);  
cout<<X<<"\n"
```

nəticədə

7.79

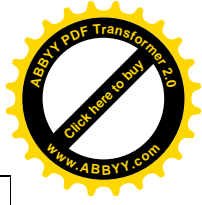
kimi görünəcəkdir.

Buradan göründüyü kimi atılan rəqəmlər yuvarlaqlaşdırılır.

Həqiqi ədədlərin yazılmasında, əsas qiymət xaricində əhəmiyyətli olan qiymətin nizamlanmış olmasıdır. Həqiqi ədədlər iki müxtəlif formada göstərilə bilər: $\pm x.xxxxE\pm xxxx$ şəklində mühəndislik görünüşü (*scientific*) və $\pm xxxx.xxxx$ şəklindəki normal görünüşü (*fixed*).

Axınlar üçün nizamlama əməliyyatlarının yerinə yetirilməsi ancaq onlar üçün təyin edilmiş `ios::floatfield` parametrinə `ios::fixed` və ya `ios::scientific` qiymətlərindən birinin yazılması ilə mümkündür. `ios::fixed` normal görünüş, `ios::scientific` isə mühəndislik görünüşü üçündür.

```
double X = 567.8990;  
cout.setf(ios::fixed, ios::floatfield);  
cout<<X<<"\n";  
cout.setf(ios::scientific, ios::floatfield);
```



```
cout<<X<<"\n"
```

Nəticədə axındakı görünüş

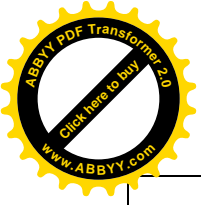
567.899
5.67899e+2

şəkilində olur.

7.4 Axınlardan Nizamlanmış Məlumat Oxunması

Axınlardan məlumatların oxunması üçün yenə də əvvəlki paragrafda şərh edilən nizamlanmış məlumat yazma funksiyalarından istifadə edə bilərsiniz. Bunlardan yalnız tam ədədlərin əsaslarının göstərilməsi əhəmiyyətlidir. Digərlərinin heç bir təsiri yoxdur.

Tam ədədi axından oxumaq üçün bu ədədin əvvəlində `0o`, `0O`, `0x`, `0X` kimi işarələr yoxdursa, daxil edilmiş ədədlər onluq ədəd kimi qəbul edilir. Ədədlərin əvvəlində `0o` və ya `0O` yazılırsa, onların səkkizlik, `0x` və ya `0X` yazılırsa, onların onaltılıq say sistemində yazıldığı qəbul edilir. Bu halda oxunacaq hər hansı bir ədədin əsası göstərilərsə, axında olan ədəd də eyni əsasda olmalıdır. Əks halda səhv qəbul edilir.



Misal 1:

```
#include <conio.h>
#include <iostream.h>

main()
{ clrscr();
  int Integer;

  cin>>Integer;
  cout<<"\n"<<Integer<<"\n";

  return 0;
}
```

Program giriş və çıxışı

```
45
45
```

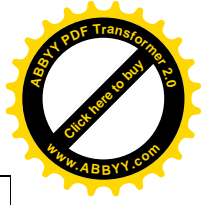
Misal 2:

```
#include <conio.h>
#include <iostream.h>

main()
{ clrscr();
  int Integer;

  cin.setf(ios::oct, ios::basefield);
  cin>>Integer;
  cout<<"\n"<<Integer<<"\n";

  return 0;
}
```



Program giriş və çıxışı

```
45
37
```

Misal 3:

```
#include <conio.h>
#include <iostream.h>

main()
{ clrscr();
  int Integer;

  cin.setf(ios::hex, ios::adjustfield);
  cin>>Integer;
  cout<<"\n"<<Integer<<"\n";

  return 0;
}
```

Program giriş və çıxışı 1

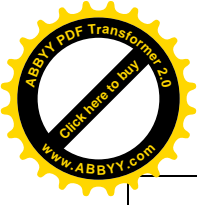
```
45
69
```

Program giriş və çıxışı 2

```
0x45
69
```

Program giriş və çıxışı 3

```
0045 <səhfdır
0
```



7.5 Səhvlərə Nəzarət

Hər hansı bir axın ilə yerinə yetirilən əməliyyatlarda səhvin baş verməsi zamanı axın özü-özlüyündə bir iş görmür. Yalnız səhvin baş verdiyi haqqında məlumatı özündə saxlayır. Daha sonra özündən soruşulan bəzi məlumatlara bu məlumata əsaslanaraq cavab verir.

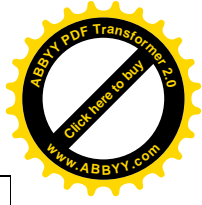
Bu suallar aşağıdakılardır:

Funksiya	(TRUE) Sıfırdan fərqli qaytarma qiymətinin səbəbi
<code>int bad()</code>	Səhv baş vermişsə.
<code>int fail()</code>	Əməliyyat səhv səbəbindən yarımçıq qalmışsa.
<code>int good()</code>	İcra olunan əməliyyatlardan sonra heç bir səhv olmamışsa.
<code>int eof()</code>	Oxuma məqsədli axından fayl sonu simvolu oxunmuşsa.

Səhvin səbəbi məlum olub düzəldildikdən sonra əməliyyatların davam etdirilməsi tələb olunarsa, meydana gələn səhvin sonrakı mərhələlərdə də səhv kimi qiymətləndirilməməsi üçün axına aid səhv məlumat silinməlidir. Bunun üçün `void clear()` funksiyasından istifadə olunur.

7.6 Fayl Üzərindəki Axınlar

Faylların axın kimi istifadə edilməsi üçün bəzi funksiyalar təyin edilmişdir. Bu funksiyaları istifadə



etmək üçün `fstream.h` başlıq faylı program koduna əlavə edilməlidir.

7.6.1 Fayla Yazma

Məlumatların fayllara yazılması üçün `ofstream()` sinfindən istifadə edilir. Bu sinfin 4 müxtəlif layihələndiricisi vardır.

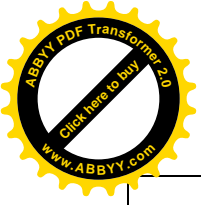
`ofstream();`

Bu layihələndirici yalnız yazma məqsədli axın təyin edir. Bu axının hansı fayl olduğu `open()` üzv funksiyası ilə müəyyən edilir. `open()` funksiyasının strukturu

`void open(char* fayl_adi, int tip);`

şəklindədir. `fayl_adi` açılacaq faylın əməliyyat sisteminin xüsusiyyətlərinə uyğun olaraq verilmiş adıdır. `tip` isə yazma məqsədli fayllar üçün `ios::out` və ya `ios::app` olmalıdır. `ios::app` əgər fayl mövcuddursa, yeni məlumatların faylın sonuna əlavə edilməsini, `ios::out` isə məlumatların faylın başlanğıcından etibarən daxil edilməsini təmin edir. Bu halda fayldakı mövcud olan məlumatlar silinir.

Axınlar açılma rejimindən asılı olmayaraq `close()` üzv funksiyası ilə bağlanmalıdır.



```
#include <fstream.h>

main()
{ ofstream Output;

  Output.open("misal1.dat", ios::out);
  if (Output.bad())
  { cerr<<"Fayl acilmir.\n";
    return 1;
  }

  Output<<"Axina yazma\n";
  Output<<8<<' '<<7.78<<"\n";

  Output.close();

  return 0;
}
```

MISAL1.DAT faylına yazılan məlumatlar

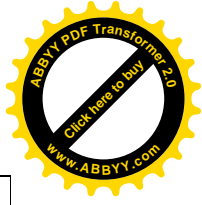
```
Axina yazma
8 7.78
```

`ofstream(char* fayl_adi);`

Bu layihələndirici də *fayl_adi* ilə verilən faylın axınla əlaqələndirilərək istifadə edilməsini təmin edir.

```
#include <fstream.h>

main()
{ ofstream Output("misal2.dat");
```



```
if (Output.bad())
{ cerr<<"Fayl acilmir.\n";
  return 1;
}

Output<<"Axina yazma\n";
Output<<8<<' '<<7.78<<"\n";

Output.close();

return 0;
}
```

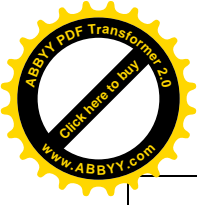
MISAL2.DAT faylına yazılan məlumatlar

```
Axina yazma
8 7.78
```

`ofstream(int handle);`

Bu layihələndirici daha əvvəl açılmış bir faylın axın kimi açılıb istifadə edilməsi üçün təyin edilmiş bir layihələndiricidir. Bu layihələndirici ilə bərabər istifadə edilən *handle* ilə verilən fayl bu təyindən sonra sadəcə axın kimi istifadə edilməlidir. Əks halda fayl daxilində səhv məlumatlara rast gələ bilərsiniz.

`ofstream(int handle, char* buffer, int uzunluq);`



Bundan əvvəl şərh edilmiş layihələndirici kimi eyni məqsədlər üçün istifadə edilən bu layihələndirici *buffer* ilə verilən *uzunluq* uzunluğundakı aralıq yaddaş vasitəsi ilə fayla yazma əməliyyatını yerinə yetirir. Yəni aralıq yaddaş dolana qədər məlumatları bu yaddaşa yazır. Yaddaş dolduqdan sonra bu məlumatların hamısını fayla yazır. Sonra aralıq yaddaşa yeni məlumatları əvvəldən yazmağa başlayır.

```
//OFSTR3.CPP
#include <io.h>
#include <fcntl.h>
#include <fstream.h>

main()
{ int OutputHandle = open("misal3.dat", O_RDWR);
  write(OutputHandle, "Kohne idareetme\n", 12);
  ofstream Output(OutputHandle);

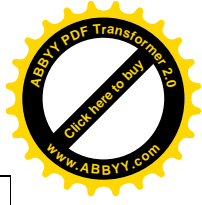
  Output<<"Axina yazma\n";
  Output<<8<<' '<<7.78<<"\n";

  Output.close();

  return 0;
}
```

MISAL3.DAT faylına yazılan məlumatlar

```
Kohne idareetme
Axina yazma
8 7.78
```



7.6.2 Fayldan Oxuma

Faylların axın olaraq təyin edilməsi üçün *ifstream()* sinfindən istifadə olunur. Bu sinfin də *ofstream()* sinfində olduğu kimi dörd müxtəlif layihələndiricisi vardır.

ifstream();

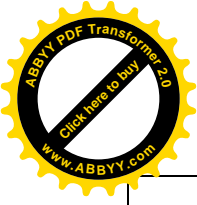
Bu layihələndirici, yalnız oxuma məqsədli axın təyin edir. Bu axının hansı fayl olduğu *open()* üzv funksiyası ilə təyin edilir. *open()* funksiyasının yazılışı bundan əvvəlki paraqraflarda olduğu kimidir. Yalnız tip kimi təyin olunmuş ikinci parametrlər oxuma məqsədli fayllar üçün *ios::in* olmalıdır. Axın necə açılmasından asılı olmayaraq *close()* üzv funksiyası ilə bağlanmalıdır.

```
//IFSTR1.CPP
#include <conio.h>
#include <fstream.h>

main()
{ clrscr();
  ifstream Input;

  int X, Y;
  double D, E;
  char *Setir;

  Input.open("input.dat", ios::in);
  if (Input.rdstate())
  { cerr<<"Fayli acma xetasi.\n";
```



```
return 1;
}

Input>>X>>D;
Input>>Y>>E;
Input>>Setir;
Input.close();

cout<<endl<<Setir<<endl;
cout<<"Tam ededler\t"<<X<<\t'<<Y<<endl;
cout<<"Heqiqi ededler\t"<<D<<\t'<<E<<endl;

return 0;
}
```

INPUT.DAT faylı

```
3 67.8
90 12e34
Misal
```

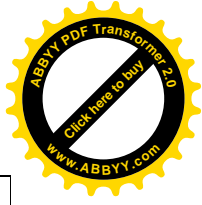
Program çıxışı

```
Misal
Tam ededler    3          90
Heqiqi ededler 67.8      1.2e+35
```

```
ifstream(char * faylin_adi);
```

Bu layihələndirici *faylin_adi* ilə verilən faylı axınla əlaqələndirərək istifadə edilməsini təmin edir. Bu halda *faylin_adi* adlı fayl mövcud olmalıdır.

```
ifstream(int handle);
```



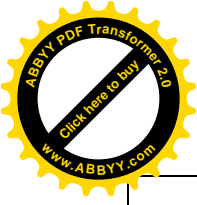
Bu layihələndirici əvvəlcədən oxuma məqsədi ilə açılmış faylın axın kimi istifadə edilməsi üçün təyin olunmuş layihələndiricidir. Bu layihələndirici ilə birlikdə istifadə edilən *handle* ilə verilmiş fayl bu təyindən sonra yalnız axın kimi istifadə edilməlidir. Əks halda fayl daxilində səhv məlumatlara rast gələ bilərsiniz.

```
ifstream(int handle, char *buffer, int uzunluq);
```

ifstream(int handle); layihələndiricisi ilə eyni məqsədlər üçün istifadəsi mümkün olan bu layihələndirici *buffer* ilə verilən *uzunluq* uzunluğundakı aralıq yaddaş vasitəsilə fayldan oxuma əməliyyatını yerinə yetirir. Əvvəlcə aralıq yaddaş dolana qədər fayldan bu yaddaşa məlumat oxunur. Sonra oxuma əməliyyatı bu yaddaş üzərindən icra olunur. Aralıq yaddaşdakı məlumatların hamısı oxunduqdan sonra bu yaddaşa fayldan yeni məlumatlar oxunaraq əməliyyat davam etdirilir.

7.7 Obyektlər və Axınlar

Programçının özünün təyin edəcəyi obyektləri axınlara << operatoru ilə yazıb və >> operatoru ilə oxuya bilməsi üçün bu obyektlər (siniflər) üçün uyğun operatorları da təyin etməlidir.



Bu operator funksiyaları ümumi şəkildə aşağıdakı kimi olmalıdır:

```
ostream& operator<<
(ostream& Stream, const YeniClass& Obyekt);
```

```
istream& operator>>
(istream& Stream, const YeniClass& Obyekt);
```

```
//POINT.CPP

#include <iostream.h>

class Point
{ int _X, _Y;

public:
    Point(int x = 0, int y = 0);

    int X() const;
    int Y() const;

    void X(int);
    void Y(int);

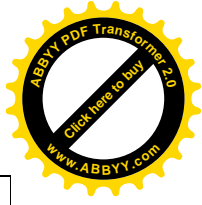
    void MoveRel(int, int);

    virtual void Draw() const;
};

ostream& operator<<(ostream&, const Point&);
istream& operator>>(istream&, Point&);

//*****

#include <graphics.h>
```



```
Point::Point(int x, int y)
{ _X = x;
  _Y = y;
}

int Point::X() const
{ return _X; }

int Point::Y() const
{ return _Y; }

void Point::X(int x)
{ _X = x; }

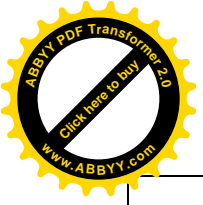
void Point::Y(int y)
{ _Y = y; }

void Point::MoveRel(int dx, int dy)
{ _X += dx;
  _Y += dy;
  if(_X < 0 || _X > getmaxx())
    _X = getmaxx() / 2;
  if(_Y < 0 || _Y > getmaxy())
    _Y = getmaxy() / 2;
}

void Point::Draw() const
{ line(_X - 2, _Y - 2, _X + 2, _Y + 2);
  line(_X - 2, _Y + 2, _X + 2, _Y - 2);
}

ostream& operator<<(ostream& stream, const Point& point)
{ stream<<point.X()<<' '<<point.Y();
  return stream;
}

istream& operator>>(istream& stream, Point& point)
{ int x, y;
```



```
stream>>x>>y;
point.X(x);
point.Y(y);

return stream;
}

//*****

class PointArray
{ Point *_Array;
  int _Size;

public:
  PointArray(int);
  ~PointArray();

  int Size() const;

  void Read();
  void Write();

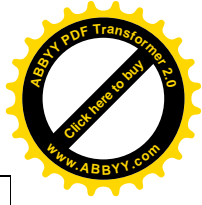
  void Load(char*);
  void Save(char*);

  void Draw() const;
  void Animate() const;

protected:
  void Create(int);
  void Destroy();
};

//*****

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



```
#include <conio.h>

PointArray::PointArray(int size)
{ Create(size); }

PointArray::~PointArray()
{ Destroy(); }

int PointArray::Size() const
{ return _Size; }

void PointArray::Read()
{ cout<<"\nNoqtelerin X ve Y qiymetlerini\n";
  cout<<"aralarina bosluq qoyaraq daxil edin\n";

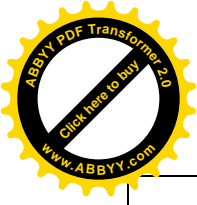
  int i;
  for(i = 0; i < _Size; i++)
  { cout<<i+1<<">";
    cin>>_Array[i];
  }
}

void PointArray::Write()
{ cout<<"Noqtenin qiymeleri\n";

  int i;
  for(i = 0; i < _Size; i++)
    cout<<i+1<<"\t"<<_Array[i]<<endl;
}

void PointArray::Load(char* fname)
{ ifstream Input(fname);
  if(Input.rdstate())
    return;

  Destroy();
  int size;
  Input>>size;
  Create(size);
```



```
int i;
for(i = 0; i < _Size; i++)
    Input>> _Array[i];
}

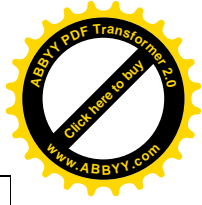
void PointArray::Save(char* fname)
{ ofstream Output(fname);
  if(Output.rdstate()) return;

  int i;
  Output<< _Size<<endl;
  for(i = 0; i < _Size; i++)
    Output<<_Array[i]<<endl;
}

void PointArray::Draw() const
{ int i;
  for(i = 0; i < _Size; i++)
    _Array[i].Draw();
}

void PointArray::Animate() const
{ randomize();
  int i = 0;
  setwritemode(XOR_PUT);
  Draw();
  outtextxy(0, getmaxy() - 12, "Dayandirmaq ucun bir duymeye
sixin");
  while(!kbhit())
  { if(i >= _Size)
    { i = 0;
      _Array[i].Draw();
      _Array[i].MoveRel(random(3) - 1, random(3) - 1);
      _Array[i].Draw();
      i++;
    }
  }

  getch();
}
```



```
}

void PointArray::Create(int size)
{ _Array = new Point [_Size = size];
  if(!_Array)
    abort();
}

void PointArray::Destroy()
{ delete []_Array; }

//*****

main()
{ clrscr();
  PointArray Noqteler(15);
  Noqteler.Read();

  clrscr();
  Noqteler.Write();
  cout<<"ndavam etmek ucun bir duymeye sixin\n";
  getch();

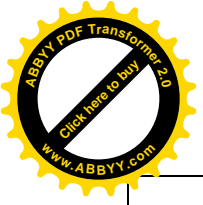
  Noqteler.Save("Noqte.ntk");
  Noqteler.Load("Noqte.ntk");

  cout<<"Fayldan oxunan qiymetler\n";
  Noqteler.Write();
  cout<<"ndavam etmek ucun bir duymeye sixin\n";
  getch();

  int D = DETECT, M = 0;
  initgraph(&D, &M, "c:\\borlandc\\bgi");
  if(graphresult() != grOk)
    abort();

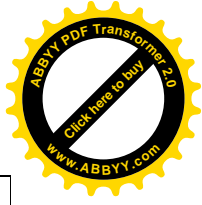
  Noqteler.Animate();

  closegraph();
}
```

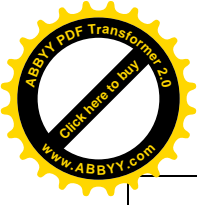
AXINLAR

```
cout<<"Animate funksiyasından sonraki qiymetler\n";  
Noqteler.Write();  
cout<<"\ndavam etmek ucun bir duymeye sixin\n";  
getch();  
  
return 0;  
}
```



Etibar Seyidzadə





VIII FƏSİL

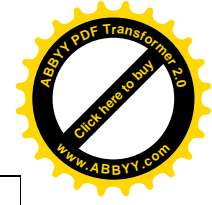
CLASS KİTABXANASI

8.1 Container Class Kitabxanası

Container Class kitabxanası proqramçılar tərəfindən çox istifadə edilən müəyyən təyinlər, məlumatlar strukturu, alqoritmlərin yazılması və təkmilləşdirilməsində, problem üzərində çox vaxt sərf etmə zamanı əhəmiyyətli rol oynayır. Bu təyinlər və məlumatlar strukturu C++-da əvvəlcədən hazırlanaraq proqramçılara təqdim edilmişdir. **Borland C++** və **Turbo C++**-da bu strukturların istifadə edilməsi ilə əlaqədar şərh verməyə ehtiyac vardır.

Bu strukturların başlıq faylları **C:\BORLANDC\CLASSLIB\INCLUDE** qovluğunda saxlanılır. Başlıq fayllarının proqrama daxil edilməsi üçün onların adı ilə birlikdə bu sətir də (yol) yazılmalıdır. Bu sətiri **OPTIONS** menyusundan **DIRECTORIES** əmrini seçərək, açılacaq dialoq pəncərəsində **INCLUDE** parametrinə uyğun gələn sətirə daxil etmək lazımdır.

Bundan başqa **class** kitabxanalarının **LINK** mərhələsində ola bilməsi üçün kitabxanaların daxilində



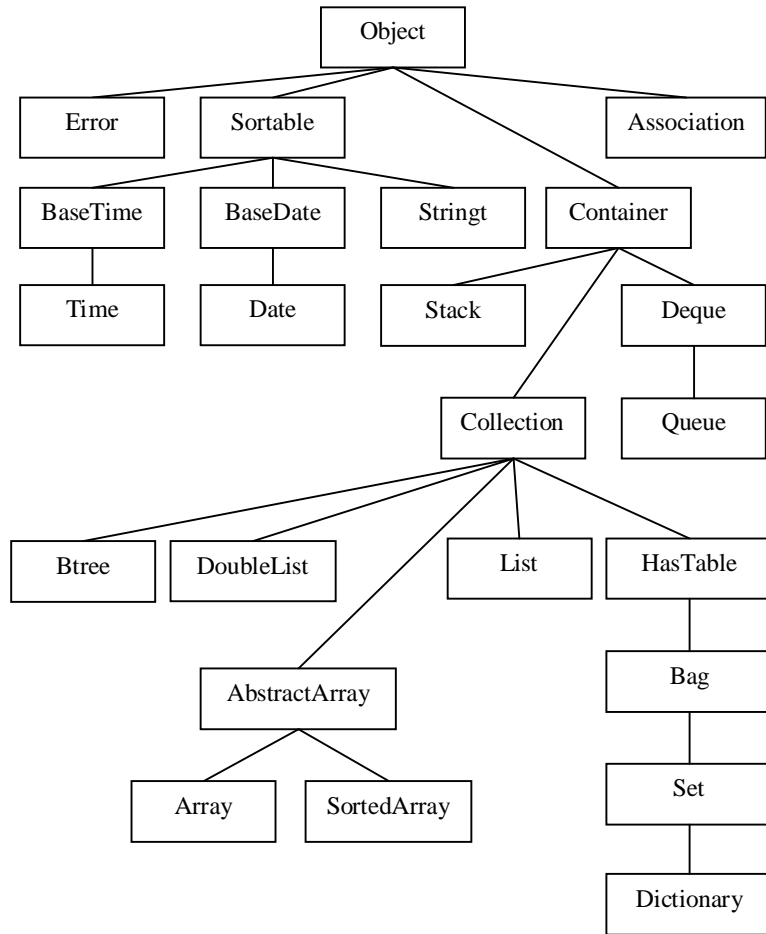
olduğu **\BORLANDC\CLASSLIB\LIB** yolu **INCLUDE** yoluna oxşar təyin edilməlidir. Bu yolu **OPTIONS** menyusundan **DIRECTORIES** əmrini seçərək, açılacaq dialoq pəncərəsində **LIBRARIES** parametrinə uyğun gələn sətirə yazmaq lazımdır. Normal olaraq istifadə ediləcək kitabxananın adı **PROJECT** daxilində göstərilməli və ya **LINK** mərhələsində kitabxana adı olaraq verilməlidir. Lakin **Borland C++** paketində **OPTIONS** menyusundan **LINKER** alt menyusunu, sonra da **LIBRARIES** əmri, açılacaq dialoq pəncərəsində "Container Class Library" sahəsindəki **STATIC** və ya **DINAMIC** parametrlərindən biri seçilməlidir.

8.2 Təyin Olunmuş Siniflər

Təyin olunmuş üç sinif vardır:

1. Verilənləri saxlayan siniflər:
 - a) bir məlumatdan ibarət olan siniflər;
 - b) iki məlumat arasında əlaqə quran siniflər;
 - c) çox məlumatdan ibarət olan siniflər.
2. Sürücülərlə işləyən siniflər;
3. Köməkçi siniflər.

Siniflərin növləri aşağıdakılardır:



8.3 Təyinlər və Tiplər

8.3.1 Tip və Sınıf Kodları

Kitabxananın kodlaşdırılmasında istifadə edilən təyinlər və tiplər aşağıdakılardır:

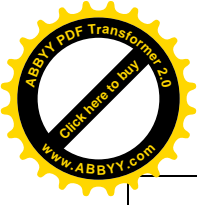
classType

`typedef unsigned int classType;`

Sınıfları bir-birindən ayırmaq üçün qiymət olaraq hər sınıfa uyğun gələn bir tam ədəd verilmişdir.

Bunlar aşağıdakılardır:

Sınıf Adı	Simvolik Sabit	Ədədi Qiyməti
Array	arrayClass	16
Association	associationClass	15
Bag	bagClass	12
Btree	btreeClass	22
Collection	collectionClass	10
Container	containerClass	6
Date	dateClass	21
Deque	dequeClass	9
Dictionary	dictionaryClass	14
DoubleList	doubleListClass	19
DoubleListElement	doubleListElementClass	5
Error	errorClass	1
HashTable	hashTableClass	11
List	listClass	18
ListElement	listElementClass	4



Sınıf Adı	Simvolik Sabit	Ədədi Qiyməti
Object	objectClass	0
PriorityQueue	priorityQueueClass	23
Queue	queueClass	8
Set	setClass	13
Sortable	sortableClass	2
SortedArray	sortedArrayClass	17
Stack	stackClass	7
String	stringClass	3
Time	timeClass	20

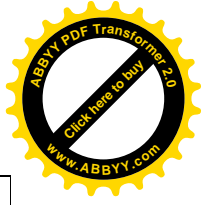
0 ilə `__lastLibClass` (qiyməti 255) arasındakı qiymətlər Borland tərəfindən istifadə edilmək üçün nəzərdə tutulmuşdur. `__firstUserClass` (qiyməti 256) ilə `__lastClass` (qiyməti 65568) arasındakı qiymətlər isə proqramçıların öz siniflərini təyin edərkən istifadə edə bilmələri üçün nəzərdə tutulmuşdur.

Yuxarıdakı cədvəldən də göründüyü kimi simvolik sinif qiymətləri sinfin adının ilk hərfinin kiçik yazılması və sonuna `Class` sözünün əlavə edilməsi ilə göstərilmişdir.

hashValueType

```
typedef unsigned int hashValueType;
```

Yaddaşdakı obyektlərə müraciət etmək məqsədilə obyektlərin müəyyən bir qiymətə görə sinifləndirilməsi və bu sinifləndirməyə görə axtarılması üçün onların verdiyi bir ədədi qiymətdir.



sizeType

```
typedef unsigned int sizeType;
```

Yaddaşda saxlanılan obyektlərin sayını təyin etmək üçün istifadə edilir.

iterFuncType

```
typedef void (*iterFuncType)(class Object&, void*);
```

`forEach` üzv funksiyasının təyininə istifadə edilir.

condFuncType

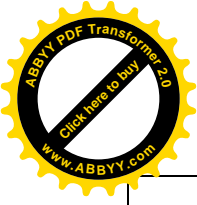
```
typedef int countType;
```

Yaddaşdakı elementləri saymaq üçün istifadə olunan bir tip təyini.

Burada göstərilən bütün tiplər və sabitlər `clstypes.h` başlıq faylı daxilində təyin edilmişdi.

8.3.2 Səhv Kodlarının Təyini

Burada şərh edilən obyekt siniflərinin tətbiqində proqramın icrasının davam etməsinə əngəl olan səhvlər meydana gəldiyi zaman proqramdan `exit()` əmri ilə çıxılır.



Meydana gələn səhvin səbəbinə görə `exit()` funksiyasının (eyni zamanda proqramın) çıxış kodları və ədədi qiymətləri aşağıda göstərilmişdir:

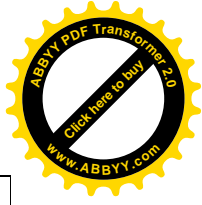
<code>__EEXPAND</code>	2	
<code>__ENOMEM</code>	3	Yaddaşda kifayət qədər yer olmadığı zaman sahə ayrıldığı hallarda
<code>__ENOTSTOP</code>	4	SortedArray sinfinin sıralana bilməməsi halında
<code>__ENOTASSOC</code>	5	Dictionary sinfinin hər hansı bir obyektinə Association sinfindən olmayan bir obyektin yerləşdirəlməsinə cəhd edilməsi halında

Bu səhv kodları "`clsdef.h`" başlıq faylı daxilində təyin edilmişdir.

8.3.3 Başlıq Faylları və Təyin Edilmiş Siniflər

Siniflərin təyin edildikləri başlıq faylları aşağıdakılardır:

Sinif Adı	Simvolik Sabit
<code>AbstractArray</code>	<code>ABSTARRY.H</code>
<code>Array</code>	<code>ARRAY.H</code>
<code>Association</code>	<code>ASSOC.H</code>
<code>Bag</code>	<code>BAG.H</code>
<code>BaseDate</code>	<code>LDATE.H</code>
<code>BaseTime</code>	<code>LTIME.H</code>
<code>Btree</code>	<code>BTREE.H</code>

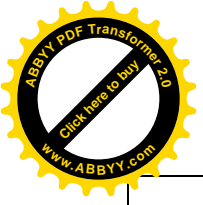


Sinif Adı	Simvolik Sabit
<code>BtreeIterator</code>	<code>BTREE.H</code>
<code>Collection</code>	<code>COLLECT.H</code>
<code>Container</code>	<code>CONTAIN.H</code>
<code>ContainerIterator</code>	<code>CONTAIN.H</code>
<code>Date</code>	<code>DATE.H</code>
<code>Deque</code>	<code>DEQUE.H</code>
<code>Dictionary</code>	<code>DICT.H</code>
<code>DoubleList</code>	<code>DBLLIST.H</code>
<code>DoubleListElement</code>	<code>DLSTELEM.H</code>
<code>Error</code>	<code>ERROR.H</code>
<code>HashTable</code>	<code>HASHTBL.H</code>
<code>HashTableIterator</code>	<code>HASHTBL.H</code>
<code>List</code>	<code>LIST.H</code>
<code>ListElement</code>	<code>LSTELEM.H</code>
<code>ListIterator</code>	<code>LIST.H</code>
<code>Object</code>	<code>OBJECT.H</code>
<code>PriorityQueue</code>	<code>PRIORITYQ.H</code>
<code>Queue</code>	<code>QUEUE.H</code>
<code>Set</code>	<code>SET.H</code>
<code>Sortable</code>	<code>SORTABLE.H</code>
<code>SortedArray</code>	<code>SORTARRY.H</code>
<code>Stack</code>	<code>STACK.H</code>
<code>String</code>	<code>STRING.H</code>
<code>Time</code>	<code>TIME.H</code>

8.4 Siniflər

8.4.1 Object

`Object` mücərrəd sinif (`abstract class`) olmaqla bərabər digər bütün `container class` kitabxanasının obyektlərinin törənməsində istifadə edilən ilk baza



sinfidir. Bu sinif törənən digər siniflərin sadə xüsusiyyətlərini təyin edir.

Bu sinif daxilində təyin edilən üzv funksiyalarının prototipləri və vəzifələri aşağıdakılardır:

```
virtual classType isA() const = 0;
```

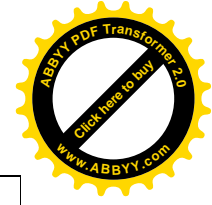
Bu funksiya bir sinfi təyin edən və sadəcə o sinfə aid olan obyektlərin daşıya biləcəyi xüsusi bir kodu qiymət kimi geri qaytarır. Bu funksiya obyektlərin eyni sinifdən olub olmadıqlarını müəyyənləşdirmək üçün istifadə edilir. Yeni törədilən **Object** bazalı hər obyekt üçün yazılması vacibdir.

```
virtual char* nameOf() const = 0;
```

Bu funksiya obyektin xüsusi adı ilə geri qaydır. Sinfin obyektlərinin adı yoxdursa, onun adı ilə geri qaydır. Yeni törədilən **Object** bazalı hər obyekt üçün yazılması vacibdir.

```
virtual hashValueType hashValue() const = 0;
```

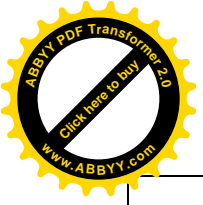
Məlumatlar strukturunun bir qrup daxilində saxlanması müxtəlif formalarda ola bilər. Saxlanılan məlumata müraciət müddəti bu formanın təyin edilməsində çox əhəmiyyətlidir. Bunun üçün bir çox məlumatlar strukturu və müraciət mexanizmləri təyin



edilmişdir. Bu mexanizmlərdən biri də **hashing** adlandırılan qruplaşdırma üsuludur. Bu üsulda məlumatlar bir neçə qrupa ayrılır və hər hansı bir məlumata bu qrupdan yalnız birinin daxilində olma haqqı verilir. Bir məlumata müraciət etmək istədiyiniz zaman o məlumatın hansı qrup daxilində ola biləcəyini bildikdə, onu yalnız o qrup daxilində axtararsınız. Bu müraciət müddətini çox qısaltır.

Məsələn, məlumatlar strukturunda ada görə axtarış aparıldığı zaman məlumatlar adın ilk hərfinə görə qruplaşdırılırsa, bütün məlumatları 32 müxtəlif qrupa ayırmaq lazımdır. Belə bir məlumatlar strukturunda "**Kənan Seyidzadə**" adlı məlumata müraciət etmək istəyərsinizsə, sadəcə, "**K**" qrupuna baxmağınız kifayətdir. Bu misalda "**Kənan Seyidzadə**" adlı bir məlumatın qrup müraciət kodu (**hash value**) "**K**"-dir.

Bu üsul **container class** kitabxanası daxilində **HashTable** sinfi tərəfindən tətbiq edilir. Bu sinfin proqramçıdan gözlədiyi yalnız bir şey isə, **HashTable** və ya bu sinifdən törənən siniflər daxilində istifadə ediləcək məlumat siniflərinin obyektlərinin saxladığıları məlumatı təmsil edən qrup nömrəsi vermələridir. **Object** sinfindən törənən hər obyektin bu qiyməti verə bilməsi üçün **hashValue()** üzv funksiyasından istifadə edilir. Bu funksiya obyektin daxilində olması lazım gələn qrupun nömrəsi ilə geri qaydır. **Object** sinfindən törənən hər obyekt üçün bu üzv funksiyasının yazılması vacibdir.

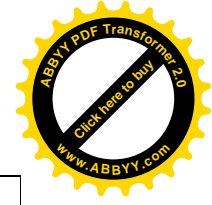


```
virtual int isEqual(const Object& Test) const = 0;
```

Object sinfindən törənən hər sinif üçün təyin edilməsi vacib olan bir digər üzv funksiya isə **isEqual** funksiyasıdır. Bu funksiya bir parametri olan **Object** sinfinin **Test** obyektı ilə əsas obyektin eyni qiymətlərə malik olub olmadığına nəzarət edir. Bu funksiya iki **Object** obyektinin bərabər olub olmadığına nəzarət etmək üçün istifadə olunur. Bunun üçün obyektlərin ilk növbədə eyni sinifdən olması vacibdir. Bu nəzarət **operator ==** vasitəsilə olunur. Ancaq eyni sinifə daxil olan iki obyekt müqayisə edilərsə, **isEqual** sinfi çağırılır. Bu da **Test** parametrinin **isEqual** sinfinin aid olduğu sinif ilə eyni sinifdən olması mənasına gəlir. Bu halda **Test** tip çevrilməsi yerinə yetirilərək üzv dəyişənləri müqayisə edilməlidir. Bu müqayisə nəticəsində iki obyektin eyni qiymətə malik olmasına qərar verilsə **1**, müxtəlif qiymətlərə malik olduqlarına dair qərar verilsə, **0** qiyməti geri qaytarılır.

```
virtual int isSortable() const;
```

Bu funksiya obyektin (sinfin) sıralanıb sıralanmayacağını təyin edən funksiyadır. Əgər bir sinfin obyektləri sıralana bilərsə **1**, əks halda **0** qiyməti bu funksiyadan geri qaytarılır. Bu qayda ilə obyektləri iki hissəyə ayırmaq mümkündür: sıralana bilən obyektlər və



sıralanmayan obyektlər. Bunlardan sıralanmayan obyektlər **Object**, sıralana bilən obyektlər isə **Sortable** sinfindən törənən siniflərdən əldə edilir. **Object** sinfi bu funksiya üçün **0**, **Sortable** sinfi isə **1** qiymətini geri qaytarır. Bu səbəbdən, bu funksiya hər sinif üçün yenidən yazılmalıdır.

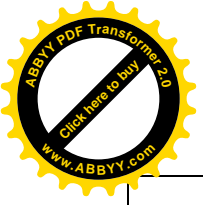
```
virtual int isAssociation() const;
```

isSortable funksiyasına oxşar olaraq, **isAssociation** funksiyası bir obyektin iki obyekt arasındakı əlaqəni (**Association**) göstərən obyekt olub olmadığını, yəni bu obyektin strukturunda bir-biri ilə əlaqələndirilmiş iki müxtəlif obyektin olub olmadığını bildirir. Normal olaraq əlaqəli məlumatlardan ibarət obyektlər **Association** obyektindən törəndiyi üçün digər siniflər üçün bu funksiya **0** qiymətini geri qaytarır.

```
virtual void forEach(iterFuncType Func, void* Data) const;
```

Bu funksiya da yenidən yazılması lazım olmayan bir funksiyadır. **forEach** ilk parametri olan **Func** funksiyasını çağırır. Çağırılan **Func** funksiyasının iki parametrindən birincisi obyektin özü, ikincisi isə **forEach** funksiyasının ikinci parametri olan **Data** göstəricisidir.

Bu funksiya həqiqətən də bir çox obyektin daxil olduğu məlumatlar strukturu üzərində bu struktura daxil



olan bütün obyektlərə eyni funksiyanın (*Func*) tətbiq edilməsini, *Data* göstəricisi isə bu funksiyanın ehtiyac duyduğu qiymətlərin funksiya müraciət etməsini təmin edir.

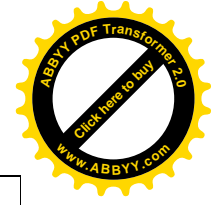
```
virtual Object& firstThat(condFuncType Func, void* Data)
const;
```

firstThat funksiyası da *forEach* funksiyasına oxşar olaraq *Func* nəzarət funksiyasını çağırır. Obyektin özünü birinci, *Data* göstəricisini də ikinci parametr kimi bu funksiya göndərir. *Func* funksiyasından geri qaytarılan qiymət sıfırdan fərqli olarsa, obyekti (**this* qiymətini), sıfır olarsa, *NOOBJECT* qiymətini geri qaytarır.

Bu funksiya bir obyektin obyekt qrupu daxilində müəyyən şərtlərə uyğun olaraq məlumatlar strukturu daxilindəki yerləşməsinə görə ilk obyekt olub olmadığını anlamaq üçün istifadə edilir.

```
virtual Object& lastThat(condFuncType Func, void* Data)
const;
```

Bu funksiya *firstThat* funksiyasının işləmə prinsipinə uyğun olaraq icra olunur. Yalnız obyektin məlumatlar qrupu daxilində yerləşməsinə görə müəyyən olunmuş şərtləri təmin edən ən son obyekt olub olmadığını anlamaq üçün istifadə edilir.



```
virtual void printOn(ostream& Stream) const = 0;
```

Bu funksiya təyin olunan hər yeni sinif üçün yazılması vacib olan bir funksiya. *Stream* ilə təyin olunan axına obyektin ifadə etdiyi məlumatların yazılmasını təmin edir.

```
friend ostream& operator<<(ostream& Stream, const Object&
obyekt);
```

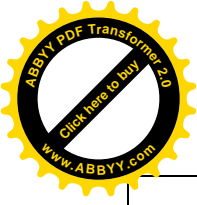
Təyin olunan bu operator funksiyası ilə *Object* sinfindən törənən bütün siniflərin obyektlərinin *ostream* və bundan törənən axınlar üzərinə yazıla biləcəyini göstərir. Bu operator funksiyası

```
obyekt.printOn(Stream);
```

şəklində təsirə səbəb olur.

```
int operator ==(const Object& Test1, const Object& Test2);
int operator !=(const Object& Test1, const Object& Test2);
```

Bu iki operator funksiyası *Test1* və *Test2* *Object* obyektlərinin müqayisəsini təmin edir. Bu funksiyalardan birincisi bərabər olmaları halında, digəri isə bərabər olmamaları halında sıfırdan fərqli bir qiymət ilə geri qaydır. Bir misala baxaq.



```
//IKILIK.CPP

#include <conio.h>
#include <object.h>
#include <cltypes.h>

#define ikilikClass __firstUserClass

class Ikilik : public Object
{ int x, y;

public:
    Ikilik(int = 0, int = 0);

    int X() const { return x; }
    void X(int);

    int Y() const { return y; }
    void Y(int);

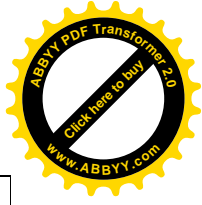
    classType isA() const;
    char* nameOf() const;
    hashValueType hashValue() const;
    int isEqual(const Object&) const;
    void printOn(ostream&) const;
};

Ikilik::Ikilik(int a, int b)
{ x = a;
  y = b;
}

void Ikilik::X(int a)
{ x = a; }

void Ikilik::Y(int b)
{ y = b; }

classType Ikilik::isA() const
```



```
{ return ikilikClass; }

char* Ikilik::nameOf() const
{ return "Ikilik"; }

hashValueType Ikilik::hashValue() const
{ return 0; }

int Ikilik::isEqual(const Object& Test) const
{ return ((Ikilik&)Test).x == x && ((Ikilik&)Test).y == y; }

void Ikilik::printOn(ostream& Stream) const
{ Stream<<nameOf()<<" "<<x<<" "<<y<<"\n"; }

//*****
struct Miqdar{
    int Dx, Dy;
};

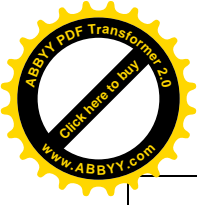
void Surustur(Object& _Obyekt, void* _Data)
{ Ikilik& Obyekt = (Ikilik&)_Obyekt;
  struct Miqdar *Data = (struct Miqdar*)_Data;

  Obyekt.X(Obyekt.X() + Data->Dx);
  Obyekt.Y(Obyekt.Y() + Data->Dy);
}

//*****
main()
{ clrscr();
  Ikilik A(30, 10);
  Ikilik B(40, 50);
  struct Miqdar SurusturmeMiqdari = { 10, 40 };

  cout<<A<<"\n\tA obyektı B obyektine ";

  if(A == B) cout<<"beraberdır\n";
  else cout<<"beraber deyildir\n";
```



```
A.forEach(Surustur, &SurusturMeMiqdari);

cout<<A<<"\n\tA obyektine ";

if(A == B) cout<<"beraberdır\n";
else cout<<"beraber deyildir\n";

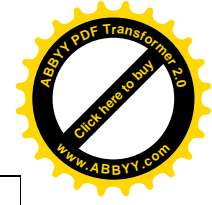
return 0;
}
```

8.4.2 Error

Error sinfi **Class** kitabxanası daxilində meydana gələn səhv vəziyyətlərini həqiqi qiymətlərdən ayırmaq üçün istifadə edilən köməkçi sinifdir. Məsələn, bir **Object** massivinin boş olan, hələ obyekt yerləşdirilməmiş elementlərinə **Error** sinfinin obyektləri mənimsədilərək bu hal müəyyənləşmiş olur. Ümumiyyətlə bu əməliyyat üçün istifadə olunan **NOOBJECT** bu sinfin bir obyektidir. **NOOBJECT** obyektinin təyin edilməsi üçün bu sinif törədilmişdir. **Error** sinfi **Object** sinfinin xüsusiyyətlərini təhvil almaqla yanaşı bəzilərini də dəyişdirmişdir.

Error sinfi ilə bərabər yenidən yazılan üzv funksiyaların davranışları aşağıdakı kimidir:

Üzv funksiya	Qiyməti
isA()	errorClass
nameOff()	"Error"



8.4.3 Sortable

Sortable öz aralarında sıralana bilən obyektlərin siniflərinin təyin edilməsi üçün nəzərdə tutulmuş bir sinifdir. **Sortable** sinfi çox istifadə olunmur. Sadəcə sıralana bilən obyektlərin törədilməsi üçün hazırlanmış baza sinfidir.

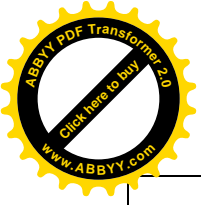
Törəndiyi obyekt sinfindən fərqli davranan üzv funksiyaları və geri qaytardığı qiymətlər aşağıdakılardır:

Üzv funksiya	Qiyməti
isA()	sortableClass
isSortable()	1

Burada **isSortable()** funksiyasının **1** qiymətini qaytardığına diqqət edin. Bundan sonra **Sortable** sinfindən törədiləcək siniflər sıralana bilən obyektlərin sinifləri olacaqları üçün bu sinifləri törədərkən **isSortable()** funksiyasını yenidən yazmaq məcburiyyətində qalmayacağıq. Lakin bu dəfə obyektlərin hansının kiçik, hansının böyük olduğunu müəyyənləşdirmək üçün **isLessThan** funksiyasına ehtiyac duyacağıq.

```
virtual int isLessThan(const Object& Test) const;
```

Bu funksiya **isEqual** funksiyasına oxşar icra olunur. **isLessThan** funksiyası təyin edilməkdə olan əsas sinif ilə



eyni sinifdən olan **Test** obyektinin müqayisəsini apararaq əsas obyektin **Test** obyektindən kiçik olması (sıralamada daha əvvəl yerləşməsi) halında sıfırdan fərqli bir qiymət ilə, böyük və ya bərabər olması halında isə sıfır qiyməti ilə geri qaytarılır.

Sortable sinfi üçün **isLessThan** funksiyasına əsaslanaraq **<=**, **<**, **>**, **>=** operatorları yenidən təyin edilmişdir. Bu operatorları **Sortable** sinfi və bu sinifdən törənən digər siniflər üçün istifadə etmək mümkündür.

8.4.4 String

String sinfi proqramlarda çox istifadə edilən hərf-rəqəm ifadələrinin **C++** ilə **Class** kitabxanası daxilində yenidən təyin edilməsi üçündür. **String** **Sortable** sinfindən törənmiş bir sinifdir.

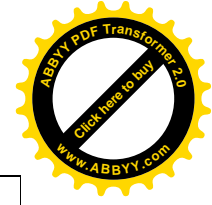
İki müxtəlif layihələndiricidən ibarətdir:

```
String(const char *S);
```

Bu layihələndirici **S** ilə verilən hərf-rəqəm ifadəsinin bir nüsxəsini obyekt daxilində saxlayır.

```
String(String& S);
```

Bu layihələndirici isə əvvəlcədən təyin edilmiş bir **String** obyektinin saxladığı hərf-rəqəm qiymətinin təyin



ediləcək yeni hərf-rəqəm sinfinə köçürülməsini təmin edir.

```
~String();
```

Bu sinfin sahib olduğu yoxedici funksiya layihələndirici tərəfindən hərf-rəqəm sahəsi üçün ayrılan yaddaşın boşaldılmasını təmin edir.

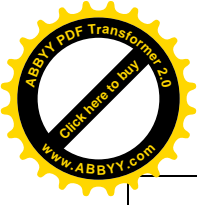
```
operator const char *() const;
```

Bu sinif eyni zamanda bir də (**char***) tip çevirmə operatoruna malikdir. Bu operatorun köməyi ilə **String** sinfinin obyektləri asanlıqla simvol göstəricisi olaraq istifadə oluna bilirlər. Bu operator daha çox **String** sinfinin obyektlərinin hərf-rəqəm qiymətini öyrənmək üçündür.

String sinfi bir axına yazılırsa, əslində saxladığı hərf-rəqəm qiymətini axına yazır. **String** sinfinin obyektlərinin bərabərliyi isə saxladıkları hərf-rəqəm qiymətlərinin bərabərliyi ilə təyin olunmuşdur.

Təyin olunmuş digər iki funksiya və qaytardığı qiymətlər aşağıdakılardır:

Üzv funksiya	Qiyməti
isA()	stringClass
nameOf()	"String"



```
//DNSTRING.CPP

#include <conio.h>
#include <strng.h>
#include <iostream.h>
#include <string.h>

main()
{ clrscr();
  char Buffer[80];

  cout<<endl<<"Adiniz :";
  cin>>Buffer;
  String Ad(Buffer);

  cout<<"Soyadiniz :";
  cin>>Buffer;
  String Soyad(Buffer);

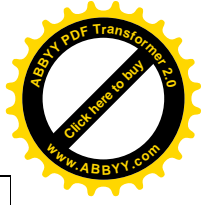
  cout<<endl<<Ad<<"\tuzunlugu " <<strlen(Ad)<<endl;
  cout<<Soyad<<"\tuzunlugu " <<strlen(Soyad)<<endl;

  cout<<Ad<<"=="<<Soyad<<"? "
    <<((Ad==Soyad)?"He":"Yox")<<endl;

  cout<<Ad<<" "<<Soyad<<"? "
    <<((Ad<Soyad)?"He":"Yox")<<endl;

  Ad="Kenan";
  cout<<"\nYeni kimlik-->\t"<<Ad<<" "<<Soyad<<endl;

  return 0;
}
```



Birinci icranın nəticəsi:

```
Adiniz :Etibar
Soyadiniz :Seyidov

Etibar    uzunlugu 6
Seyidov   uzunlugu 7
Etibar==Seyidov? Yox
Etibar<Seyidov? He

Yeni kimlik--> Kenan Seyidov
```

İkinci icranın nəticəsi:

```
Adiniz :Memmed
Soyadiniz :Eliyev

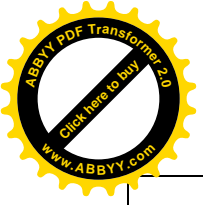
Memmed    uzunlugu 6
Eliyev     uzunlugu 6
Memmed==Eliyev? He
Memmed<Eliyev? Yox

Yeni kimlik--> Kenan Eliyev
```

8.4.5 BaseDate

BaseDate tarix məlumatlarının bir obyekt kimi saxlanması üçün **Sortable** sinfindən törənmiş birbaşa istifadə edilməyən, tarixlə əlaqədar törədilə biləcək siniflərə baza yaratması üçün hazırlanmış bir sinifdir.

Qorunmuş (**protected**) kimi təyin edilmiş üç layihəndiricisi vardır.



```
BaseDate(unsigned int Ay, unsigned int Gun, unsigned int Il);
```

İstənilən il (*Il*), ay (*Ay*) və günün (*Gun*) tarix olaraq yeni yaradılan obyektə saxlanmasını təmin edir. Səhv qiymətin daxil edilməsi proqramın qırılmasına səbəb olur.

```
BaseDate();
```

Obyektin yaradıldığı cari tarixi saxlayır.

```
BaseDate(BaseDate& date);
```

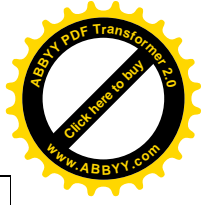
Əvvəlcədən təyin olunmuş tarix obyektinə əsaslanaraq yeni yaradılan tarix obyektinə əvvəlki obyektin göstərdiyi tarixin qiymətini mənimsədir.

Obyektin göstərdiyi tarixin müəyyən edilməsi üçün ümumi (*public*) kimi aşağıdakı üzv funksiyaları təyin edilmişdir:

```
unsigned Day() const;
```

Obyektin göstərdiyi tarixin gününü 1-dən 31-ə qədər bir ədədlə ifadə edir.

```
unsigned Month() const;
```



Obyektin göstərdiyi tarixin ayını 1-dən 12-yə qədər bir ədədlə ifadə edir.

```
unsigned Year() const;
```

Obyektin göstərdiyi tarixin ilini ifadə edir.

Mövcud bir tarix obyektinin göstərdiyi tarixin dəyişdirilməsi tələb olunarsa, bu dəfə yenə ümumi (*public*) kimi təyin olunmuş aşağıdakı üç funksiya istifadə edilir:

```
void SetDay(unsigned char Gun);
```

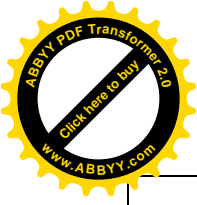
Bu üzv funksiyası tarixə mənimsədilən yeni günün 1-dən 31-ə qədər qiymət alması şərtlə, tarixin yalnız gününü dəyişdirir.

```
void SetMonth(unsigned char Ay);
```

Bu üzv funksiyası tarixə mənimsədilən yeni ayın 1-dən 12-yə qədər qiyməti olmaqla, tarixin yalnız ayını dəyişdirir.

```
void SetYear(unsigned char Il);
```

Bu üzv funksiyası da *Il* ilə göstərilən ili tarixin ilinə mənimsədir.



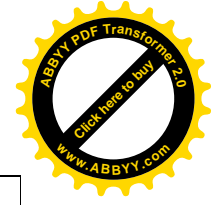
Bundan başqa **BaseDate** sinfi iki tarixin qiymətini müqayisə edir. Əgər iki tarixin il, ay və gün qiymətləri bir-birinə bərabərdirsə, hər iki tarix bərabər sayılır. Sıralama baxımından tarixlər müqayisə edilərsə, // qiyməti böyük olan tarix daha böyükdür. // qiymətləri də eyni olduğu halda **Ay**, **Ay** qiymətləri eyni olduğu halda isə **Gun** qiymətlərinə baxılaraq **Gun** qiyməti daha böyük olan tarixin böyük olduğu qəbul edilir.

8.4.6 Date

BaseDate sinfindən törənən **Date** sinfinin **BaseDate** sinfindən yeganə fərqi **printOn** funksiyasının yenidən yazılmasıdır. Bu da **Date** sinfinin çox istifadə olunmasını təmin edir. Bu funksiya **Date** sinfinin obyektlərinin "ay, gün, il" formatı ilə başlamasını təmin edir. Məsələn, "13-01-1970" tarixi "January, 01, 1970" olaraq göstərilən axına yazılır. Ay adı göründüyü kimi ingilis dilindədir.

Date sinfinin **BaseDate** sinfində olduğu kimi üç layihələndiricisi vardır. Bu layihələndiricilər **BaseDate** ilə eyni funksiyaları yerinə yetirirlər. Yalnız bu layihələndiricilər **public** səviyyəsində təyin edilmişdirlər.

Üzv funksiya	Qiyməti
isA()	dateClass
nameOf()	"Date"



```
//DNDATE.CPP

#include "tarix.h"
#include <conio.h>
#include <ldate.h>
#include <iostream.h>

Date Today;
Date BirthdayOfKenan(10, 18, 1997);
Date BirthdayOfKamran(10, 27, 1996);

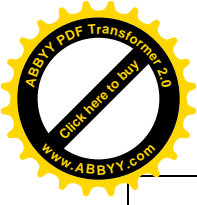
Tarix Bugun;
Tarix KenaninDogumGunu(18, 10, 1997);
Tarix KamraninDogumGunu(27, 10, 1996);

main()
{ clrscr();

  cout<<endl<<"Date"<<endl;
  cout<<"Bugun"<<Today<<endl;
  cout<<"Kenan"<<BirthdayOfKenan<<endl;
  cout<<"Kamran"<<BirthdayOfKamran<<endl;
  if(BirthdayOfKenan == BirthdayOfKamran)
    cout<<"Kenan ile Kamran eyni gunde dogulmusdur.\n";
  else if(BirthdayOfKenan < BirthdayOfKamran)
    cout<<"Kenan Kamrandan daha evvel dogulmusdur.\n";
    else cout<<"Kenan Kamrandan daha sonra dogulmusdur.\n";

  cout<<endl<<"Tarix"<<endl;
  cout<<"Bugun"<<Bugun<<endl;
  cout<<"Kenan"<<KenaninDogumGunu<<endl;
  cout<<"Kamran"<<KamraninDogumGunu<<endl;
  if(KenaninDogumGunu == KamraninDogumGunu)
    cout<<"Kenan ile Kamran eyni gunde dogulmusdur.\n";
  else if(KenaninDogumGunu < KamraninDogumGunu)
    cout<<"Kenan Kamrandan daha evvel dogulmusdur.\n";
    else cout<<"Kenan Kamrandan daha sonra dogulmusdur.\n";

  return 0;
```



```
}
```

```
//TARIX.H
```

```
#ifndef __TARIX_H
#define __TARIX_H

#include <date.h>

#define tarixClass 10000

class Tarix:public BaseDate
{
public:
    Tarix();
    Tarix(unsigned int, unsigned int, unsigned int);

    virtual classType isA() const;
    virtual char* nameOf() const;
    virtual void printOn(ostream&) const;

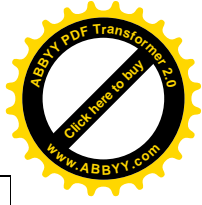
protected:
    int day() const;
    unsigned long julday() const;
};

#endif
```

```
//TARIX.CPP
```

```
#include "tarix.h"
#include <math.h>
#include <iomanip.h>

Tarix::Tarix()
```



```
{}
```

```
Tarix::Tarix(unsigned int Gun, unsigned int Ay, unsigned int Il)
:BaseDate(Ay, Gun, Il)
{}

classType Tarix::isA() const
{ return tarixClass; }

char* Tarix::nameOf() const
{ return "Tarix"; }

#define MaxBufferLen 30

void Tarix::printOn(ostream& Stream) const
{ static char *AyAdlari[] = {"Yanvar", "Fevral", "Mart", "Aprel",
    "May", "Iyun", "Iyul", "Avqust", "Sentyabr",
    "Oktyabr", "Noyabr", "Dekabr"};

    static char *GunAdlari[] = {"Bazartesi", "Cersembe axsami",
    "Cersembe",
        "Cume axsami", "Cume", "Sembe", "Bazar"};

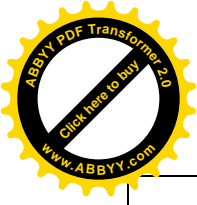
    char buffer[MaxBufferLen];
    ostream ostr(MaxBufferLen, buffer);
    ostr<<setw(2)<<Day()<<" "<<AyAdlari[Month() - 1]<<" "
        <<setw(4)<<Year()<<" "<<GunAdlari[day()]<<ends;

    Stream<<buffer;
}

#define IGREG (15 + 31L * (10 + 12L * 1582))

unsigned long Tarix::julday() const
{ unsigned long jul;
    int ja, jy, jm, iyyy, mm;

    iyyy = Year();
    mm = Month();
```



```
if(mm > 2)
{ jy = iyyy;
  jm = mm + 1;
}
else
{ jy = iyyy - 1;
  jm = mm + 13;
}

jul = (unsigned long) (floor(365.25 * jy) +
  floor(30.6001 * jm) + Day() + 1720995L);

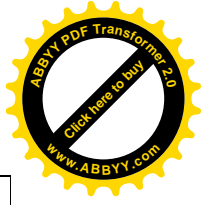
if(Day() + 31L * (Month() + 12L * iyyy) >= IGREG)
{ ja = 0.01 * jy;
  jul += 2 - ja + (int)(0.25 * ja);
}

return jul;
}

int Tarix::day() const
{ return (int)((julday() + 1) % 7); }
```

8.4.7 BaseTime

BaseDate sinfinə oxşar məntiqlə hazırlanan **BaseTime**, **Sortable** sinfindən törənmiş və təyin olunacaq zaman ilə əlaqədar siniflərin törədilməsinə baza yaradan bir sinifdir. **Sortable** üzvlərini istifadə edərkən eynilə **isA()**, **nameOf()** və **printOn()** funksiyalarının yenidən yazılması tələb olunur.



Bu sinif zamanı, saat, dəqiqə, saniyə və saniyənin yüzdə biri mərtəbəsində günlük saxlayır. Saatlar **0-23** arasında, dəqiqə və saniyələr **0-59** arasında, saniyənin yüzdə biri isə **0-99** arasında qiymətlər ala bilər.

Bu sinfin üç **protected** səviyyəsində layihələndiricisi vardır.

```
BaseTime(unsigned char saat, unsigned char deqiqe = 0,
  unsigned char saniye = 0, unsigned char yuzdebir = 0);
```

Günün istənilən bir anını

```
saat:dəqiqə:saniyə.yuzdebir
```

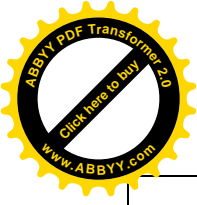
olaraq təyin edir. Səhv qiymətin daxil edilməsi proqramın qırılmasına səbəb olur.

```
BaseTime();
```

Obyektin yaradılma vaxtını obyektə saxlayır.

```
BaseTime(BaseTime& date);
```

Əvvəlcədən təyin edilmiş zaman obyektinə əsaslanaraq yeni yaradılan zaman obyektinə əvvəlki obyektin göstərdiyi zamanın qiymətini mənimsədir.



BaseDate obyektinin göstərdiyi tarixin öyrənilməsi üçün ümumi (**public**) olaraq aşağıdakı üzv funksiyalar təyin edilmişdir:

```
unsigned hour() const;  
unsigned minute() const;  
unsigned second() const;  
unsigned hundredths() const;
```

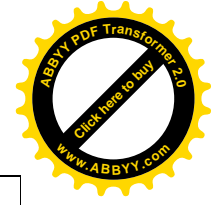
hour saat, **minute** dəqiqə, **second** saniyə, **hundredths** isə saniyənin yüzdə birini müəyyənləşdirmək üçün istifadə edilir.

Mövcud bir zaman obyektinin göstərdiyi zamanın dəyişdirilməsi tələb olunarsa, ümumi (**public**) kimi təyin edilmiş aşağıdakı funksiyalar istifadə edilə bilər:

```
unsigned setHour(unsigned char Saat) const;  
unsigned setMinute(unsigned char Dəqiqə) const;  
unsigned setSecond(unsigned char Saniyə) const;  
unsigned setHundredths(unsigned char Yuzdəbir) const;
```

Bu funksiyalar ardıcıl olaraq zamanın, saat, dəqiqə, saniyə və saniyənin yüzdə birinin qiymətlərinin bir-birindən ayrılıqda dəyişdirilməsinə imkan verirlər.

BaseTime sinfi zaman məlumatlarının bərabərliyini, hər iki zaman obyektinin saat, dəqiqə, saniyə və saniyənin yüzdə birinin qiymətlərinin bərabər olması kimi təyin edilmişdir. Eyni zamanda kiçiklik əlaqəsini də təyin edir.



8.4.8 Time

Time isə **BaseTime** sinfindən törənmiş sadə bir zaman sinfidir. İstifadə olunması baxımından əhəmiyyətlidir.

Time sinfinin ən çox istifadə edilən funksiyaları aşağıdakılardır:

Üzv funksiya	Qiyməti
isA()	timeClass
nameOf()	"Time"

Bu sinfin də **BaseTime** sinfindəkilərə uyğun olan, lakin **public** səviyyəsində üç layihələndirici funksiyası vardır:

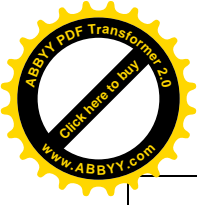
```
Time(unsigned char saat, unsigned char dəqiqə = 0,  
      unsigned char saniyə = 0, unsigned yuzdəbir = 0);
```

Günün istənilən bir anının

saat:dəqiqə:saniyə.yuzdəbir

formatında saxlanılmasını təmin edir. Səhv qiymətin daxil edilməsi proqramın qırılmasına səbəb olur.

```
Time();
```



Obyektin yaradıldığı zamanı obyektə saxlayır.

```
Time(BaseTime& date);
```

Əvvəlcədən təyin edilmiş zaman obyektinə əsaslanaraq yeni yaradılan zaman obyektinə əvvəlki obyektin göstərdiyi zamanın qiymətini mənimsədir.

```
printOn(ostream& Stream) const;
```

funksiyası zamanı ikirəqəmli ədədlər şəklində

SS:DD:ss.YY ZM

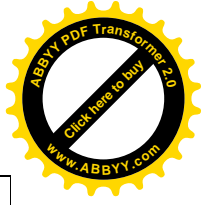
SS saat, DD dəqiqə, ss saniyə, YY saniyənin yüzdə biri və ZM günortadan əvvəl/günortadan sonranı müəyyənləşdirərək Stream axınına yazır. ZM günortadan əvvəl saat 0-12 arasındakı qiymətlər üçün am, günortadan sonra saat 12-24 arasında isə pm şəklində göstərilir.

```
//TIME.CPP
```

```
#include <conio.h>
#include <time.h>
```

```
main(void)
{ clrscr();
  Time Baslangic;
```

```
  cout<<"Proqramdan cixmaq ucun ESC duymesini sixin...\n";
```



```
_setcursortype(_NOCURSOR);

int Davam = 1;
while(Davam)
{ cout<<Time()<<"\r";
  if(kbhit()) Davam = getch() != 27;
}

_setcursortype(_SOLIDCURSOR);
Time Son;

cout<<"\n\nProqramin baslama saati = "<<Baslangic;
cout<<"\nProqramin bitmesi saati = "<<Son;
cout<<endl;

return 0;
}
```

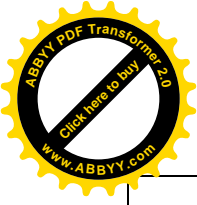
Proqram çıxışı

Proqramdan cixmaq ucun ESC duymesini sixin...
9:47:27.60 pm

Proqramin baslama saati = 9:47:27.20 pm
Proqramin bitmesi saati = 9:47:27.60 pm

8.4.9 Association

Obyekt sinfindən törənmiş Association sinfi struktur olaraq tərkibində iki Object sinfindən törənmiş obyekt (təqdimat kimi) saxlayan bir növ xüsusi obyekt sinfidir. Association sinfi tərkibində təqdimat



məlumatlarını saxladığı bu iki sinif arasında əlaqənin olmasını təyin edir. Bu siniflərdən birincisi açar (**key**), ikincisi isə qiymət (**value**) adlandırılır. **Association** sinfi daha çox bir qrup kimi daxilində müəyyən açara uyğun gələn qiyməti tapmaq üçün **Dictionary** sinfinin təməlini yaratmaq üçün layihələndirilmişdir.

Üzv funksiya	Qiyməti
isA()	associationClass
nameOf()	"Time"

Bu sinfin iki layihələndirici funksiyası vardır:

Association(Object& *key*, Object& *value*);

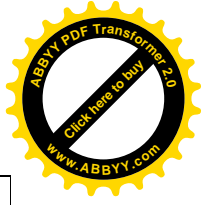
Parametr siyahısındakı **key** (açar) ilə **value** (qiymət) arasında əlaqə olduğunu bilən **Association** obyektini təyin edir.

Association(const **Association**& *association*);

Bu layihələndirici mövcud **association** obyektinə əsaslanaraq eyni əlaqəli **Association** obyektini təyin edir.

Yenidən təyin edilmiş digər üzv funksiyalar və onların vəzifələri aşağıdakılardır:

hashValueType **hashValue**() const;



Association sinifli bir obyektin **hashValue** qiyməti, obyektin əlaqələrini təyin etdiyi obyektlərdən **key** (açar) olaraq ayrılmış obyektin **hashValue** qiymətidir. **Value** (qiymət) obyektini nəzərə alınmaz. **Key** (açar) obyektini bu əlaqə nəticəsində meydana gələn obyektin təkbəşinə təmsil edir. **Association** sinfindən iki obyektin qarşılaşdırılması zamanı eyni cür davranış gözlənilir.

int **isEqual**(const object& *Test*) const;

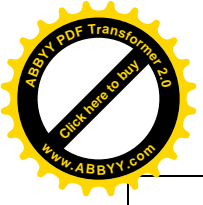
Bu funksiya **Association** sinfinə daxil olan iki obyektini müqayisə edir və onlar bərabər olarsa, sıfırdan fərqli bir qiymət, fərqli olarsa, sıfır qiymətini geri qaytarır. **Association** sinifli iki obyektin bərabərliyi iki **Association** obyektinin də **key** (açar) qiymətlərinin bərabər olması kimi təyin edilmişdir.

void **printOn**(ostream& *Stream*) const;

Association sinfinin obyektlərinin axın üzərinə yazılması normal hallarda

sinif_adı { **key_object**, **value_object** }

formatı ilə olur. **sinif_adı** isə **nameOf()** ilə öyrənilirdiyi üçün **Association** olur. Təbii ki, bu formatı **Association** sinfindən törədiləcək yeni siniflərin **printOn** funksiyasını yenidən yazaraq dəyişdirmək mümkündür.



Association sinfi ilə təyin edilmiş iki yeni funksiya isə **Association** obyektinin aralarında əlaqə qurduğu **key** (açar) və **value** (qiymət) obyektlərinə müraciət edilməsini təmin edir.

Bu iki funksiya aşağıdakı prototipləri ilə təyin edilir:

```
Object& key() const;  
Object& value() const;
```

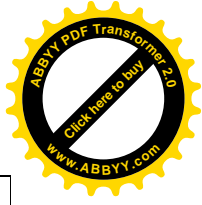
8.5 Məlumatlar Sturukturu Sinifləri

Mövcud proqramlaşdırma dilləri məlumatlar qrupu olan massivləri (**ProLog** və **LISP** siyahı strukturlarını) dəstəkləyir. Bu baxımdan massivlər (cədvəllər də adlandırılır) proqramlaşdırma dili üçün çox əhəmiyyətli bir mövzunu təşkil edir. Məlumatlar sturukturu isə sadəcə, massiv və ya siyahıdan ibarət deyildir. Xüsusi məqsədlər üçün olduğu kimi, ümumi məqsədlər üçün də müxtəlif məlumatlar qrupu təyin etmək olar.

İndi də **Container Class** kitabxanasında təyin olunmuş məlumatlar strukturlarını gözdən keçirək.

8.5.1 Container

Çox istifadə olunmayan **Container** sinfi əsasən məntiqi olaraq müxtəlif tipli (həmsi eyni də ola bilər)



məlumatları tərkibində saxlayan və bu məlumatların yaddaşda saxlanılmasını nizamlayan bir sinifdir. Bu sinif digər yaddaş siniflərinin törədilməsi üçün baza təşkil etdiyi üçün çox əhəmiyyətlidir.

Bu sinif də **Object** sinfindən törəndiyi üçün onun xüsusiyyətlərini miras almış və dəyişdirmişdir.

```
virtual classType isA() const = 0;  
virtual char* nameOf() const = 0;  
virtual hashValueType hashValue() const = 0;
```

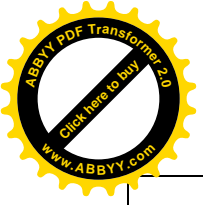
Bu üzv funksiyaları **Object** sinfində olduğu kimi eyni mənanı ifadə edirlər. Lakin **Container** sinfi daxilində kodlaşdırılmayaraq ondan törədiləcək siniflər üçün kodlaşdırılmaları vacibdir.

```
virtual int isEqual(const Object& Test) const;
```

Bu üzv funksiyası isə iki **Container** obyektinin bərabərliyini yoxlayır. Bərabər olmaları üçün hər iki **Container** obyektinin eyni sayda elementdən ibarət olması və eyni mövqedəki element obyektlərinin bərabər olması şərtidir.

Container sinfi ilə birlikdə təyin olunan yeni üzv funksiyaları isə aşağıdakılardır:

```
int isEmpty() const;
```



Container obyektini daxilində heç bir obyekt olmazsa, sıfır qiyməti ilə geri qaydır. Bu funksiya **Container** obyektinin sanki boş olub olmadığını müəyyənləşdirir.

```
countType getItemsInContainer() const;
```

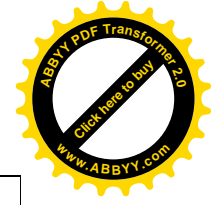
Bu funksiyası isə **Container** obyektinə daxil olan obyektlərin sayını **protected** səviyyəsində təyin etdiyi

```
countType itemsInContainer;
```

dəyişkənində saxlayır. **Container** obyektinə hər dəfə yeni obyekt əlavə edildiyi zaman bu qiymət bir vahid artır. Xaric edilən hər obyekt üçün də bir vahid azalır.

```
virtual void printOn(ostream& Stream) const;  
virtual void printHeader(ostream& Stream) const;  
virtual void printSeperator(ostream& Stream) const;  
virtual void printTrailer(ostream& Stream) const;
```

Container sinfi obyektinin axına yazılmasını təmin edən əsas funksiya **printOn()** funksiyasıdır. **printOn()** **printHeader()** funksiyasını çağıraraq əməliyyata başlayır. **printHeader()** axına göndəriləcək məlumatların əvvəlinə başlıq əlavə edilməsi məqsədilə istifadə olunur. Məlumatların əvvəlinə əlavə olunan bu başlıq **Container** sinfi üçün obyektin adı (**nameOf()** ilə öyrənilir) və "{" işarəsidir. **printOn()** funksiyası daha sonra **Container** daxilindəki obyektləri ardıcıl olaraq yazmağa başlayır.



Yazdığı obyektlər arasında isə **printSeperator()** funksiyasını çağırır. **printSeperator()** isə yazılan obyektlərin bir-birindən ayrılmasını təmin edir. Bunun təsiri **Container** sinfi üçün "\n" mesajının yazılması şəklində görünür. **Container** daxilindəki bütün obyektlər yazıldıqdan sonra da **printTrailer()** funksiyası çağırılaraq yazma əməliyyatı tamamlanır. Bu funksiya da **Container** sinfi üçün "}" simvolunun yazılması şəklində təsir göstərir.

```
virtual void forEach(iterFunctionType iterfunc, void *param);
```

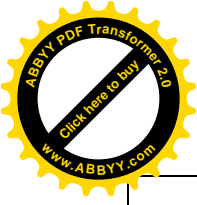
Bu funksiya **Object** sinfinin təyin edilməsindən fərqlənir. **forEach** funksiyası **iterfunc** parametri ilə verilən

```
void iterfunc(Object& obyekt, void *param);
```

şəklində funksiyanın, **Container** sinfinə daxil olan bütün obyektlərə ardıcıl olaraq tətbiq edilməsini təmin edir. **iterfunc** funksiyasının ikinci parametri olan **param** ilə **forEach** funksiyasının ikinci parametri olan **param** eyni qiymətlərdir.

```
virtual Object& firstThat(contFuncType condfunc, void *param)  
const;  
virtual Object& lastThat(contFuncType condfunc, void *param)  
const;
```

Bu funksiyalar



```
int confunc(Object& obyekt, void* param);
```

şəklində təyin edilən və birinci parametr ilə göstərilən bir müqayisə funksiyasına **Container** obyektı daxilində olan obyektləri ardıcıl olaraq tətbiq edir. **condfunc** funksiyası obyektlərin müəyyən olunmuş şərti ödəyib ödəmədiklərini yoxlayır. Uyğun şərti ödəyən obyektlər üçün sıfırdan fərqli qiyməti, şərti ödəməyən obyektlər üçün isə sıfır qiymətini qaytarır.

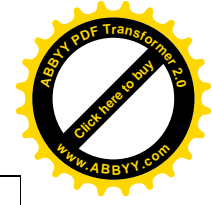
Bu funksiyalardan **firstThan** **condfunc** funksiyasından sıfırdan fərqli bir qiymət ilə geri qaytarılan ilk obyektı, **lastThan** isə son obyektı geri qaytarma qiyməti kimi qaytarır.

```
virtual ContainerIterator& initIterator() const = 0;
```

Bu funksiya isə yeniləyicilər bölməsində şərh ediləcək bir yeniləyici (**iterator**) **Container** sinfinin obyektı üçün hazırlayaraq geri qaytarma qiyməti kimi qaytarır.

8.5.2 Stack

Əsas məlumatlar strukturu olan **Stack**, **Last In First Out – LIFO** (son girən ilk çıxar) qaydası ilə işləyən bir **Container** sinfidir. **Stack** strukturunda yaddaşa yeni yerləşdiriləcək obyekt digər obyektlərin üzərinə



yerləşdirilir. Yaddaşdan bir obyektin geri qaytarılması tələb olunduğu zaman da üstdəki obyekt geri qaytarılır. Araya bir obyekt əlavə etmək və ya aradan bir obyektı çıxarmaq mümkün deyildir.

Stack sinfi **Container** sinfindən törənmişdir.

Yenidən təyin olunduqları funksiyalar aşağıdakılardır:

Üzv funksiya	Qiyməti
isA()	stackClass
nameOf()	"Stack"

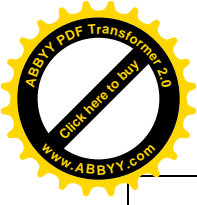
Stack strukturuna yeni əlavə edilən üzv funksiyalar isə aşağıdakılardır:

```
void push(Object& obyekt);
```

Bu funksiya **obyekt** parametri ilə verilən **Object** sinfindən törənmiş bir obyektin **Stack** daxilinə yerləşdirilməsini təmin edir. Yerləşdirmə təbii olaraq **Stack** strukturundakı bütün obyektlərin üzərinə olacaqdır.

```
Object& pop();
```

Bu funksiya **Stack** yaddaşının ən üstdəki obyektı geri qaytarma qiyməti kimi qaytararkən bu obyektin yaddaşdan çıxarılmasına da səbəb olur.



Object& top();

Bu funksiya **Stack** yaddaşının ən üstdəki obyektinin öyrənilməsini təmin edir və obyektı yaddaşdan çıxartmır.

virtual ContainerIterator& initIterator() const;

Bu funksiya **Stack** yaddaşı üzərində bir yeniləyici təyin etmək üçün istifadə olunur. Yeniləyici **Stack** daxilindəki obyektləri son daxil olandan ilk daxil olana qədər yeniləşdirir.

//DNSTACK.CPP

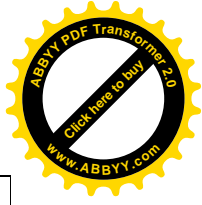
```
#include <conio.h>
#include <stack.h>
#include <string.h>
```

```
main()
{ clrscr();
```

```
    Stack Yigin;
    char oxu[80];
```

```
    cout<<"Verilenlerin girisini tamamlamaq ucun '\.' daxil edin...\n";
    do
    { cout<<"> ";
      cin>>ws>>oxu;
      Yigin.push(*new String(oxu));
    }
    while(oxu[0] != '.');
```

```
    cout<<Yigin<<endl;
```



```
    cout<<"Yiginda "<<Yigin.getItemsInContainer()
    <<" element var.\n";
    while(!Yigin.isEmpty())
    { Object& Obyekt = Yigin.pop();
      cout<<Obyekt<<endl;
      delete &Obyekt;
    }
    cout<<endl;

    return 0;
}
```

Program çıxışı

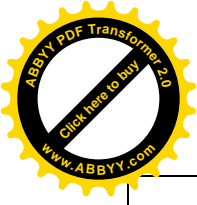
Verilenlerin girisini tamamlamaq ucun '.' daxil edin...

```
> Baki
> Gence
> Quba
> Sumqayit
> .
```

```
List {
    Sumqayit,
    Quba,
    Gence,
    Baki }
```

Yiginda 5 element var.

```
.
Sumqayit
Quba
Gence
Baki
```



8.5.3 Deque

Container sinfindən törənmiş **Deque** sinfi **Stack** sinfinə oxşar formada işləyir. **Deque** sinfində obyekt yaddaşa onun sol (alt) və ya sağ (üst) ucundan başlayaraq yerləşdirilə bilər. **Deque** yaddaşından alınacaq obyekt onun sol və ya sağ ucundakı obyekt ola bilər. Ortadakı bir obyekt **Deque** yaddaşından çıxartmaq mümkün deyildir.

Deque sinfinin **Container** sinfindən alıb dəyişdirdiyi funksiyalar aşağıdakılardır:

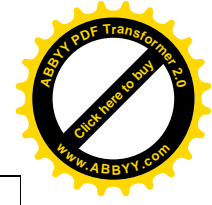
Üzv funksiya	Qiyməti
isA()	dequeClass
nameOf()	"Deque"

`virtual ContainerIterator& initIterator() const;`

Bu funksiya **Deque** yaddaşı üzərində bir yeniləyici təyin etmək üçün istifadə edilir. Yeniləyici **Deque** daxilindəki obyektləri **Deque** yaddaşının sol ucundan sağ ucuna doğru yeniləyir. Əgər obyektlərin tərs ardıcılıqda yenilənməsi tələb olunarsa, bunun üçün

`ContainerIterator& initReverselIterator() const;`

üzv funksiyasından istifadə etmək olar. Bu funksiya virtual deyildir. Sadəcə **Deque** üçün təyin edilmişdir.



Deque üçün təyin edilmiş digər funksiyalar isə aşağıdakılardır:

```
void putLeft(Object& obyekt);  
void putRight(Object& obyekt);
```

Bu funksiyalar obyektlərin **Deque** yaddaşına əlavə edilmələrini təmin edir. Bunlardan **putLeft()** obyektləri yaddaşın sol ucuna, **putRight()** isə sağ ucuna yerləşdirir.

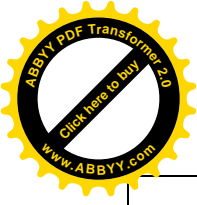
```
Object& peekLeft();  
Object& peekRight();
```

Bu funksiyalar isə **Deque** yaddaşının sağ və sol uclarındakı obyektlərin müəyyənləşdirilməsini təmin edir. **peekLeft()** sol ucundakı, **peekRight()** isə sağ ucundakı obyektləri müəyyənləşdirir.

```
Object& getLeft();  
Object& getRight();
```

Bu funksiyalar uyğun olaraq **peekLeft()** və **peekRight()** kimi icra olunmaları ilə bərabər, uclardakı obyektləri müəyyənləşdirməklə yanaşı bu obyektləri **Deque** yaddaşından çıxarırlar.

```
//DNDEQUE.CPP  
#include <conio.h>  
#include <deque.h>
```

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

main()
{ clrscr();

  Deque Yigin;
  char oxu[80];

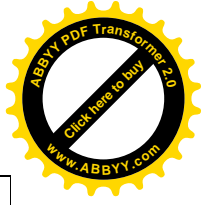
  cout<<"Verilenlerin girisini tamamlamaq ucun \\'\' daxil edin...\n";
  do
  { cout<<"> ";
    cin>>ws>>oxu;
    if(isupper(oxu[0]))
      Yigin.putLeft(*new String(oxu));
    else Yigin.putRight(*new String(oxu));
  }
  while(oxu[0] != '.');

  cout<<Yigin<<endl;
  cout<<"Yiginda "<<Yigin.getItemsInContainer()
    <<" element var.\n";
  while(!Yigin.isEmpty())
  { Object& Obyekt = Yigin.getRight();
    cout<<Obyekt<<endl;
    delete &Obyekt;
  }
  cout<<endl;

  return 0;
}
```

Program çıxışı

```
Verilenlerin girisini tamamlamaq ucun \\'\' daxil edin...
> Kenan
> Ismayil
> Amil
```



```
> Ramil
> Kamil
> .
DoubleList {
  Kamil,
  Ramil,
  Amil,
  Ismayil,
  Kenan,
  . }

Yiginda 6 element var.
.
Kenan
Ismayil
Amil
Ramil
Kamil
```

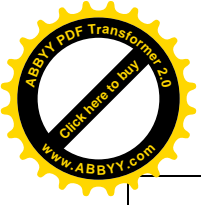
8.5.4 Queue

Deque yaddaşının məhdudlaşdırılmış bir tətbiqi olan **Queue** sinfində obyektlər yaddaşa bir ucdan yerləşdirilib digər ucdan çıxarıla bilər. Başqa sözlə, **Queue** yaddaşının işləmə forması “ilk girən ilk çıxar” (**Firs In First Out – FIFO**) şəkilindədir. **Queue** bir yaddaş növbəsidir.

Queue sinfinin **Container** sinfindən alıb dəyişdirdiyi funksiyalar aşağıdakılardır:

Üzv funksiya
isA()

Qiyməti
queueClass

`nameOf()``"Queue"`

Bu sinif üçün təyin olunmuş üzv funksiyalar aşağıdakılardır:

`void put(Object& obyekt);`

üzv funksiyası obyektı növbənin sonuna əlavə edir.

`Object& get();`

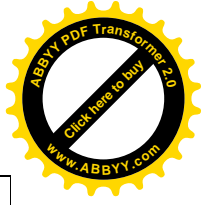
üzv funksiyası isə növbənin əvvəlindəki obyektı növbədən çıxararaq geri qaytarma qiyməti olaraq qaytarır.

`Object& peekLeft();
Object& peekRight();`

isə `Deque` sinfində olduğu kimi `Queue` sinfində növbənin başlanğıcında və sonundakı obyektlərin yalnız müəyyənləşdirilməsi üçün istifadə edilir. Bunlardan `peekLeft()` növbənin sonundakı, `peekRight()` funksiyası isə növbənin əvvəlindəki obyektı göstərir.

8.5.5 PriorityQueue

`PriorityQueue` sinfindən törənmiş xüsusi bir növbə tipidir. `PriorityQueue` növbəsi "ən böyük ilk



çıxar" ("Greatest In First Out" – GIFO) və ya "ən kiçik ilk çıxar" ("Smallest In First Out") – SIFO) qaydalarından biri ilə işləyir. Yəni növbəyə daxil olan obyektlər prioritetə görə sıralanmış olur. Növbədən bir qiymətin alınması lazım gəldiyi zaman prioritetinə görə birinci gələn obyekt alınır.

`PriorityQueue` növbəsi daxilindəki obyektlərin yerləşəcəkləri yer, `isLessThan` funksiyası vasitəsilə müəyyənləşdirilir. Kiçik olan obyekt birincidir. Bir-birinə bərabər olan obyektlər isə "birinci gələn ilk prioritetlidir" qaydası ilə yerləşdirilir. Buna görə də `PriorityQueue` növbəsi daxilində ancaq `Sortable` sinfindən törənmiş siniflər aşağıdakılardır.

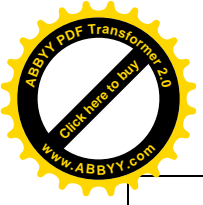
Üzv funksiya	Qiyməti
<code>isA()</code>	<code>priorityQueueClass</code>
<code>nameOf()</code>	<code>"PriorityQueue"</code>

`Object& get();`

Növbənin başlanğıcındakı birinci olan obyektı geri qaytarma qiyməti olaraq qaytararkən bu obyektı eyni zamanda növbədən çıxarır.

`Object& peekLeft();`

`get()` funksiyasına oxşar şəkildə çalışan bu funksiya isə sadəcə prioritetinə görə birinci olan obyektin



müəyyənləşdirilməsi üçün istifadə edilir. Obyekti növbədən çıxarmır.

```
void detachLeft();
```

Bu funksiya isə növbə daxilində birinci olan obyektin sadəcə növbədən çıxarılmasını təmin edir. Obyektin müəyyənləşdirilməsi üçün istifadə edilməz.

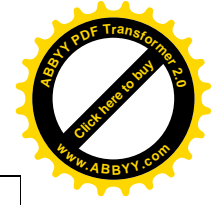
```
void put(Object& obyekt);
```

Bu funksiya parametr siyahısında verilən *obyekt* obyektini, növbə siyahısında prioritetinə uyğun yerdə yerləşdirir.

8.5.6 Collection

Collection sinfi *Conatainer* sinfində tətbiq edilən obyektlərin müəyyən qaydaya görə yaddaşa yerləşdirilib, yenə müəyyən bir qayda ilə yaddaşdan xaric edilməsini nəzərə alaraq obyektlərin yaddaş daxilində axtarılıb tapılması, istənilən zaman çıxarılması, istənilən yerə qoyulması əsasında işləyir. *Collection Container* kimi istifadə olunan bir sinif deyildir. Oxşar siniflərə baza yaradır.

```
virtual classType isA() const = 0;  
virtual char* nameOf() const = 0;
```



```
virtual hashValueType hashValue() const = 0;  
virtual containerIterator& initIterator() const = 0;
```

Container sinfindən alınan bu üzv funksiyalar yenidən yazılaraq *Collection* sinfindən törənəcək siniflərə verilir.

Collection üçün yeni təyin edilən üzv funksiyalar isə aşağıdakılardır:

```
virtual void add(Object& obyekt) = 0;
```

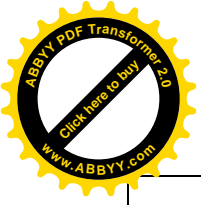
Bu üzv funksiyası yeni bir obyekt kolleksiyaya daxil etmək üçün istifadə edilir.

```
virtual void detach(const Object& obyekt, int tip = 0) = 0;
```

Bu üzv funksiyası isə birinci parametr ilə verilən obyektin kolleksiyadan çıxarılmasını təmin edir. Verilməməsi halında sıfır qiymətini alan *tip* parametrinin sıfır qiyməti üçün obyekt sadəcə kolleksiyadan çıxarıldığı zaman, sıfırdan fərqli qiymətlər üçün isə kolleksiyadan çıxarıldıqdan sonra yaddaşdan da silinəcəkdir.

```
void destroy(const Object& obyekt);
```

destroy funksiyası *detach* funksiyasını *tip* parametri 1 olması ilə çağırır. Yəni *obyekt* obyektinə bərabər



obyekti kolleksiyada taparaq oradan çıxarır və yaddaşdan silir.

```
virtual Object& findMember(const Object& obyekt) const;
```

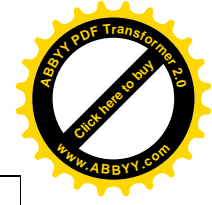
`findMember` üzv funksiyası kolleksiya daxilində *obyekt* obyektinə bərabər olan obyekti axtarır və tapdığı ilk obyekti geri qaytarır. Axtarılan obyekt tapılmazsa, **NOOBJECT** qiymətini qaytarır.

```
virtual int hashMember(const Object& obyekt) const;
```

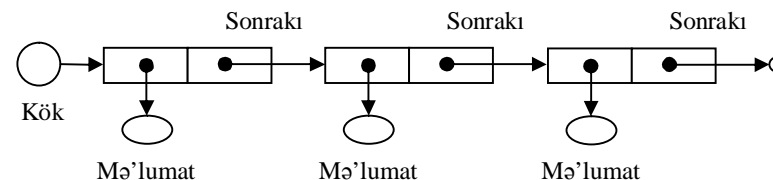
Bu üzv funksiya isə verilən obyektin **Collection** daxilində tapılıb tapılmadığına nəzarət edir. Uyğun obyekt **Collection** daxilində tapılarsa, sıfırdan fərqli bir qiyməti ilə tapılmazsa, sıfır qiymətini qaytarır.

8.5.7 List

Məlumatlar strukturları arasında ən əhəmiyyətliələrindən biri olan **list** strukturu, əsas məlumatı göstərən **təqdimat (referans)** və bu məlumatdan sonra gələn siyahı elementini göstərən bir göstəricidən ibarətdir. Bu siyahı elementlərinin bir-birinin ardınca əlavə edilməsini zəncir formasında siyahı strukturları əldə edilir. Siyahı daxilindəki hər element öz



məlumatına və sonrakı elementə müraciət edə biləcək məlumata malikdir.

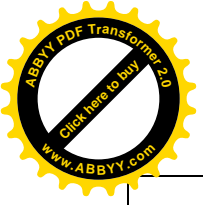


Stack, **Queue**, **Referans** və **Set** kimi digər məlumatlar strukturları da siyahı strukturundan istifadə edərək, sadəcə bu struktura məlumatların daxil/xaric edilməsi ilə əlaqədar məhdudiyyətlər gətirmişdir. Bununla da siyahı strukturları proqramlaşdırma baxımından əhəmiyyətli yer tuturlar.

List sinfinin **Collection** sinfindən miras aldığı və yenidən təyin etdiyi funksiyalar aşağıdakılardır:

Üzv funksiya	Qiyməti
<code>isA()</code>	<code>listClass</code>
<code>nameOf()</code>	"List"
<code>hashValue()</code>	0

```
Object& peekHead() const;
```

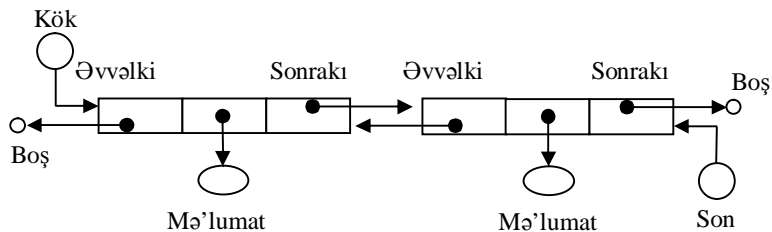


Bu üzv funksiyası siyahıya ən son əlavə edilən obyekti geri qaytarma qiyməti olaraq qaytarır. Siyahı boş olarsa, bu funksiya **NOOBJECT** qiymətini qaytarır.

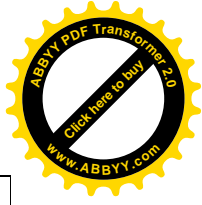
Bunun xaricindəki üzv funksiyalar **Collection**, **Container** siniflərində təyin olunan funksiyalardır. Və bu siniflərin gördüyü əməliyyatları yerinə yetirirlər.

8.5.8 DoubleList

Collection sinfindən törənən **DoubleList** sinfi **List** sinfindən fərqli olaraq məlumatı göstərən məlumat təqdimatı və sonrakı siyahı elementini göstərən sonrakı məlumat təqdimatı ilə bərabər əvvəlki məlumat elementini göstərən əvvəlki məlumat təqdimatına da malikdir.



Bu struktur biristiqamətli siyahı strukturlarına nisbətən daha çox yaddaş tələb edir, lakin hər iki



istiqamətdə də irəliləmək mümkün olduğundan bəzi alqoritmlər üçün daha uyğun bir strukturdur.

Bu sinif daxilində əvvəlki və sonrakı hallar ilə əlaqədar eyni əməliyyatı yerinə yetirən simmetrik iki əmr hər zaman mövcuddur.

```
void addAtHead(Object& obyekt);  
void addAtTail(Object& obyekt);
```

Bu funksiyalar **Collection** üçün təyin edilən **add** əmri kimi icra olunur. Aralarındakı əsas fərq, **addAtHeader** obyekti siyahının əvvəlinə əlavə etdiyi halda, **addAtTail** isə sonuna əlavə edir.

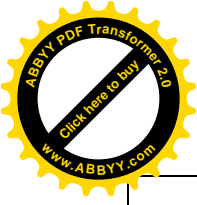
```
void destroyFromHead(const Object& obyekt);
```

Bu funksiya siyahını əvvəldən axıra qədər gözdən keçirərək **obyekt** obyektinə bərabər olan tapdığı ilk obyekti siyahıdan çıxarır və yaddaşdan silir.

```
void destroyFromTail(const Object& obyekt);
```

Bu funksiya isə siyahını axırdan əvvələ qədər gözdən keçirərək tapdığı **obyekt** obyektinə bərabər olan ilk obyekti siyahıdan çıxarır və yaddaşdan silir.

```
void detachFromHead(const Object& obyekt, int tip = 0);  
void detachFromTail(const Object& obyekt, int tip = 0);
```



Bu funksiyalar siyahını gözdən keçirərək (**detachFromHead** əvvəldən axıra qədər, **detachFormTail** axırdan əvvələ qədər) qarşılarına çıxdığı ilk obyektı sadəcə siyahıdan çıxarırlar. *tip* qiymətinin 0-dan fərqli olması obyektin yaddaşdan silinməsi əməliyyatının da eyni zamanda icra edilməsini, yəni bu əməllərin **destroyFromHead** və ya **destroyFromTail** funksiyalarına uyğun icra edilməsini təmin edir.

Object peekAtHead() const;

siyahının əvvəlindəki,

Object peekAtTail() const;

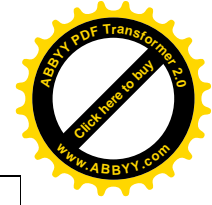
isə siyahının sonundakı obyektin müəyyənləşdirilməsi üçün istifadə edilir.

virtual ContainerIterator& initIterator() const;

Bu funksiya siyahı məlumatlarının siyahının əvvəlindən axırına doğru,

virtual ContainerIterator& initReverseIterator() const;

isə siyahının axırından əvvəlinə doğru yenilənməsini təmin edir.



Collection üçün təyin edilən bəzi funksiyalar **DoubleList** üçün uyğun olaraq icra olunurlar. Bu funksiyaların parametr siyahılarının eyni olmasına nəzər yetirin.

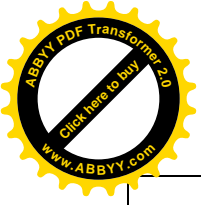
Collection	DoubleList
add	addAtHeader
destroy	destroyFromHead
detach	detachFromHead

Təyin edilən digər klassik funksiyalar və onların qiymətləri isə aşağıdakı kimidir:

Üzv funksiya	Qiyməti
isA()	doubleListClass
nameOf()	"DoubleList"
hashValue()	0

8.5.9 HashTable

Collection sinfindən törənən **HashTable** onun daxilindəki obyektlərə daha tez müraciət edilməsini təmin edir. Normal olaraq obyektlərin istər massiv daxilində, istərsə də siyahı strukturu daxilində saxlanılmasından asılı olmayaraq axtardığımız obyektin yerini bilmiriksə, obyekt kolleksiyasında axtarılan obyektə müraciət edəne qədər bütün obyektlərə bir-bir baxmaq lazım gəlir. Bu da kolleksiyanın başlanğıcına



yaxın obyektlərin axtarılmasında vaxt baxımından yaxşı nəticə verməsinə baxmayaraq uzaq obyektlər üçün mənfi nəticə verir. Buna görə də istənilən bir elementə müraciət etmək üçün, bütün elementləri gözdən keçirməmək üçün məlumatlar strukturu təyin edilmişdir. **HashTable** bu məlumatlar strukturundan yalnız biridir. **HashTable** məlumatlar strukturunun işləmə qaydasını **Object** sinfi daxilində **hashValue** funksiyasını şərh edərkən göstərmişdik.

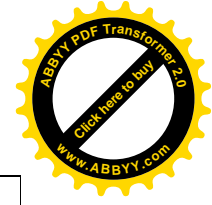
HashTable kolleksiyaları daxilindəki obyektlər **hashValue** funksiyalarının verdikləri qiymətlərə görə qruplaşdırılır. Axtarış apararkən də yalnız axtarılan obyekt **hashValue** funksiyası ilə müəyyən edilən qrup daxilində axtarılacaqdır.

HashTable kolleksiyasının neçə qrupdan ibarət olacağı proqramçı tərəfindən bu kolleksiya tərtib edilərkən müəyyən edilir. Buna görə də **HashTable** sinfinin layihələndiricisi

```
HashTable(sizeType grup_sayi =  
DEFAULT_HASH_TABLE_SIZE);
```

şəklində təyin edilmişdir. Əgər

```
HashTable Memurlar(50);
```



kimi bir **Memurlar** kolleksiyası tərtib edilərsə, bu kolleksiya 50 qrupdan ibarət bir **HashTable** olacaqdır. Yox əgər

```
HashTable Memurlar;
```

kimi təyin edilərsə,

```
DEFAULT_HASH_TABLE_SIZE
```

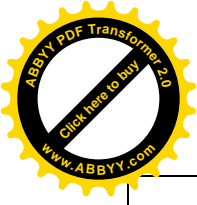
aktiv qiymətini istifadə edəcəkdir ki, bu qiymət də 111-dir. Yəni 111 müxtəlif qrupu olan **Memurlar** adında **HashTable** kolleksiyası yaradılacaqdır.

Layihələndirici xaricindəki funksiyalar **Collection** daxilində təyin edilən funksiyalardır və eyni mənanı daşıyırlar.

Üzv funksiya	Qiyməti
isA()	hashTableClass
nameOf()	"HashTable"
hashValue()	0

8.5.10 Btree

Btree, **Binary-Tree** (ikilik ağac) üsulunun yaddaş sinifləri ilə tətbiq edilməsidir. İkilik ağac üsulunda məlumatlar sanki bir ağac üzərində kökdən budaqlara doğru yerləşdirilmiş kimi saxlanılır. Bu üsulda yaddaş



daxilində yerləşdiriləcək məlumatların öz aralarında sıralanması lazımdır. Məlumatlar bir yarpaq kimidir və budaqlar üzərində yerləşirlər. Budaqlarda yarpaq və ya qiymət budaqları yerləşir. Bu budaqlar üzərində sıralamaya görə ardıcıl (bir-birini təqib edən) məlumatlar yerləşir. Budaqlar da bir-birinə görə ardıcıl məlumat qruplarından ibarət budaqlar şəklində ağac üzərində yerləşirlər. Beləliklə, məlumatı qısa müddətdə gözdən keçirməklə sürətli axtarmaq olar. Məlumat yaddaşda olmasa da, bütün məlumatlar gözdən keçirilmədən bu məlumatın yaddaşda olub olmamasını müəyyənləşdirmək olar.

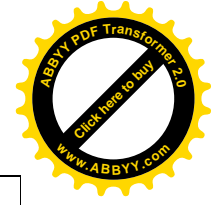
Btree sinfi **Collection** sinfindən törənmiş sinifdir.

Üzv funksiya	Qiyməti
isA()	btreeClass
nameOf()	"Btree"

Btree sinfinin təyin edilmiş bir layihələndiricisi vardır.

Btree(int *Yarpaq_Sayi* = 3);

Bu layihələndiriciyə parametr kimi normal olaraq bir budaq üzərində olması lazım gələn **Yarpaq_Sayi** daxil edilir. Bu qiymət verilmədikdə **3** olur. Budaq üzərindəki yarpaqların sayının az olması axtarma əməliyyatını sürətləndirir. Lakin ağac üzərindəki budaqların sayını



artırır. Buna görə də məlumatlar artdıqca axtarma müddəti də artır. Yarpaqların sayı çox olduğu zaman isə məlumatların sayının az olmasına baxmayaraq axtarma müddəti uzana bilər. Buna görə də yarpaqların sayını məlumat çoxluğunu nəzərə alaraq müəyyənləşdirmək faydalıdır.

int order();

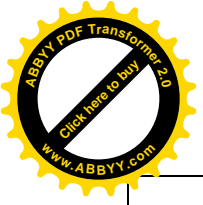
Əvvəlcədən yaradılmış bir ağacın budaqları üzərində ola biləcək yarpaqların sayının müəyyənləşdirilməsi üçün istifadə olunur. Geri qaytarma qiyməti olaraq budaq üzərində ola biləcək yarpaq sayını qaytarır.

virtual void add(Object& Obyekt);

Bu funksiya parametr kimi verilən **Obyekt** obyektinin ağac üzərində yerləşdirilməsini təmin edir. Burada ağac üzərindəki məlumatların sıralana bilən obyekt olduqlarını nəzərə alaraq **Obyekt** obyektinin **Sortable** sinfindən törənmiş sinfin obyektı olmasına diqqət etmək lazımdır.

Object& operator[](long Index) const;

Ağac daxilində yerləşən obyektlər ağac üzərində kiçikdən böyüyə doğru sıralanmış olurlar. İlk sıra



nömrəsi 0 olamaqla istənilən sıradakı obyektin öyrənilməsi üçün [] operatorundan istifadə etmək olar. Bunun üçün Btree sinfinin obyektlərinin hər birini massiv olaraq düşünmək və istifadə etmək mümkündür.

```
Btree Stock;  
...  
cout<<"en kicik element = "<<Stock[0]<<"dir\n";
```

kimisi. Yalnız [] operatoru vasitəsilə hər hansı bir sıraya obyekt mənimsənilə bilməz. Bu operator sadəcə istənilən sıradakı elementin öyrənilməsini təmin edir.

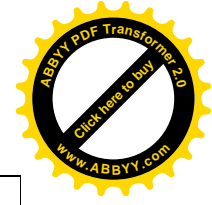
```
long rank(const Object& Obyekt) const;
```

Btree ağacı daxilində olduğu məlum olan obyektin ağacın hansı sırasında olduğunu öyrənmək üçün rank funksiyasından istifadə etmək olar.

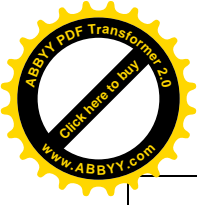
Collection, Container və Object sinifləri üçün təyin edilmiş digər funksiyaları Btree daxilində eyni məqsədlə istifadə etmək olar.

```
//BTREE.CPP
```

```
#include <sortable.h>  
#define integerClass 300  
  
class Integer : public Sortable  
{ int _integer;
```



```
public:  
    Integer(int = 0);  
  
    char* nameOf() const;  
    classType isA() const;  
    hashValueType hashValue() const;  
    int isEqual(const Object&) const;  
    int isLessThan(const Object&) const;  
    void printOn(ostream&) const;  
};  
  
Integer::Integer(int x)  
{ _integer = x; }  
  
char* Integer::nameOf() const  
{ return "Integer"; }  
  
classType Integer::isA() const  
{ return integerClass; }  
  
hashValueType Integer::hashValue() const  
{ return _integer; }  
  
int Integer::isEqual(const Object& object) const  
{ return _integer == ((Integer&)object)._integer; }  
  
int Integer::isLessThan(const Object& object) const  
{ return _integer<((Integer&)object)._integer; }  
  
void Integer::printOn(ostream& stream) const  
{ stream<<_integer; }  
  
//*****  
  
#include <conio.h>  
#include <btree.h>  
#include <stdlib.h>  
#include <stdio.h>  
#include <time.h>
```



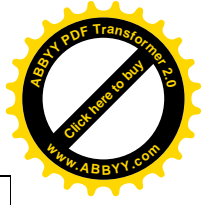
```
Btree Stock(15);

main()
{ clrscr();
  int i;
  char buffer[120];

  for(i = 0; i < 2500; i++)
    Stock.add(* new Integer(random(3000)));

  cout<<"\nStokta axtardiginiz nomreleri daxil edin\n"
    "bitirmek ucun SON yazin\n";
  while(!cin.rdstate())
  { int X;
    cin>>X;
    if(cin.rdstate())
      break;
    Integer Find(X);
    cout<<"Axtarisa baslama saati : "<<Time()<<endl;
    cout<<Find<<" Stokda"<<(Stock.hasMember(Find) ?
      " Movcuddur" : " Mevcud deyil")<<endl;
    cout<<"Axtarisin bitmesi saati : "<<Time()<<endl;
  }
  cin.clear();
  cin>>ws>>buffer;

  cout<<"Rank qiymetini oyrenmek ucun "
    "nomreleri daxil edin\nbitirmek ucun SON yazin\n";
  while(!cin.rdstate())
  { int X;
    cin>>X;
    if(cin.rdstate())
      break;
    Integer Find(X);
    cout<<"Rank("<<Find<<") = "<<Stock.rank(Find)<<endl;
  }
  cin.clear();
  cin>>ws>>buffer;
```

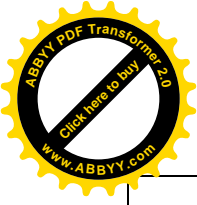


```
cout<<"Oyrenmek istediyniz melumatın"
  " sıra nomresini daxil edin\n"
  "bitirmek ucun SON yazın\n";
while(!cin.rdstate())
{ int X;
  cin>>X;
  if(cin.rdstate())
    break;
  cout<<'# '<<X<<"-->"<<Stock[X]<<endl;
}
cin.clear();
cin>>ws>>buffer;

return 0;
}
```

Program çıxışı

```
Stokta axtardiginiz nomreleri daxil edin
bitirmek ucun SON yazin
50 200 700 3000 122 son
Axtarisa baslama saati : 8:57:25.10 pm
50 Stokda Movcuddur
Axtarisin bitmesi saati : 8:57:25.10 pm
Axtarisa baslama saati : 8:57:25.10 pm
200 Stokda Mevcud deyil
Axtarisin bitmesi saati : 8:57:25.10 pm
Axtarisa baslama saati : 8:57:25.10 pm
700 Stokda Mevcud deyil
Axtarisin bitmesi saati : 8:57:25.10 pm
Axtarisa baslama saati : 8:57:25.10 pm
3000 Stokda Mevcud deyil
Axtarisin bitmesi saati : 8:57:25.10 pm
Axtarisa baslama saati : 8:57:25.10 pm
122 Stokda Mevcuddur
Axtarisin bitmesi saati : 8:57:25.10 pm
Rank qiymetini oyrenmek ucun nomreleri daxil edin
```



bitirmək ucun SON yazın
50 200 3 son
Rank(50) = 44
Rank(200) = 177
Rank(3) = 2
Oyrenmek istediyniz melumatın sira nomresini daxil edin
bitirmək ucun SON yazın
0 1 2 2499 2500 son
#0 --> 1
#1 --> 2
#2 --> 3
#2499 --> 2997
#2500 --> Error

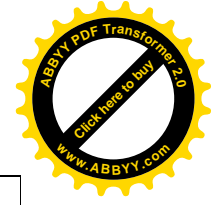
8.5.11 Bag

Müəyyən məqsədlərə görə bir-birinə bərabər olan obyektlərin məlumatlar strukturu daxilində olmasına icazə verilə və ya verilməyə bilər. İndiyə qədər gözdən keçirdiyimiz siniflərin hamısı kimi **Collection** sinfindən törənmiş **Bag** sinfi də buna icazə verir. **Set** və **Dictionary** sinifləri isə icazə vermir. **Bag** sinfinin **HashTable** sinfinə oxşar layihələndiricisi vardır. Eyni məntiqə işləyir. Sadəcə aktiv qiyməti fərqlənir.

Bag(sizeType grup_sayı = DEFAULT_BAG_SIZE);

DEFAULT_BAG_SIZE parametrinin standart təyin olunmuş qiyməti 29-dur.

Üzv funksiya	Qiyməti
--------------	---------



isA()	bagClass
nameOf()	"Bag"

8.5.12 Set

Set sinfi **Bag** sinfindən törənmiş bir sinif kimi ondan fərqli olaraq bir obyektin **Collection** daxilində yalnız bir dəfə olmasına icazə verir. Ekvivalenti kolleksiya daxilində olan obyekt kolleksiyaya qəbul etmir. **Bag** ilə eyni layihələndiriciyə malikdir. Yalnız aktiv qiyməti fərqlidir.

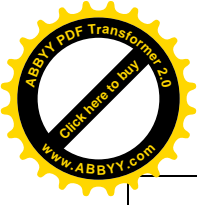
Set(sizeType grup_sayı = DEFAULT_SET_SIZE);

DEFAULT_SET_SIZE parametrinin standart təyin olunmuş qiyməti 29-dur.

Üzv funksiya	Qiyməti
isA()	setClass
nameOf()	"Set"

8.5.13 Dictionary

Dictionary (Lüğət) axtarışa görə məlumat kolleksiyası daxilində bir obyekt tapıb bu obyektə daha çox məlumat əldə etmə üsuludur. Məsələn, bir şəxsin adını göstərərək o şəxsin soyad, doğum yeri,



doğum tarixi kimi digər məlumatlarının da öyrənilməsidir. Bunun ən klassik tətbiqi lüğətdə öz əksini tapır. Bir sözə uyğun gələn o sözün verdiyi mənalara və ya digər dillərdəki mənalara tapılıb müəyyənləşdirilir.

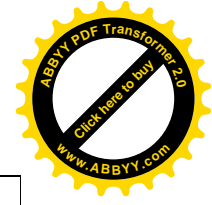
Dictionary Set sinfindən törənən **Association** sinfindən törənmiş obyektlərin daxilində ola biləcək xüsusi bir kolleksiya növüdür.

Dictionary sinfinin üzv funksiyaları **Collection** sinfi daxilində şərh etdiyimiz üzv funksiyalardır. Eyni prototiplərinin olmasına baxmayaraq bu funksiyalardakı *obyekt* parametrlərinə uyğun obyekt olaraq **Association** sinfindən törənmiş obyektlər istifadə edilə bilər.

Üzv funksiya	Qiyməti
isA()	dictionaryClass
nameOf()	"Dictionary"

8.5.14 AbstractArray

AbstractArray xüsusilə də bütün proqramlaşdırma dillərinə daxil olduğuna görə çox istifadə edilən məlumatlar strukturu tipli massivlərdir. **Container Class** kitabxanası daxilində ümumi məqsədli iki sinif təyin edilmişdir. Bunlar ümumi məqsədli massivlər üçün **Array** və sıralanmış obyektlər üçün **SortedArray** sinifləridir.



AbstractArray isə **Collection** sinfindən törənən və bu iki sinfə baza yaradan mücərrəd bir sinfidir.

Əvvəlcə massiv anlayışını aydınlaşdıraraq. Massiv eyni tipli elementlərin kolleksiyasıdır. Elementlər massiv daxilində ardıcıl yerləşir və hər bir elementin sıra nömrəsi olur ki, buna da elementin indeksi deyilir. Hər hansı bir elementə bu indekslə müraciət edilir. İndekslər **1** və ya **0**-dan başlayır. Üst sərhəd isə proqramçı tərəfindən təyin olunur və bu sərhəddi keçmək olmaz.

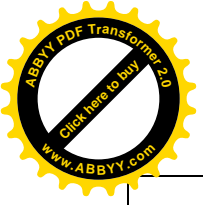
Container Class kitabxanası daxilində təyin olunmuş massiv sinifləri üçün də oxşar qaydalar qüvvədədir.

Aşağıdakı fərqlər vardır:

- Massivlər daxilində **Object** sinfindən törənmiş bütün obyektlər ola bilər;
- Alt və üst sərhədlər proqramçı tərəfindən təyin edilə bilər;
- Massivin elementlərinin sayı kifayət etməzsə, o genişləndirilə bilər;
- Qiyməti verilməyən elementlər **NOOBJECT** kimi qəbul edilir.

Massivlər üçün **AbstractArray** sinfi daxilində təyin edilən funksiyalar aşağıdakılardır:

```
int lowerBound() const;
```



Bu funksiya massivin alt sərhəddini geri qaytarma qiyməti kimi qaytarır. Bu, massiv daxilində olan ilk qiymətin sıra nömrəsidir.

```
int upperBound() const;
```

Massivin üst sərhəddini geri qaytarma qiyməti kimi qaytarır. Bu, massiv daxilində olan son qiymətin sıra nömrəsidir.

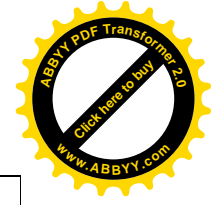
```
sizeType arraySize() const;
```

Bu funksiya massiv daxilindəki elementlərin sayını müəyyənləşdirir. `getItemInContainer` funksiyasından bu xüsusiyyəti ilə fərqlənir. `getItemInContainer` kolleksiya daxilində olan elementlərin sayını müəyyənləşdirir.

```
virtual void detach(int indeks, int tip = 0);
```

Bu funksiya *indeks* ilə yeri verilən obyektı massiv daxilində olduğu zaman massivdən xaric edir. *tip* parametrinin qiyməti 0-dan fərqlidirsə, obyektı massivdən xaric etməklə bərabər onu yaddaşdan da silir.

```
void destroy(int indeks);
```



Bu funksiya isə *indeks* ilə yeri verilən obyektı massiv daxilində olduğu zaman massivdən xaric edir və sonra bu obyektı yaddaşdan silir.

```
virtual int isEqual(const Object& Obyekt) const;
```

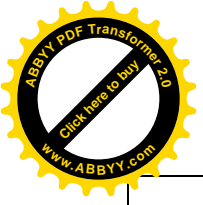
Massivlər üçün yenidən təyin edilən bu funksiya digər siniflərdə olduğu kimi müqayisə edilən hər iki massivin eyni sırada və bir-birinə bərabər olan obyektlərinin olması şərtini qoyur. Bundan başqa massivlərin alt və üst sərhədləri bərabər olamlıdır.

```
virtual void PrintOn(ostream& Stream) const;
```

Massiv daxilindəki obyektlərin axına yazılması tələb olunarsa, `printOn` funksiyası və ya "`<<`" operatoru istifadə edilir. Bunların istifadə olunması nəticəsində massiv daxilindəki obyektlər axına yazılır, qiyməti verilməmiş elementlər isə "error" mesajı ilə əks olunur (boş elementlərə `Error` sinfinin obyektı olan `NOOBJECT` mənimsədilir). Əgər buna ehtiyac olmazsa, yəni yalnız həqiqi obyektlərin göstərilməsi tələb olunarsa,

```
virtual printContainerOn(ostream& Obyekt) const;
```

üzv funksiyasından istifadə etmək olar.



8.5.15 Array

Array sinfi ümumi məqsədlər üçün istifadə olunan bir sinifdir. **AbstractArray** sinfindən törənmişdir. **Array** sinfinin obyektı olan massivlərə **Object** sinfindən törənmiş bütün siniflərin obyektlərini yerləşdirmək mümkündür. Yerləşdirilən hər element yerləşdiyi yerdə qalır.

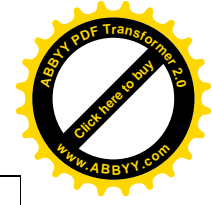
Bu sinfin layihələndiricisi aşağıdakı kimi təyin edilmişdir:

```
Array(int Ust, int Alt = 0, sizeType Delta = 0);
```

Burada ilk parametr olan **Ust** massivin ən böyük indeksini göstərir. Bu qiymət mütləq verilməlidir. İkinci parametr olan **Alt** isə massivin ən kiçik indeksini göstərir. Bu qiymət verilməzsə, avtomatik olaraq **0** qəbul edilir. Üçüncü parametr olan **Delta** isə massivə **add** üzv funksiyası ilə element yerləşdirildiyi zaman boş yer olmazsa, massivin üst sərhəddinin nə qədər genişləndiriləcəyini göstərir. Bu qiymət də verilməzsə, **0** olduğu, yəni belə bir halda hər hansı bir genişlənmənin olmayacağını göstərir.

Qeyd edildiyi kimi massivə obyekt əlavə etmək üçün

```
virtual void add(Object& Obyekt);
```



üzv funksiyası istifadə edilir. Bu funksiya **Obyekt** obyektini massiv daxilindəki ilk boş yerə yerləşdirir. Boş yer olmazsa, massiv layihələndirmə zamanı müəyyən edilən miqdarda genişləndirilir. Genişlənmə olmazsa, səhv baş verir. Massivə əlavə edilən obyektlərin massiv daxilində hansı sırada yerləşəcəyi yenə bu funksiya ilə müəyyən edilir. Əgər yerləşəcəyi yerin də göstərilməsi tələb olunarsa,

```
void addAt(Object& Obyekt, int Indeks);
```

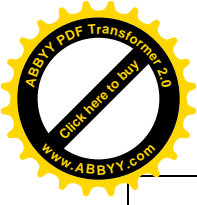
üzv funksiyasından istifadə etmək lazımdır. Bu halda **Obyekt Indeks** ilə verilən yerə yerləşdirilir. Yerləşdirmə əməliyyatından əvvəl yerləşdiriləcək yerdə başqa bir obyekt olarsa, bu obyekt massivdən çıxarılır və əgər silinə bilərsə, silinir.

Massiv daxilində bir obyekt **findMember** üzv funksiyası ilə axtarmaq olar. Əgər yeri məlumdursa,

```
Object& operator[ ](int Indeks) const;
```

operator funksiyası istifadə edilə bilər. Məsələn, bir massivin **5**-ci elementinə müraciət etmək üçün

```
Object& Obyekt = mas[5];
```



və ya ekrana çıxarmaq üçün

```
cout<<mas[5]<<endl;
```

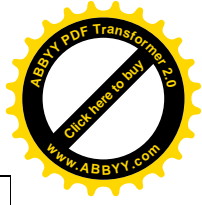
sətrini yazmaq olar.

Digər funksiyaların geri qaytarma qiymətləri isə aşağıdakılardır:

Üzv funksiya	Qiyməti
isA()	arrayClass
nameOf()	"Array"

8.5.16 SortedArray

SortedArray əsasən massivdir. Lakin **Array** sinfindən fərqli olaraq elementlərini kiçikdən böyüyə doğru sıralanmış şəkildə saxlayır. Buna görə də ancaq **Sortable** sinfindən törənən siniflərin obyektlərini tərkibində saxlayır. **SortedArray** massivləri daxilində olan obyektlər sıralanmış şəkildə massivin əvvəlində yerləşirlər. Əlavə edilən hər obyekt sıralamadakı uyğun yerdə yerləşdirilir. Bu əməliyyat massivin daxilindəki digər obyektlər sürüşdürülərək yerinə yetirilir. Bir obyekt massivdən çıxarıldıqdan sonra əmələ gələn boşluğu doldurmaq üçün obyektlər bu dəfə də əks istiqamətdə sürüşdürülür.



Bu sinfin layihələndiricisi də **Array** sinfinin layihələndiricisi kimi təyin edilmişdir.

```
SortedArray(int Ust, int Alt = 0, sizeType Delta = 0);
```

Parametrlər **Array** sinfinin layihələndiricisinin parametrləri ilə eyni funksiyalara malikdirlər. **Ust** massivin ən böyük, **Alt** isə ən kiçik indeksi, **Delta** isə kifayət qədər yer olmadığı zaman genişlənmə miqdarını göstərir.

Digər funksiyaların geri qaytarma qiymətləri aşağıdakılardır:

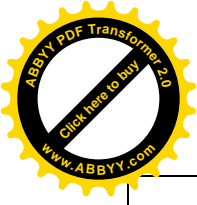
Üzv funksiya	Qiyməti
isA()	sortedArrayClass
nameOf()	"Array"

Bu sinifdə yeri bəlli olan bir obyekt üçün

```
const Sortable& operator[] (int Indeks) const;
```

operator funksiyasından istifadə etmək olar.

Bir diskdəki aktiv qovluqda olan faylların genişlənmələrinin siyahısının görünməsinə aid bir proqram tərtib edək.



//EXT.CPP

```
#include <clsdefs.h>
#include <sortable.h>
#include <sortarry.h>
#include <set.h>
#include <dir.h>
#include <dos.h>
#include <string.h>
#include <iomanip.h>
#include <conio.h>

class NameExt : public Sortable
{ char ext[4];

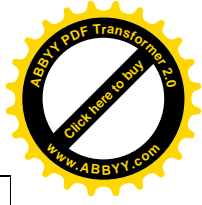
public:
    NameExt();
    NameExt(const NameExt&);
    NameExt(char *n);
    classType isA() const;
    char* nameOf() const;
    hashValueType hashValue() const;
    int isEqual(const Object&) const;
    int isLessThan(const Object&) const;
    void printOn(ostream&) const;
};

NameExt::NameExt()
{ ext[0] = '\0'; }

NameExt::NameExt(const NameExt& O)
{ strcpy(ext,O.ext); }

NameExt::NameExt(char *n)
{ strcpy(ext, n); }

classType NameExt::isA() const
{ return __firstUserClass; }
```



```
char* NameExt::nameOf() const
{ return "File Name Extension"; }

hashValueType NameExt::hashValue() const
{ return (ext[0] == '\0' || ext[0] == ' ') ? 0 : ext[0] - 64; }

int NameExt::isEqual(const Object& Test) const
{ return strcmp(ext, ((NameExt&)Test).ext) == 0; }

int NameExt::isLessThan(const Object& Test) const
{ return strcmp(ext, ((NameExt&)Test).ext) < 0; }

void NameExt::printOn(ostream& Stream) const
{ Stream<<" "<<setw(3)<<setiosflags(ios::left)<<ext<<" "; }

//*****

class Extension : public Set
{ int sta;

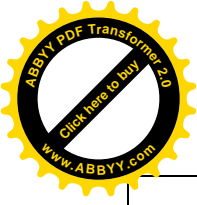
public:
    Extension(int);

    void printHeader(ostream&) const;
    void printSeparator(ostream& ) const;
    void printTrailer(ostream&) const;
};

Extension::Extension(int _sta) : Set(26)
{ sta = _sta; }

void Extension::printHeader(ostream& Stream) const
{ Stream<<"Genislenme siyahisi"<<endl<<endl; }

void Extension::printSeparator(ostream& Stream) const
{ if(sta)
    Stream<<"\t";
  else Stream<<"\n";
}
```

```
void Extension::printTrailer(ostream& Stream) const
{ Stream<<endl; }

//*****

Extension ExtList(1);

void scan(char *s)
{ struct ffbk SF;

  int Ok;

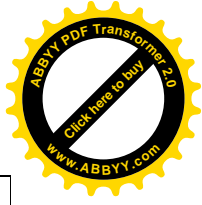
  Ok = findfirst(s, &SF, FA_ARCH);
  while(Ok == 0)
  { char *p = strchr(SF.ff_name, '.');
    NameExt* N = new NameExt((p == NULL) ? " " : p + 1);
    ExtList.add(*N);
    Ok = findnext(&SF);
  }
}

//*****

main(int c, char *a[])
{ clrscr();
  char match[MAXDIR];

  if(c > 1)
  { strcpy(match, a[1]);
    if(strchr(match, '.') == NULL)
      strcat(match, ".");
  }
  else strcpy(match, "*.");

  scan(match);
  if(ExtList.isEmpty())
    cout<<"Verilen genişlenmeli fayl yoxdur.";
  else cout<<ExtList<<"\nCemi "
    <<ExtList.getItemsInContainer()
```



```
<<" eded genişlenme var";
return 0;
}
```

Proqram çıxışı

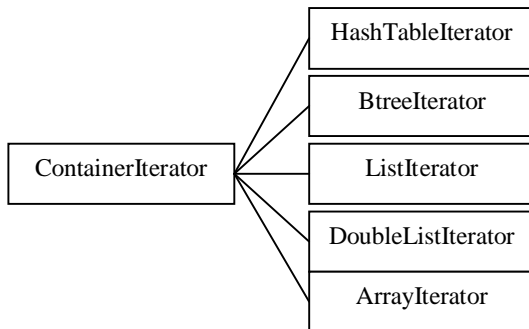
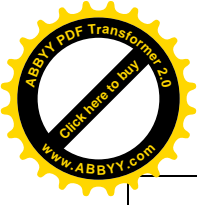
Genişlenme siyahısı

"BAT"	"BAK"	"CSM"	"386"	"CFG"	"COM"	"CPP"	"DAT"
	"DFM"	"DSK"					
"DPR"	"DOS"	"DIF"	"DLL"	"ERR"	"EXE"	"FIL"	"FON"
"HLP"							"H"
"ILD"	"ILC"	"ILF"	"ILS"	"INI"	"ICO"	"MAK"	"OVL"
	"OBJ"	"PIF"					
"SYM"	"SWP"	"SYS"	"TDS"	"TXT"	"TC"	"TAH"	"TCH"
	"TFH"	"TDH"					

Cemi 40 eded genişlenme var

8.6 Yeniləyicilər (Iterators)

Yeniləyicilər **Container** sinfindən törənmiş siniflərin obyektləri üzərində göstərici kimi işləyərək, bu obyektlərə hər hansı bir zərər vermədən, yaddaş xüsusiyyətinə malik olan obyektlər daxilindəki obyektlərə bir-bir müraciət edilməsini təmin edirlər. Yeniləyici siniflərinin hər biri **ContainerIterator** sinfindən törənmişdir.



Müxtəlif yaddaş tipləri üçün təyin olunmuş yeniləyicilər ümumi xüsusiyyətlərə malikdirlər.

Yalnız **DoubleListIterator** sinfinin əlavə bir xüsusiyyəti vardır.

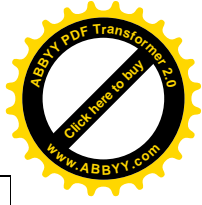
Yeniləyicinin layihələndirilməsi üçün hər yeniləyici bir layihələndiriciyə sahib olmaqla bərabər, yeniləmə əməliyyatının yerinə yetiriləcəyi yaddaş obyektinin

```
virtual ContainerIterator& initIterator() const;
```

kimi təyin edilmiş üzv funksiyasından istifadə etmək məqsədəuyğundur. Məsələn, **S Stack** strukturu üzərində yeniləmə əməliyyatını aparacaq **I** yeniləyicisini

```
ContainerIterator& I = S.initIterator();
```

şəklində təyin etmək olar. Lakin burada qeyd etmək lazımdır ki, **initIterator()** üzv funksiyası **new** operatoru ilə



yaradıldığı üçün yeniləyici ilə işimizi tamamladıqdan sonra **delete** operatoru ilə onu silmək lazımdır.

```
delete &I;
```

Yeniləyicilər üçün təyin edilmiş ortaq funksiyalar aşağıdakılardır:

```
virtual operator int();
```

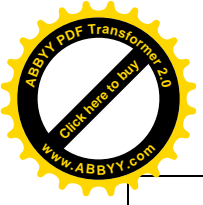
Yeniləyicini **int** tipli bir ədədə çevirmək lazım gələrsə, onun yaddaş sinfi üzərində yeniləyəcəyi başqa obyektin qalıb qalmayacağı sualı ortaya çıxır. Nəticə sıfırıdırsa, yenilənəcək obyekt qalmır, sıfırdan fərqlidirsə, yenilənəcək digər obyektlərin də olduğu bəlli olur.

```
virtual Object& operator++();  
virtual Object& operator++(int);
```

Bu operator isə yeniləyicinin göstərdiyi obyektə geri qaytarma qiyməti kimi qaytardıqdan sonra onun yaddaş daxilində növbəti obyektə göstərməsini təmin edir.

```
virtual operator Object&();
```

Bu operator isə yeniləyicinin göstərdiyi obyektin öyrənilməsini təmin edir. Başqa bir əməliyyatı yerinə yetirməz.



```
virtual void restart();
```

Bu funksiya yeniləyicinin yaddaş daxilində ilk obyektı göstərməsini təmin edir. Beləliklə, yaddaş üçün yeni bir obyekt təyin edilmədən yaddaş daxilindəki obyektlər yenidən gözdən keçirilə bilər.

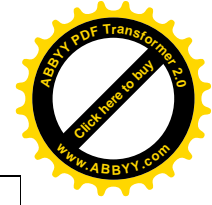
8.6.1 DoubleListIterator

`DoubleListIterator` `DoubleList` sinfi üzərində yeniləmə əməliyyatını yerinə yetirmək üçün layihələndirilmişdir. `DoubleList` sinfi məlumatları bir-biri ilə həm irəliyə, həm də geriye doğru əlaqələndirdiyi üçün `DoubleListIterator` yeniləyicisi ilə uyğun istiqamətlərdə getmək mümkündür.

Bunun üçün ++ operatoruna oxşar olaraq bu sinif üçün -- operatoru da təyin edilmişdir.

```
virtual Object& operator--();  
virtual Object& operator--(int);
```

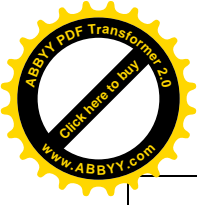
Bu operatorlar yeniləyicinin göstərdiyi obyektı geri qaytararaq, onun əvvəlki obyektı göstərməsini təmin edirlər.



8.7 Misal

Ekran üzərində müxtəlif böcəklər vardır. Bu böcəklərin davranışları müxtəlifdir. Məsələn, bəziləri ekran kənarlarına toxunduqları zaman geri qayıdaraq yollarına davam edirlər. Bəziləri isə bir kənardan çıxıb digər kənardan girərək yollarına davam edirlər. Bir böcək isə klaviatura düymələrinin sıxılması ilə verilən əmrlərlə (ox düymələrinə sıxaraq) hərəkət etdirilir. Oyunun məqsədi də bu böcəkləri ekrandan təmizləməkdir.

```
//CCDEMO.CPP  
  
#include <queue.h>  
#include <conio.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <dos.h>  
  
#define SCREENWIDTH      79  
#define SCREENHEIGHT     24  
#define WAITTIME         150  
  
#define KB_HOME           71  
#define KB_TOP            72  
#define KB_PGUP           73  
#define KB_LEFT           75  
#define KB_RIGHT          77  
#define KB_END            79  
#define KB_BOTTOM         80  
#define KB_PGDN           81  
  
class Bocek;  
//Bocek adli sinfin daha sonra teyin edileceyini bildirir
```



```
Queue Bocekler;
//Boceklerin siyahisi

Bocek* Canavar;
//Bocek sinfinin yalnız gostericisi istifade oluna biler

//***** Bocek Sinfi *****//

class Bocek : public Object
{ protected:
    int type;           //Bocegin tipi
    int x, y;           //Bocegin yeri
    int u, v;           //Bocegin sur'etleri

public:
    Bocek();
    Bocek(int, int, int, int, int);
    virtual ~Bocek();

    virtual char* nameOf() const
    { return "Bocek"; }

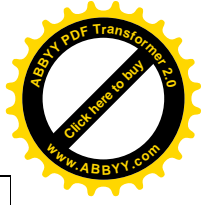
    virtual classType isA() const
    { return 12000; }

    virtual hashValueType hashValue() const
    { return type; }

    virtual void printOn(ostream&) const;
    virtual int isEqual(const Object&) const;

    virtual void toxunma(Bocek&);
    //Parametr ile verilen bocege toxunması
    //neticesinde cagrilacaq uzv funksiya

    virtual void solSerhed();
    virtual void sagSerhed();
```



```
virtual void ustSerhed();
virtual void altSerhed();
//Ekran serhedlerine toxunduğu zaman
//cagirilacaq uzv funksiyalar

virtual void sicra();
//Hereket vaxti geldiği zaman
//hereketi temin eden uzv funksiya

void oldun();
//Bir boceyin basqa birine toxunması zamanı
//olmesi halinda cagirilacaq uzv funksiya

int olumu();
//Bir boceyin olu olub olmadigini anlamaq
//ucun istifade edilecek uzv funksiya

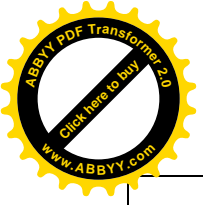
protected:
    virtual void nezaret();
    //Bocegin ekran serhedlerine catib catmadigina
    //ve diger boceklere toxunub toxunmadigina
    //nezaret edecek funksiya

    virtual void sil();
    //Bocegi ekrandan silecek funksiya

    virtual void cek();
    //Bocegi ekranda cekecek funksiya

private:
    int oldx, oldy;
    //Bocegin evvelki yeri
};

Bocek::Bocek()
{ type = 12000;
  oldx = oldy = x = y = u = v = 1;
}
```



```
Bocek::Bocek(int _t, int _x, int _y, int _u, int _v)
{ type = _t;
  oldx = x = _x;
  oldy = y = _y;
  u = _u;
  v = _v;
}

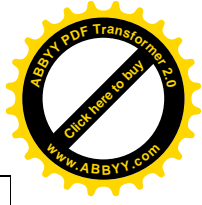
Bocek::~Bocek()
{ }

void Bocek::printOn(ostream& Stream) const
{ Stream<<nameOf()<<"Tip : "<<type;
  Stream<<" Yer : "<<x<<" , "<<y;
  Stream<<" Addim : "<<u<<" , "<<v;
}

int Bocek::isEqual(const Object& Test) const
{ return type == ((Bocek&)Test).type &&
  x == ((Bocek&)Test).x &&
  y == ((Bocek&)Test).y &&
  u == ((Bocek&)Test).u &&
  v == ((Bocek&)Test).v;
}

void Bocek::toxunma(Bocek& B)
{ if(B == *Canavar)
  oldun();
else
{ u = -u;
  v = -v;
  B.u = -B.u;
  B.v = -B.v;
  if(type == B.type && Bocekler.getItemsInContainer() < 20)
    Bocekler.put(*new Bocek(type, x * 2, y * 2, -u, -v));
}
}

void Bocek::solSerhed()
```



```
{ x = SCREENWIDTH; }

void Bocek::sagSerhed()
{ x = 1; }

void Bocek::ustSerhed()
{ y = SCREENHEIGHT; }

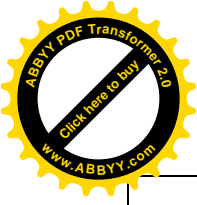
void Bocek::altSerhed()
{ y = 1; }

void Bocek::sicra()
{ sil();
  x+= u;
  y+= v;
  nezaret();
  cek();
}

void Bocek::nezaret()
{ if(x <= 1)
  solSerhed();
else if(x >= SCREENWIDTH)
  sagSerhed();
if(y <= 1)
  ustSerhed();
else if(y >= SCREENHEIGHT)
  altSerhed();

  ContainerIterator& Iter = Bocekler.initIterator();
  while(int(Iter))
  { Bocek* Item = (Bocek*) & (Iter++);
    if(Item->x == x && Item->y == y)
      toxunma(*Item);
  }
  delete &Iter;
}

void Bocek::sil()
```



```
{ gotoxy(oldx, oldy);
  putchar(' ');
}

void Bocek::cek()
{ gotoxy(x, y);
  putchar(type);
  oldx = x;
  oldy = y;
}

void Bocek::oldun()
{ u = v = 0;
  sil();
}

int Bocek::olumu()
{ return u == 0 && v == 0; }

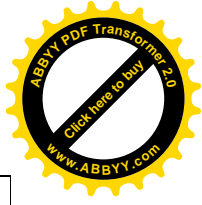
//***** KenarBocek Sinfı *****//

//Bir kenardan cixib diger kenardan giren bocek tipi //

class KenarBocek : public Bocek
{ public:
  KenarBocek() : Bocek()
  { }
  KenarBocek(int a, int b, int c, int d, int e)
  : Bocek(a, b, c, d, e)
  { }

  virtual void toxunma(Bocek&);
  virtual void solSerhed();
  virtual void sagSerhed();
  virtual void ustSerhed();
  virtual void altSerhed();
};

void KenarBocek::carpdin(Bocek& B)
```



```
{ if(B == *Canavar)
  oldun();
else if(Bocekler.getItemsInContainer() < 20)
  Bocekler.put(*new KenarBocek(type, x + 1, y, -u, -v));
}

void KenarBocek::solSerhed()
{ u = 1; }

void KenarBocek::sagSerhed()
{ u = -1; }

void KenarBocek::ustSerhed()
{ v = 1; }

void KenarBocek::altSerhed()
{ v = -1; }

//***** NezaretBocek Sinfı *****//

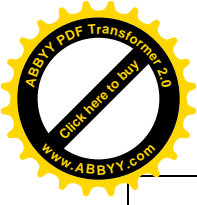
//Oyuncunun ox duymeleri ile hereket etdirdiyi bocek //

class NezaretBocek : public Bocek
{ public:
  NezaretBocek() : Bocek()
  { u = v = 1; }

  NezaretBocek(int a, int b, int c)
  : Bocek(a, b, c, 1, 1)
  { }

  virtual void toxunma(Bocek&);
  virtual void solSerhed();
  virtual void sagSerhed();
  virtual void ustSerhed();
  virtual void altSerhed();

  virtual void sicra();
};
```



```
void NezaretBocek:: toxunma(Bocek& B)
{ if(B == *Canavar)
  return;
  B.oldun();
}

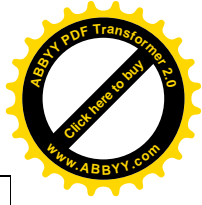
void NezaretBocek::solSerhed()
{ x = 1; }

void NezaretBocek::sagSerhed()
{ x = SCREENWIDTH; }

void NezaretBocek::ustSerhed()
{ y = 1; }

void NezaretBocek::altSerhed()
{ y = SCREENHEIGHT; }

void NezaretBocek::sicra()
{ if(kbhit())
  { int ch = getch();
    if(ch == 27 || ch == 3)
      exit(1);
    if(ch == 0)
    { ch = getch();
      switch(ch)
      { case KB_HOME : y--;
        case KB_LEFT : x--; break;
        case KB_PGDN : y++;
        case KB_RIGHT : x++; break;
        case KB_PGUP : x++;
        case KB_TOP : y--; break;
        case KB_END : x--;
        case KB_BOTTOM : y++; break;
      }
      sil();
      nezaret();
    }
  }
```



```
}
cek();
}

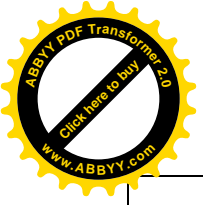
//***** Ana Program *****/

//Proqrami icra eden, ardıcıl olaraq bütün boceklere mesaj
//gonderen alt proqram

void Run()
{ Bocek* Item;
  int i;
  while(*(Item = (Bocek*)&Bocekler.get()) != NOOBJECT)
  { if(Item->olumu())
    { delete Item;
      continue;
    }
    Item->sicra();
    Bocekler.put(*Item);
    for(i = 0; i < WAITTIME; i++)
      Canavar->sicra();
    if(Bocekler.getItemsInContainer() <= 1)
    { clrscr();
      gotoxy(20, 12);
      cprintf("\a\aOyun Bitdi");
      gotoxy(20, 14);
      cprintf("Her hansı bir duymeni sixin");
      getch();
      while(kbhit())
        getch();
      break;
    }
  }
}

//Proqramdan cixarken istifade olunan altproqram

void Cix()
{ clrscr();
```



```
_setcursortype(_NORMALCURSOR);
}

//Proqramin baslamasi ucun qurmaq

main()
{ int i;

//Boceklerin yaradilmasi

    for(i = 0; i < 6; i++)
        Bocekler.put(*new Bocek('&', i + 10, 15, -1, 1));

//KenarBoceklerin yaradilmasi

    for(i = 0; i < 6; i++)
        Bocekler.put(*new KenarBocek('*', i + 30, i + 5, 1, 1));

//Oyuncunun nezaret edecegi bocegin yaradilmasi

    Canavar = new NezaretBocek('#', SCREENWIDTH / 2,
                                SCREENHEIGHT / 2);
    Bocekler.put(*Canavar);

//Ekran gorunusunun hazirlanmasi

    clrscr();
    _setcursortype(_NOCURSOR);

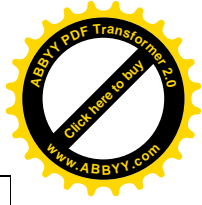
//Cixis funksiyasinin menimsedilmesi

    atexit(Cix);

//Oyunun baslanmasi

    Run();

//Oyundan normal cixis
```



```
    return 0;
}
```