C++ PROGRAMLAMA DİLİ İLE NESNE YÖNELİMLİ PROGRAMLAMA

ELEKTRONİK YÜKSEK MÜHENDİSİ İLHAN ÖZKAN
Ankara, 2025

ÖNSÖZ

Ülkemizde her alanda olduğu gibi yazılım alanında da bir terminoloji karmaşasının yaşandığı aşikardır. Son dönemde yazılanlar adeta çeviri yapan programların bir gecede ürettiği çıktılar birleştirilerek hazırlanmaktadır. Birçok programlama dili kullanarak yazılım geliştiriyor olmama rağmen, yazılım alanında içeriği anlama konusunda oldukça zorlanıyorum.

Bu kitapla birlikte tarihsel gelişimle birlikte programlama mantığının yapısal programlamaya geçişi sürecini de içerecek şekilde yazılımım gelişme süreci anlatılmıştır.

Yapısal programlamanın en çok uygulandığı Pascal dilidir. Pascal diline yakın zamanda ortaya çıkan C dili, kısa sürede sistem dili olarak tarihte yerini almıştır. Günümüzde sistem programlama ve yüksek hız gerektiren gömülü sistemlerde vazgeçilmez bir dil olmuştur. Yapısal programlamada, yazılımı yapılacak sürece ilişkin nelerin yapılacağını değil, işin nasıl yapılacağını belirtirler.

Yapısal dillerde çok büyük projelerde değişim yönetiminin zorlaşması, hata ayıklamanın zor olması ve aynı kodların benzer projelerde tekrar tekrar yazılması gibi problemlerden dolayı yazılımcıların imdadına Simula ve Smalltalk programlama dillindeki yaklaşım yetişmiştir. Bu diller oldukça yavaş olmasına rağmen getirdiği çözümler oldukça yenilikçiydi. Smalltalk dilinde nesneler temel yapıtaşlarıdır. Nesneler; durum ve davranışlara sahiptir ve programlama, nesnelerin birbirlerine ileti göndermesiyle yapılır.

C++ programlama dili ise C diline yapılan nesne yönelimli programlama eklentileri yapılarak geliştirilmiş bir programlama dili olup ilk sürümü 1985 yılında yayınlanmıştır. Hala popüler olarak yüksek performans isteyen projelerde çokça kullanılmaktadır.

Nesne yönelimli programlama mantığını oluşturan kavramların programcıların kafasında aynı ışığı yakması için bu kitap hazırlanmıştır. Bu dilin eğitimini verecek tüm kişiler için serbestçe kullanılabilir.

Bu kitabın güncel sürümüne https://github.com/HoydaBre/cplusplus adresinden erişilebilir. Amacım Türkçe düşünen e konuşan öğrencilerimize kavramları doğru bir şekilde aktarmaktır. Katkılarınızı bekler iyi çalışmalar dilerim.

İlhan ÖZKAN

ilhanozkan[at]outlook.com

İÇİNDEKİLER

TEMEL BİLGİSAYAR KAVRAMLARI

Mühendis ve Teknisyen

Mühendis (engineer), ihtiyaç duyulan ürünü; ekonomik (cost), güvenli (safe) ve kullanışlı (functional) olarak zamanında (on time) ortaya koyan kişilerdir. Bunu yapmak için; keşif (invent), tasarım (design), analiz (analysis), uygulama (build) ve sınama (test) yaparlar[1].

Teknisyenler (technician) ise genellikle laboratuvarda teknik ekipmanlarla ilgilenen veya bu ekipmanlar ile pratik çalışmalar yapan kişilerdir. Genel olarak amacı ekipmanın bakım ve kullanımına ilişkindir.

Konumuz yazılım olduğundan, yazılım mühendisleri ise istenilen bilgisayar yazılımını ekonomik, güvenli ve kullanışlı olarak zamanında ortaya koyan kişilerdir. Programcılar ise teknisyenlere karşılık gelir. Yani geliştirme aşamasında bazı yazılım kısımlarının kodlanmasını veya geliştirilen yazılımın çalışır tutulmasını sağlarlar.

Bilgisayara Niçin İhtiyaç Duyulur?

Bir çiftçi niçin traktöre ihtiyaç duyar? Bir berber niçin elektrikli tıraş makinesi kullanır? Benzer şekilde bir mühendis niçin bilgisayara ihtiyaç duyar?

İşte bütün bu soruların cevabı yapılacak işleri daha kısa sürede yapmaktır. Yani bütün araç gereç ve makinalara olan ihtiyacımız hız (speed) ile ilgilidir. Bütün makinelere gibi bilgisayar da yapılacak işleri hızlandırmak için kullanılır. Bilgisayarlar; veriden hızlıca bilgi elde etmemizi ve tekrarlanan hesaplamaları hatasız yapmamızı hızlıca sağlayan makinelerdir.

Veri sorumlusu, veri hazırlama kontrol işletmeni gibi unvanları bu sebeple ortaya çıkmıştır. Bu unvanlara ihtiyaç duyulmasının sebebi verinin belli bir şekle uygun olarak bilgisayara girilmesi ve işlenen verinin saklanması ve raporlanmasının bilgisayar dünyasında önemli bir yer tutmasıdır.

Bilgisayarlar ilk zamanlarda daha çok cebirsel ifadeleri hatasız ve hızlı bir şekilde yapmak için kullanılmışlardır. Bunu daha sonradan ALGOL (ALGOritmic Language) adını alacak IAL (International Algebraic Language) ya da FORTRAN (FORmula TRANslation) dilinden de anlayabiliriz.

Veri ve Bilgi

Veriler (data), genellikle bir ölçüm ya da sayım sonucu elde edilen ve hakkında az şey bilinen varlıklardır. Örneğin ölçülen hava sıcaklığı, derse giren öğrenci sayısı, tartılan sebzenin ağırlığı gibi şeyler.

Bilgi (information) ise verinin işlenerek otaya çıkan anlamlı verilerdir. Yani ortalama hava sıcaklığı, ölçülen yerde bir ölçüm yapmaya gerek kalmadan sıcaklığı yaklaşık olarak bilmeye yarayan bir bilgidir. Bir sınıftaki not ortalaması, o sınıftan rastgele seçilen bir öğrencinin notunu yaklaşık belirleyen bir bilgidir. Kısacası bilgi, işlenmiş veridir.

En Basit Bilgisayar

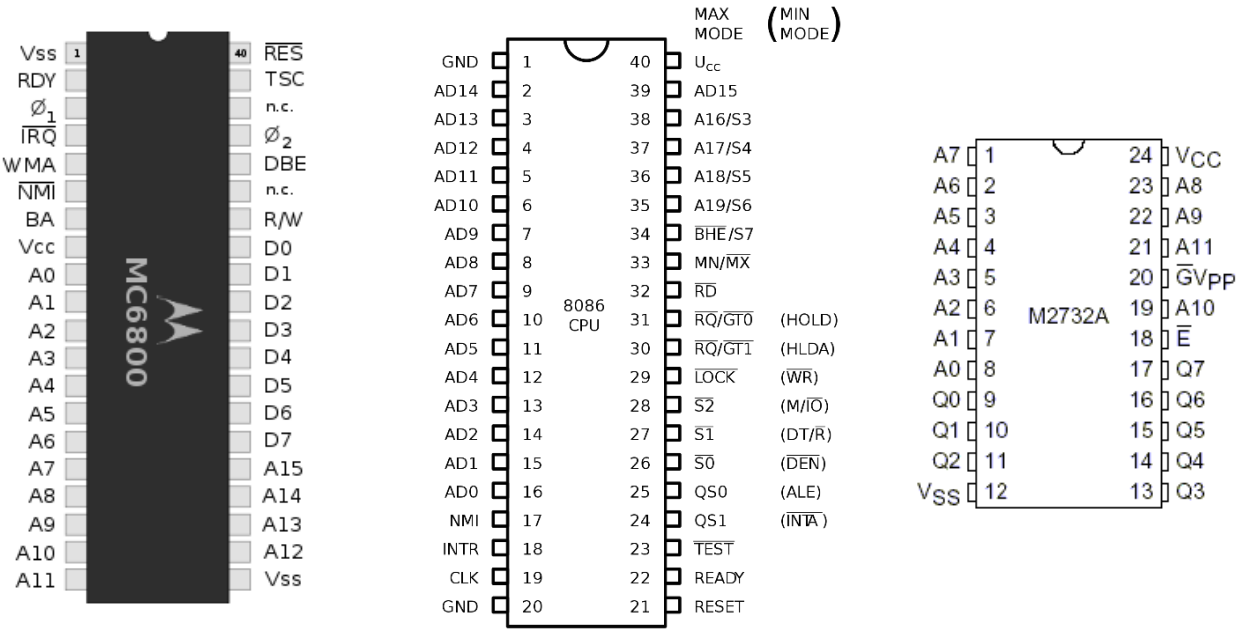
En basit bilgisayar;

- Merkezi işlem birimi (Central Processing Unit-CPU) kısaca işlemci,
- Bellek/Hafıza (memory) ve
- Bunları birbirine bağlayan elektrik yollarından (bus) oluşur.

Bellek işlemcinin işleyeceği emirleri barındırır. Belleğin hem adres (address) hem de veri (data bus) bacakları bulunur. Benzer şekilde işlemcinin de bellek ile iletişimini sağlayan adres ve veri bacakları

[1] https://www.bls.gov/

bulunur. Belleğin bu bacakları elektrik yolları ile işlemcinin adres ve veri bacaklarına bağlıdır. Bunların yanında işlemci ve belleğin okuma (read), yazma (write), kesme (interrupt) veya kristal bağlanan bacakları gibi kontrol (control) bacakları da bulunur.



Şekil 1. Motorola 6802, Intel 8086 İşlemciler ile 2732 EPROM Belleği

Şekil 1'de görüldüğü üzere 6801 işlemcisinin 15 adet adres bacağı, 8 adet veri bacağı bulunmaktadır. 8086 işlemcisinin ise 20 adet adres bacağı bulunakta olup bu bacakların 16 adedi aynı zamanda veri bacağı olarak kullanılmaktadır. 2732 belleğin ise 12 adet adres bacağı, Q ile gösterilen 8 adet veri bacağı bulunaktadır. Bu bacaklar birbirlerine elektrik yolları (bus) ile bağlanır.

İşlemci ile bellek arasında bağlantı sağlayan yollar (bus) adres veya veri taşır. Bu yolların veri taşıyanlarına veri yolu (data bus), adres taşıyanına ise adres yolu (address bus) adı verilir. Veri yolu; hem işlemci tarafından belleğe veri yazmak veya bellekten işlemciye taşınacak veriler için kullanıldığından çift yönlüdür. Adres yolu ise belleğin hangi bölgesindeki veriye ulaşılacağını belirleyen hatlar olup tek yönlüdür.

Bilgisayarlar elektrikle çalışan aletlerdir. İşlemci ile bellek arasında bir hatta elektrik varsa "1" yoksa "0" olarak anlamlandırılır. Bunlar ikili sayı sisteminin rakamlarıdır (BInary DigiT) ve kısaca bit olarak adlandırılır. Dolayısıyla; 1 elektrik hattının taşıyacağı veri 0 ya da 1 olabilir. 2 elektrik hattının taşıyacağı veri 00, 01, 10 ve 11 olabilir. Yani 2^2 kadar farklı veri taşıyabilir. n adet elektrik hattının taşıyacağı veri 2^n farklı alternatifte bilgi olacaktır.

İşlemcinin veri yolunun (data bus) genişliği kelime uzunluğudur ve işlemcinin aynı anda işlem yaptığı bit sayısını da verir. 8 Bitlik veriye bayt (byte) adı verilir ve bellek miktarı genellikle bayt olarak ölçülür. Veri yolu, 8 bit ise 8 bitlik-BYTE işlemci, 16 bit ise 16 bitlik-WORD işlemci, 32 bit ise 32 bitlik-DWORD (Double WORD) işlemci, 64 bit ise 64 bitlik-QWORD (Quad WORD) işlemci olarak adlandırılır. Bu nedenle veri yolu genişliği ile işlemcinin akümülatör ve kaydedicilerin bit sayısı da aynı olur.

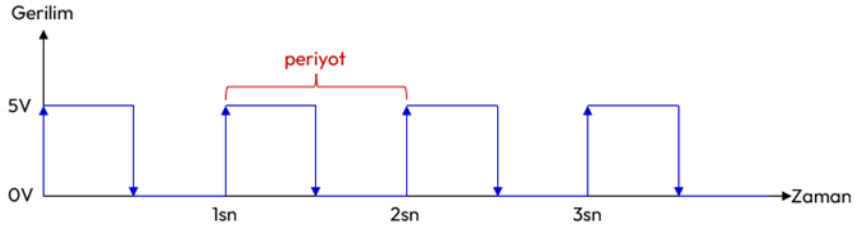
Adreslenebilir bellek miktarı ise işlemcinin adres hatlarının sayısı yani adres yolu (address bus) ile belirlenir. Adres yolu ne kadar geniş ise adreslenebilir bellek miktarı da o kadar artar.

İşlemci	Veri Yolu	Adres Yolu	Adreslenebilir Bellek: bayt	kilobayt	megabayt	gigabayt
Motorola 6802	8 Bit	16 Bit	2^{16}=65.536	64		
Intel 8086	16 Bit	20 Bit	2^{20}=1.048.576	1024	1	
Intel 80286	16 Bit	24 Bit	2^{24}=16.777.216		16	
Intel Pentium	32 Bit	32 Bit	2^{32}=4.294.967.296		4096	4
Intel i7	64 Bit	36 Bit	2^{36}=68.719.476.736			64

Tablo 1. Bazı İşlemcilerin Adres ve Veri Yolu Genişlikleri

Bilgisayarlarda kablodaki gerilimin değeri genelde 5V, 3.3V, 2.8V, 2V, 1.5V veya 1V gibi değerler alır. Kullanıcılar için gerilimin ne olduğu değil varlığı önemlidir. Gerilim değerin yüksek olması fazla akım çekme ve ısınma anlamına gelir. Bu nedenle zamanla daha düşük gerilimle çalışan işlemciler tasarlanmıştır.

İşlemciler kare dalga olarak oluşturulmuş elektriksel bir işareti saat olarak kullanır. Bu işaret işlemciye Ø ya da CLK bacaklarından uygulanır. Saniyede 1 kez bir dalga üreten işaretin frekansı 1/1sn = 1 Hertz= 1 Hz. olarak hesaplanır. Bir mikro saniyede bir kez dalga üreten işaretin frekansı; 1/1µs = $1/10^{-6}$s = 10^{6} Hertz = 1.000.000 Hertz = 1 MHz olarak hesaplanır.



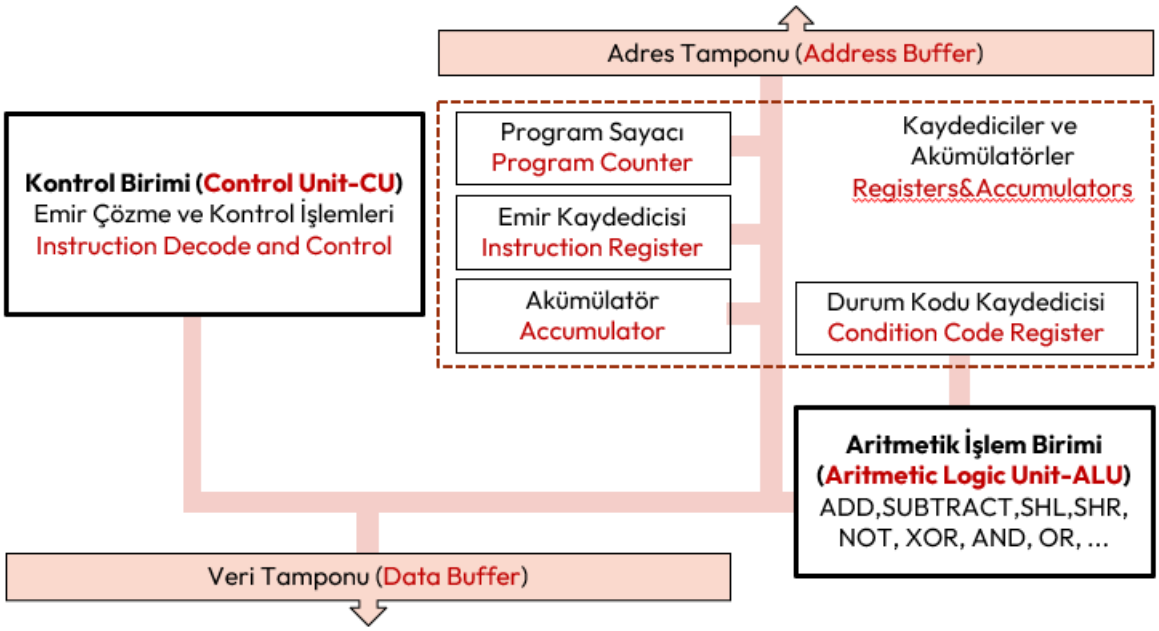
Şekil 2. Frekansı 1 olan Kare Dalga İşareti

İşlemciye uygulanan saatin frekansı ne kadar yüksek olursa işlemci o kadar hızlanır. Ancak işlemcilerin de tasarımından ötürü karşılayabileceği belli bir frekans değeri bulunmaktadır. Günümüzde 4 GHz frekansını karşılayan işlemciler bulunmaktadır.

İşlemcinin Emri Alma, Çözme ve İcra Döngüsü

Basit bilgisayarda;

- Kaydediciler (register), işlemci içerisinde adres ve veri saklayan mini belleklerdir.
- Akümülatörler ise bellekten alınan verileri saklamak için en sık kullanılan kaydedicilerdir.
- Program sayacı (program counter), işlemcinin icra edeceği emri (instruction) takip için kullanılır. Getirilecek bir sonraki emrin bellek adresini içerir.
- Emir Kaydedicisi (instruction register) işlemcinin o an icra ettiği emirdir.
- Koşul kodu kaydedicisi (condition code register), herhangi bir işlemin durumunu gösteren farklı bayraklar (flag) içerir.



Şekil 3. Basit bir İşlemcinin Yapısı

Bu bilgisayar tasarımı aynı zamanda depolanmış program (stored program) kavramı olarak adlandırılır ve *Von Neumann Mimarisi* olarak da bilinir[2]. *John Von Neumann* tarafından 1946-1949 yılları arasında tasarlanan bu mimari günümüzde hala kullanılmakta ve modern bilgisayarlarda hala kullanılmaktadır. Basit bir bilgisayara elektrik verildiğinde aşağıdaki üç adımlık çevrim elektrik kesilene kadar sürekli tekrarlanır;

1. Emri Alma (fetch): Program sayacının (program counter) tuttuğu adresteki emir (instruction) bellekten okunur.
2. Çözme (decode): İşlemci içerisindeki kontrol birimi (control unit) tarafından emrin ne anlama geldiği ve buna bağlı nelerin yapılacağını (örneğin ulaşılacak bellek adresi değiştirilir) belirler ve program sayacı bir sonraki emir için bir artırılır.
3. İcra (execute): Çözülen emir koduna bağlı olarak gerekirse yine işlemci içinde bulunan aritmetik işlem birimini (Aritmetic Logic Unit-ALU) çalıştırılır ve emrin işlenmesi sonucunda ortaya çıkan değerler ise işlemci içinde bulunan akümülatörlere (accumulator) veya kaydedicilerde (register) saklanır.

Burada çözülecek emirlerin (instruction) her biri bir tamsayıya karşılık gelir. Bu saylar emir kodu (instruction code) veya makine kodu (opcode) olarak da bilinirler. İşlemciler tüm sayıları emir kodu olarak anlamaz. Yani sınırlı sayıda sayı emri tanırlar. Yukarıda verilen Motorola 6802 işlemcisi 72 farklı emri, Intel 8086 işlemcisi ise 116 farklı emri tanır. Çözülebilecek tüm emirlerin oluşturduğu kümeye emir seti (instruction set) adı verilir.

İşlemci Tasarlama Stratejileri

İşlemciler, aşağıda verilen üç strateji ile tasarlanırlar;

1. İndirgenmiş Emir Setli İşlemciler (Reduced Instruction Set Computing-RISC): Az sayıda emir alan yüksek hızlı işlemci tasarımı. RISC işlemciler az sayıda emre sahip olduğundan emrin çözülmesi ve icrası da kısa zaman almaktadır. Bu da çalıştırılacak kodu büyütür.
2. Karmaşık Emir Setli İşlemciler (Complex Instruction Set Computing-CISC): Çok fazla emri anlayan karmaşık nispeten yavaş işlemci tasarımı. CISC işlemciler, çok sayıda emir tanıyabildiğinden emrin çözülmesi ve icra edilmesi nispeten daha uzun sürer. Ancak yüksek düzey dillerin daha kolay makine koduna çevrilmesini kolaylaştırır.
3. Özel İşlemciler: Bunlar grafik kartları ve yapay zekâ gibi belli amaçlar için tasarlanmış işlemcilerdir.

Genel olarak işlemcilerde "*fetch-decode-execute*" çevrimi (instruction cycle) 1 saat periyodunda tamamlanmaz. RISC işlemciler için bu çevrim, birçok emir için 1 saat periyodunda tamamlanır. CISC işlemciler için ise birden fazla saat periyodu gerekir. Çünkü CISC işlemcilerin emir seti daha geniş olduğundan emrin çözülmesi daha fazla vakit alır. Bu nedenle RISC işlemciler CISC işlemcilere göreceli olarak daha hızlıdır.

Günümüzde ticari olarak satılan işlemcilerin çoğu ortak emir seti kullanmaya başlamışlardır;

– Masaüstü bilgisayarların birçoğu Intel ve AMD Marka işlemciler kullanır ve bunlar IA-16, IA-32, x86-16, x86-64, AMD64, ... emir setleri ve eklentilerini desteklemektedir.
– Bilinen birçok telefon işlemcisi Acorn Risc Machine-ARM tabanlı işlemciler kullanır. Bu işlemcilerde ARMv7, ARMv8, ... emir setleri ve eklentileri kullanılmaktadır.

Program ve Programlama

Belli bir işlemi gerçekleştirmek üzere, işlemciye verilen bir dizi emre (instruction) bilgisayar programı (computer program) denir[3]. Bu bir dizi emrin, işlemcinin çalıştıracağı şekilde hazırlaması işlemine programlama (programming) denir.

[2] https://www.youtube.com/watch?v=ByIlwN8q2ss
[3] "ISO/IEC 2382:2015". ISO. 2020-09-03. [Software includes] all or part of the programs, procedures, rules, and associated documentation of an information processing system.

Programlama emir kodları (instruction code/opcode) ile yapıldığından, programın dili, makine dili (machine language) olarak adlandırılır. Her işlemci üreticisi, her bir emre farklı makine kodu verdiğinden programlama oldukça zorlu ve teknik bilgi gerektirmekteydi. Burada daha önce yazılmış bir programın ne yaptığını anlamak için her defasında emir listelerine defalarca bakmak gerekiyordu.

Sembolik İsim (Mnemonic)	16 Tabanında Emir Kodu (Hexadecimal Opcode)	Tanım
ABA	1B	ADD A to B → A
INCA	4C	INCREMENT A
INCB	5C	INCREMENT B
SBA	10	SUBTRACT B from A→ A
LDA	96	LOAD M to A, M: Memory
LDB	D6	LOAD M to B, M: Memory

Tablo 2. 6802 Emir Seti Örneği ve Sembolik İsim Listesi

Programın okunurluğunu artırmak için her bir emre sembolik isimler (mnemonic) verilmeye başlandı. Bu sembolik isimlerle yazılan program montaj kodu (assembly code) olarak adlandırılır. Montaj kodları bir metin dosyasına yazılarak bir dosyaya saklanır. Bu dosyalar montaj dosyaları olarak adlandırılır. Montaj dosyalarını makine diline çeviren programlar ilk derleyicilerdir (compiler).

Aşağıda x86 emir seti kullanan Linux işletim sisteminde örnek montaj (assembly) kodu örneği verilmiştir; Program genel amaçlı `CL` kaydedicisine koyulan sayıya kadar `2` ve `3`'e bölünebilen sayıların toplamını bulup yine genel amaçlı `DX` kaydedicisine koyar[4].

```
section  .text          ; programı oluşturan emir kodları buradan başlar
         global   _start
_start:
         mov   dx, 0    ; dx 0 olsun. Toplamı tutatcak.
         mov   cl, 99   ; 99 kez tekrarlanacak.
sum:
         mov   ax, cx   ; cx, ax e kopyalanır. Bölme işlemi için.
         mov   bl, 3    ; bl 3 olsun.
         div   bl       ; ax/bl.
         cmp   ah, 0    ; Kalan olup olmadığı kontrol ediliyor.
         jne   cont     ; Kalan yoksa, 'cont' etiketine git.
         mov   bl, 2    ; Sonra bl 2 olsun.
         div   bl       ; ax/bl
         cmp   ah, 0    ; Kalan olup olmadığı kontrol ediliyor.
         jne   cont     ; Kalan yoksa, 'cont' etiketine git.
         add   dx, c    ; Değilse, cx deki sayı 2 ve 3'>e bölünür. Toplama ekle.
cont:
         loop  sum      ; Bir sonraki sayı için 'sum' etiketine dön.
exit:
         mov   eax, 1   ; Artık programdan çıkıyoruz.
         mov   ebx, 0   ; Return 0.
         int   80h      ; linux çekirdeğini çağır. (soft interrupt 80h!).
```

Genel Amaçlı Bilgisayarlar ve İşletim Sistemleri

Genel amaçlı yani herkesin kendi amacına göre kullanacağı bilgisayarlar; Veri girişini sağlayan klavye, verileri görüntüleneceği monitör, işlenen verilerin elektrik kesildiğinde saklanacağı disk ve benzeri donanımlara sahip olmalıdır. Genel amaçlı bilgisayarın kullanıcı isteklerine cevap verebilmesi için;

1. Çeşitli programlara sahiptirler. İşte bu programlar bütününe işletim sistemi (operating system) adı verilir. DOS, Windows, Unix, Linux, MacOs, ChromeOS, OS/2 bunların en çok bilinenleridir.

[4] https://github.com/armut/x86-Linux-Assembly-Examples

2. Elektrik verildiğinde klavye, monitör ve disk olup olmadığı kontrol eden Basic Input Output System-BIOS programı koşturulur. BIOS gerekli kontrollerden sonra bilgisayara işletim sistemini yükler. İşletim sistemi yüklendikten sonra, işletim sistemi kullanıcının vereceği komutları bekler.
3. Kullanıcı işletim sitemine vereceği konutlarla yeni bir program derleyebilir, derlenmiş bir programı çalıştırabilir.
4. Kullanıcının vereceği komutlar grafik olmayan işletim sistemlerinde (DOS, Unix, Linux gibi) komut yazılarak verilir. Komut yazılan bu ekrana konsol, terminal ya da komut satırı adı verilir. Sunucu işletim sistemlerinde hala tercih edilmektedir.
5. Grafik ara yüzlerine sahip işlettim sistemlerinde (Windows ve MacOs İşletim Sistemleri ile KDE, Mate, GNome, Cinnamon, LXQt, XFce ve Deepin gibi grafik ara yüzleri içeren Linux İşletim Sistemleri) fare tıklamasıyla çoğu komut verilebilmektedir.

İşletim sistemlerinin yaygınlaşması ile kullanıcının bilgisayarı donanım bilmeden yapabilmesi sağlanmıştır. Günümüzdeki işletim sistemlerinin temel programlama dili olarak C ve c++ dili kullanılmaktadır.

TEMEL YAZILIM KAVRAMLARI

Tarihçe

1642 Yılında *Blaise Pascal* tarafından icat edilen Pascalline Hesap Makinesi, eldeli toplama, ödünç almalı çıkarma yapıyordu.

1671 Yılında *Gottfried Wilhelm Leibniz* tarafından icat edilen Leibniz Çarkı, toplama, ödünç almalı çıkarmanın yanında bölme, çarpma ve karekök işlemlerini yapıyordu.

1801 Yılında *Joseph Maria Jacquard* dokuma tezgahlarında desenleri oluşturmak için delikli kartlar (punch card) kullanmayı icat etmiştir. Sonradan bilgisayara veri girdisi sağlamak ve sonuçların çıktısını göstermek için kullanılacaktır.

1830 Yılında *Charles Babbage* fark makinesi olan analitik makineyi icat etmiştir. Buhar gücü kullanan bu makinenin mantıksal işlem birimi, veri depolama birimi, giriş çıkış üniteleri bulunuyordu. *Augusta Ada Byron*, Babbage ile beraber çalışmıştır. Byron, analitik makinenin bir dizi matematiksel işlemi gerçekleştirebildiğini fark etti ve bu makineyi *Bernoulli* sayılarını üretmek için kullandı. Bu sayıların hesaplanması için yazdığı algoritma tarihteki ilk bilgisayar programı olarak kabul edilir. Bu sebeple *Byron*, ilk programcı olarak kabul görür.

1854 Yılında *George Boole* tarafından ikili sayı sistemini geliştirmiş bugün bilgisayarlarımızın kullandığı bool cebiri üzerine çalışmalar yapmıştır.

1890 Yılında *Herman Hollerith* tarafından delikli kartlarla bilgilerin yüklenebildiği ve bu bilgiler üzerinde toplama işlemlerinin yapılabildiği bir elektro mekanik araç geliştirdi. ABD'nin 1890 nüfus sayımında başarılı biçimde kullanıldı. *Hollerith* başarılı olunca bir şirket kurdu ve daha sonra üç firma işle birleşerek 1924 yılında adını IBM olarak değiştirdi.

1941 Yılında *Konrad Zuze* Z3 isimli elektrik motorları ile çalıştırılan mekanik bir bilgisayar yaptı. Bu (Z1, Z2, Z3 ve Z4 serisi) program kontrollü ilk bilgisayardır.

1944 Yılında *Howard Aitken*, IBM ile iş birliği yaparak MARK I'i yaptı. Bilgiler, MARK I'e delikli kartlarla veriliyor ve sonuçlar yine delikli kartlarla alınıyordu. Saniyede 5 işlem yapabiliyordu. Boyutları,18 m uzunluğunda ve 2,5 m yüksekliğinde idi. İnsan müdahalesi olmadan sürekli olarak, hazırlanan programı yürüten ilk bilgisayar idi ve tam olarak elektronik bir bilgisayar değildi.

1943-1945 Yılları arasında *Konrad Zuze*, ilk yüksek düzey programlama dili olan Plankalkül'ü, Z1 bilgisayarı için geliştirmiştir. Emir kodlarını (instruction code) ve sembolik isimleri (mnemonic) içermeyen programlama dilleri yüksek düzey programlama dili (high-level programming language) olarak adlandırılır. Yüksek düzey programlama dilleri çoğunlukla İngilizce sözcükleri kullanır.

1946 Yılında *John Von Neumann* önderliğinde Pensilvanya Üniversitesinde ENIAC (Elektronik Sayısal Hesaplayıcı ve Doğrulayıcı) isimli sayısal elektronik bilgisayar tamamlandı. Askeri amaçla üretildi ve top mermilerinin menzillerini hesaplamak için kullanıldı. *Neumann* tarafından geliştirilmiş bu mimari (Stored-program computer and Universal Turing machine § Stored-program computer), günümüzdeki bilgisayarlarda hala kullanılmaktadır. 18,000 adet elektronik tüp kullanılan ENIAC; 150 kW gücünde idi ve 50 ton ağırlığıyla 167 m^2 yer kaplıyordu. Saniyede 5.000 toplama işlemi yapabiliyordu. Mark-I'den 1000 kat daha hızlıydı.

Bir önceki bölümde anlatıldığı üzere emir kodu veya sembolik isimlerle yazılan programlar düşük düzey programlama (low-level programming) olarak adlandırılır. Bu programlamanın yapılabilmesi için hem işlemcinin hem de hafıza ve işlemciye bağlanan diğer bileşenlerin nasıl çalıştığı ve tasarlandığını çok iyi bilinmesi gerekir. Daha çok elektronik ve bilgisayar mühendisleri ya da sistemi tasarlayan matematikçiler bu programlama türünü kullanır.

Genel Amaçlı Yüksek Düzey Diller

Sembolik isimler yerine bildiğimiz konuşma diline yakın talimatlar olarak metin olarak yazılan programlara ilişkin diller, yüksek düzey programlama dilleri (high-level programming language) olarak adlandırılır. Yani emir kodu (instruction code) ve sembolik isim (mnemonic) kullanmayan programlama dilleri yüksek düzey programlama dileridir.

Yüksek düzey dillerde talimatlar (statements) olarak yazılan programlar yine kendine has derleyiciler tarafından makine diline çevrilir. Her talimat, genellikle birden fazla makine kodu ya da emirden (instruction) oluşur.

1954 Yılında FORTRAN (FORmula TRANslation), *John Backus* liderliğinde IBM 704 için geliştirilirdi. 32 farklı talimata (statement) sahiptir. İlk genel amaçlı, yüksek düzey bir programlama dilidir.

Yüksek seviyeli bir dilde yazılmış bir talimat (statement), bilgisayara belirtilen bir eylemi gerçekleştirmesini söyler. Yüksek seviyeli bir dildeki tek bir talimat, birkaç emri (instruction) içerebilir. Talimatlar, yapılaması gerekeni kısa ve net olarak belirten mantıksal cümlelerdir (logical sequences).

Aşağıda üç kenarı teyp ünitesinden alınan bir üçgenin alanını hesaplayan FORTRAN II programı verilmiştir[5]. Görüldüğü üzere talimatlar kısa öz ve anlaşılırdır.

```
C AREA OF A TRIANGLE - HERON'S FORMULA
C INPUT - CARD READER UNIT 5, INTEGER INPUT
C OUTPUT -
C INTEGER VARIABLES START WITH I,J,K,L,M OR N
      READ(5,501) IA,IB,IC
  501 FORMAT(3I5)
      IF (IA) 701, 777, 701
  701 IF (IB) 702, 777, 702
  702 IF (IC) 703, 777, 703
  777 STOP 1
  703 S = (IA + IB + IC) / 2.0
      AREA = SQRT( S * (S - IA) * (S - IB) * (S - IC) )
      WRITE(6,801) IA,IB,IC,AREA
  801 FORMAT(4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= ,F10.2,
     $13H SQUARE UNITS)
      STOP
      END
```

1956 Yılında ticari amaçlarla kullanılabilen ve seri halde üretimi yapılan ilk bilgisayar UNIVAC I oldu. UNIVAC, giriş-çıkış birimleri manyetik bant idi ve bir yazıcıya sahipti. UNIVAC 1951-1959 arasında üretilen bilgisayarlarda vakum tüpleri kullanıldı. Bu tüpler bir ampul büyüklüğünde, çok fazla enerji harcamakta ve çok fazla ısı yaymakta idiler. Veri ve programlar manyetik teyp ve tambur gibi bilgi saklama araçlarıyla saklandı. Veriler ve programlar bilgisayara delgi kartları ile yükleniyordu. 1959 yılı sonrasında üretilen bilgisayarlarda transistörler (yaklaşık 10 bin adet) kullanıldı. Artık transistor içeren ikinci kuşak bilgisayarlar hayatımıza girmiştir.

Arapsaçı Kod

1955-1959 Yılları arasında FLOW-MATIC:B0 (Business Language Version 0) Dili, *Grace Hopper* tarafından UNIVAC I için geliştirildi. İngilizceye benzeyen ilk yüksek düzey programlama dilidir. Aşağıda B0 Dilinde örnek bir program verilmiştir[6].

```
 (0) INPUT INVENTORY FILE-A PRICE FILE-B ; OUTPUT PRICED-INV FILE-C UNPRICED-INV
     FILE-D ; HSP D .
 (1)  COMPARE PRODUCT-NO (A) WITH PRODUCT-NO (B) ; IF GREATER GO TO OPERATION 10 ;
     IF EQUAL GO TO OPERATION 5 ; OTHERWISE GO TO OPERATION 2 .
```

[5] https://github.com/scivision/fortran-II-examples/blob/main/herons_formula.f

[6] https://gist.github.com/marcgg/f9f2da31a973ba319809

```
 (2)  TRANSFER A TO D .
 (3)  WRITE-ITEM D .
 (4)  JUMP TO OPERATION 8 .
 (5)  TRANSFER A TO C .
 (6)  MOVE UNIT-PRICE (B) TO UNIT-PRICE (C) .
 (7)  WRITE-ITEM C .
 (8)  READ-ITEM A ; IF END OF DATA GO TO OPERATION 14 .
 (9)  JUMP TO OPERATION 1 .
(10)  READ-ITEM B ; IF END OF DATA GO TO OPERATION 12 .
(11)  JUMP TO OPERATION 1 .
(12)  SET OPERATION 9 TO GO TO OPERATION 2 .
(13)  JUMP TO OPERATION 2 .
(14)  TEST PRODUCT-NO (B) AGAINST ZZZZZZZZZZZZ ; IF EQUAL GO TO OPERATION 16 ;
     OTHERWISE GO TO OPERATION 15 .
(15)  REWIND B .
(16)  CLOSE-OUT FILES C ; D .
(17)  STOP . (END)
```

Görüldüğü üzere bu tarihlere kadar yazılan programlarda bir sürü GO TO ya da JUMP TO talimatı (statement) bulunmaktadır. Bu durumda programın ne yaptığı konusunda fikir yürütmeye çalıştığımızda okunaklılık oldukça azdır. İşte okunaklılığın az olduğu bu program kodlarına arapsaçı kod (spaghetti code) adı verilir. Günümüzde ne yaptığı anlaşılamayan kodlar için bu kavram çokça kullanılmaktadır.

1958 Yılında LISP programlama dili, *John McCarthy* önderliğinde *Steve Russell* tarafından geliştirilmiş ikinci yüksek düzey programlama dilidir.

1959 Yılında Common Business-Oriented Language-COBOL programlama dili, *CODASYL* komitesi tarafından İngilizceye benzer ikinci yüksek düzey bir dil olarak geliştirilmiştir.

1958 Yılında daha sonra ALGOritmic Language-ALGOL adını alan International Algebraic Language-IAL Programlama Dili geliştirilmiştir. Bu dil daha sonra ortaya çıkan birçok dile önderlik etmiştir. Bu dil ile birlikte kod blokları (code block), iç içe fonksiyon (nested function), değişken kapsamı (scope) ve dizi (array) kavramları programcının hayatına girmiştir. ALGOL dili, barındırdığı özellikler sebebiyle, kendisinden sonra ortaya çıkan bütün programlama dilleri için bir esin kaynağı olmuştur.

İşin doğası gereği makine diline çevrilecek tüm metinler gözle okunabilmektedir. Yani bütün kaynak kodlar gözle okunabilir.

Değişken

Değişkenler (variable) aslında cebirden bildiğimiz değişkenlerdir. Cebir (algebra) işlemlerinde doğrudan problemde verilen sayıları değil, onları temsil eden değişkenleri kullanırız.

$$3x^2 + 2x + 10$$

$$c^2 = a^2 + b^2$$

$$c = 2\pi r$$

Yukarıdaki cebirsel ifadelerin ilkinde bir polinom, ikincisinde ise bir dik üçgenin kenar uzunlukları arasındaki ilişki verilmiştir. Bu ifadelerde `x`, `a`, `b` ve `r` bağımsız değişken, `c` ise bağımlı değişkendir. Aslında problemi çözerken bunlar sayılara karşılık gelir. Örneklerdeki `3`,`2`,`10`,`2` ve `π` ise değişmez (literal) sayılardır.

Bilgisayarda ise sayıları tutacak değişkenler, bir miktar bellek bölgesini işgal eder ve bu bellek bölgesinde çeşitli biçimlerde tutulurlar. Tamsayılar, ikili (binary) sayı olarak bellekte tutulur. Bu ikili sayının en anlamlı bitinin işaret biti olarak kullanılıp kullanılmayacağına göre bellekte biçimlendirilir;

Bit Sayısı	Bellek Miktarı	Sayı Sınırları
8 Bit	bayt	-127 ile +128 arasında veya 0 ile 255 arasında

16 bit	2 bayt	-32.767 ile +32.768 arasında veya 0 ile 65.535 arasında
32 bit	4 bayt	-2.147.483.648 ile +2.147.483.647 veya 0 ile 4.294.967.295 arasında
64 bit	8 bayt	-9.223.372.036.854.775.808 ile +9.223.372.036.854.775.807 arasında veya 0 ile 18,446,744,073,709,551,615 arsında

Tablo 3. Tamsayı Değişkenler ve Sayı Sınırları

Gerçek sayılar kayan noktalı sayı (floating-point number) olarak biçimlendirilir; 1914 yılında *Leonardo Torres Quevedo* tarafından Charles Babage'ın analitik makinesine dayalı kayan nokta analizi yayınladı.

$$12345 = 1234{,}5x10^1 = 123{,}45x10^2 = 12{,}345x10^3 = 1{,}2345x10^4 = 0{,}12345x10^5$$

Yukarıda ondalık tabanda verilen gerçek bir sayıya ilişkin kesir noktasının kaydırılması ile aynı sayı ifade edildiği bir örnek verilmiştir. Kesir noktası sola kaydırıldığında 10 ile çarpılmış, sağa kaydırıldığında ise 10 ile bölünmüş olur. Kayan noktalı sayının üs kısmı dışında kalan kısmı taban (base) veya kesir (fraction) olarak adlandırılır. Tabanın kesir noktası sağında kalan hanelerine ise mantis (mantissa) adı verilir.

1938 yılında *Konrad Zuse*, ilk ikili (binary) programlanabilir mekanik bilgisayar olan Z1'i yapmıştı. Bu bilgisayar, 7 bitlik işaretli üs, 17 bitlik anlamlı değer ve bir işaret biti içeren 24 bitlik ikili tabanda kayan noktalı sayı kullanmıştır.

Bilgisayarlarda tamsayılarda olduğu gibi kayan noktalı sayılar da IEEE-754 standardında ikili sayı tabanında tutulur[7]. Örneğin ondalık tabanda 50,25 sayısı ikili tabanda 1.1001001 x2^5 olarak ifade edilir.

Bit Sayısı	Kapladığı Bellek Miktarı	Sayı Sınırları
32 bit	4 bayt	$1{,}2x10^{-38}$ ... $3{,}4x10^{+38}$ Tek hassasiyetli gerçek sayılar; en soldaki 1 bit işaret biti, sonraki 8 bit üs ve geri kalan 23 bit ise kesir olarak kullanılan ikilik tabanda saylardır.
64 bit	8 bayt	$2{,}3x10^{-308}$ ila $1{,}7x10^{+308}$ Çift hassasiyetli gerçek sayılar; en soldaki 1 bit işaret biti, sonraki 11 bit üs ve geri kalan 52 bit ise kesir olarak kullanılan ikilik tabanda saylardır.

Tablo 4. Kayan Noktalı Sayılar ve Sınırları

Program yazılırken tanımlanacak değişkenlere ilişkin bellek ihtiyacı programcı tarafından belirlenir. Programcı değişkene tahsis edilecek bellek miktarına ve tutacağı içeriğe göre veri tipi (data type) belirler. Programcı tarafından kullanılacağı amaca göre belirleyeceği veri tipinde istediği kadar değişken tanımlayabilir.

Fonksiyon

Matematikte fonksiyon (function), bir veya daha fazla kümenin elemanını tek bir kümenin elemanına eşleyen ilişkidir (relationship). Girdi olan bağımsız değişkenler ile çıktı olan bağımlı değişken arasındaki bu ilişki, matematiksel olarak aşağıdaki şekilde ifade (expression) edilir.

Girdi	İlişki	Çıktı
1	x2	2
2	x2	4
3	x2	6
4	x2	8
6	x2	12
...		

Tablo 5. İki ile çarpma fonksiyonu.

$$f(x) = 2x$$

[7] IEEE Computer Society (2019-07-22). IEEE Standard for Floating-Point Arithmetic. IEEE STD 754-2019. IEEE. pp. 1–84.

Burada f, fonksiyon anlamında x ise girdi anlamında olup parametre olarak adlandırılır. Böylece tabloda verilen fonksiyon tablosuna her girdiyi yazmak zorunda kalmayız. Bazen fonksiyona aşağıdaki örneği verilen şekilde bir ad verilmez;

$$y = 2x$$

Bazı durumlarda ise girdi daha detaylı tanımlanır;

$$\mathbf{x \in R} \text{ olmak üzere } \mathbf{f(x)=2x+3}$$

Bu fonksiyon, reel sayı kümesindeki her `x` elemanını, hesaplanan `f(x)` değerine eşleyen bir ifadedir. Yani `x=1.0` argümanını `f(1.0)=5.0` reel sayısına eşler. Kısaca fonksiyon `5.0` reel sayısını hesaplayıp döndürür (return).

$$\mathbf{a \in R} \text{ ve } \boldsymbol{b \in Z} \text{ olmak üzere } \mathbf{g(a,}\boldsymbol{b}\mathbf{)=2a+b}$$

Bu fonksiyon ise reel sayı kümesindeki her `a` elemanı ile tamsayı kümesindeki `b` elemanını, hesaplanan `g(a,b)` değerine eşleyen bir ifadedir. Yani `a=1.0` ve `b=1` argümanlarını `g(1.0,1)` fonksiyonu yine reel sayı kümesinden `3.0` reel sayısına eşler. Kısaca fonksiyon, `3.0` reel sayısını hesaplayıp döndürür (return). Buradaki `x`, `a` ve `b` parametre olarak adlandırılır. `f(1.0)` ifadesindeki `1.0` ise parametrenin o an andığı değeri belirtir ve gerçek parametre (actual parameter) ya da argüman (argument) olarak adlandırılır. Bu fonksiyonlar, `f(3.0)+g(3.0,2)+f(2.0)+g(1.0,3)` gibi tanımlandıktan sonra tekrar tekrar çağrılabilirler (call).

Yapısal Programlama

1958 Yılındaki FORTRAN-II diline çıktı üretmeyen fonksiyonlar için SUBROUTINE, değer üreten fonksiyonlar için FUNCTION, CALL ve RETURN özellikleri eklenmiştir.

1958 ile 1969 Yılları arasında ALGOL diline çıktı üretmeyen fonksiyonlar için PROCEDURE ve BEGIN-END arasında kod bloğu özellikleri eklenmiştir.

1966 Yılındaki FORTRAN-IV diline INTEGER, REAL, DOUBLE PRECISION, COMPLEX ve LOGICAL veri tipleri (data type) ile Mantıksal IF, aritmetik (üç yollu) IF ve DO döngüsü talimatları (statement) eklenmiştir.

Bu yıllarda her ne kadar çıktı üretmeyen fonksiyonlar için SUBROUTINE, birden fazla çağrı noktası olabilen COROUTINE ve FUNCTION kullanılıyor olmasına rağmen hala GO TO/JUMP TO ifadeleri kullanılmakta ve kafa karışıklığına devam etmekteydi. 1968 Yılında *Edsger W. Dijkstra* önderliğinde GO TO/JUMP TO ifadesini zararlı olarak ilan edilmiştir.

1970 Yılında *Niklaus Wirth* tarafından Pascal dili geliştirildi. Bu dil ilk geliştirilen yapısal programlama dilidir. Yapısal programlamada (structural programming) verileri taşıyan veri yapıları (değişkenler ve diziler) ile bunu işleyen kontrol yapıları (if, for, do, repeat) birbirinden ayrılmıştır. Pascal dili, kendi kendini çağıran özyinelemeli (recursive) fonksiyonlar ile değişkenlerin adreslerini tutan göstericileri (pointer) desteklemektedir[8].

Yapısal programlamanın genel çerçevesi;

1. Programın başladığı yeri belirten bir ana fonksiyon (main function) tanımlanır. Kodlaması yapılacak işi birbirini çağıran fonksiyonlar yazarak şeklinde burada yazarız. Yani programlama genel olarak fonksiyonların birbirini çağırmasıyla yapılır.
2. Her fonksiyonda ilk önce veri yapıları (data structure) tanımlanır. Veri yapıları;
   a. En basit veri yapısı `char`, `integer`, `float` ve `double` gibi ilkel veri tiplerinden (primitive data type) olabilen değişken (variable) olabileceği gibi,
   b. İlkel veri tiplerinden meydana gelen dizi (array),
   c. İlkel veri tiplerinin birkaçından veya dizilerinden bir araya gelmesiyle oluşan yapılar (structure),

[8] Wirth, Niklaus (1976). Algorithms + Data Structures = Programs. Prentice-Hall. p. 126

   d. Ya da dinamik olarak yani çalıştırma anında birbirine bağlanmış verilerden oluşan bağlı liste (linked list) olabilir.
3. Her fonksiyonda, tanımlanan veri yapılarının ardından, bu yapıları işleyecek kontrol yapıları (control structure) tanımlanır. Kontrol yapıları bir veya daha fazla veya iç içe `if`, `if-else`, `switch`, `do`, `while`, `for`, `continue`, `break`, `goto` ve `return` talimatlarından (statement) oluşur.

Pascal dilinde ana fonksiyon nokta ile biten BEGIN-END bloğudur. C ve C++ dillerinde ana fonksiyon, `main()` fonksiyonudur.

C Programlama Dili, *Dennis M. Ritchie* tarafından Bell Laboratuvarlarında UNIX işletim sistemini geliştirmek için geliştirilen genel amaçlı, üst düzey bir dildir. C, ilk olarak 1972'de DEC PDP-11 bilgisayarında çalıştırılmıştır. *Ken Thompson* tarafından geliştirilen B adlı daha eski bir dilden gelişmiştir.

1978 yılında *Brian Kernighan* ve *Dennis Ritchie*, günümüzde K&R standardı olarak bilinen C'nin ilk kamuya açık kılavuzunu hazırlamışlardır. C programlama dili;

– Öğrenmesi kolaydır,
– Yapısal (structural) programlama dilidir,
– Hızlı ve verimli (efficient) programlar üretir,
– Assembly dili desteği ile düşük düzey programlamayı (low-level programming) da destekler,
– Standart başlık dosyaları sayesinde çeşitli bilgisayar platformlarında derlenebilir.

Bu özellikleri dolayısıyla neredeyse tüm işletim sistemlerinin çekirdeği C dilinde yazılmıştır ve sistem dili olarak da adlandırılmaktadır.

Yapısal programlamada veri ile bunu işleyen yapılar birbirinden ayrıldığından arapsaçı programlamanın aksine okunaklılık oldukça yüksektir.

Nesne Yönelimli Programlama İhtiyacı

C++ programlama dili, C diline yapılan nesne yönelimli programlama (object oriented programming-OOP) eklentileri yapılarak geliştirilmiş bir programlama dilidir. 1979 senesinde bir Danimarkalı bilgisayar bilim adamı olan *Bjarne Stroustrup*, sonradan C++ olarak bilinecek olan "C with Classes" üzerinde çalışmaya başladı ve 1985 yılında ilk sürümünü yayınladı.

Yapısal programlamada talimatlar (statements) art arda koda yazılarak programlama yapılır ve programların neler yaptığı bu talimatlar izlenerek anlaşılabilir. Talimatların zincirin halkaları gibi birbirinin peşi sıra yazılarak yapılan programlamaya emreden programlama (imperative programming) paradigması adı verilir. Bu paradigmaya sahip diller, yazılımı yapılacak sürece ilişkin nelerin yapılacağını değil, işin nasıl yapılacağını belirtirler. Emreden paradigmanın bir örneği olan yapısal programlamada;

– Kod ne kadar büyürse, modüllere ayırma imkânı olmasına rağmen, fonksiyonlar arasındaki bağımlılık da o kadar artar. Bir fonksiyonun parametrelerindeki değişiklik, onun kullanıldığı tüm yerlerde değişiklik gerektirir. Değişim yönetimi (change management) çok zordur ve uzun zaman alır.
– Hata durumunda yapılacak işlemlere ilişkin kod ile iş sürecini gerçekleştiren kod iç içedir. Aynı fonksiyonun bazı durumlarda hata, bazı durumlarda ise değer döndürmesi mümkündür. Çoğu durumda hataların izini sürmek (error handling) işi zorlaşır.
– Yapısal programlamada, gösterici (pointer) kullanımında erişilmesi istenmeyen bellek bölgelerine erişilmesi halinde istenmeyen program davranışları ortaya çıkar. Kodun güvenli (safe code) olarak çalıştığının incelenmesi çok zordur.
– Sürekli olarak benzer projelerde aynı kodları tekrar yazmak (duplicate code) gereklidir.
– Emreden paradigmanın burada belirtilen sebepler başta olmak üzere çeşitli problemleri bulunmaktadır. Bu nedenle 1980'li yıllarda birçok yazılım projesi başarısız (fail) olmuştur. Bir yazılım; Kullanıcı ihtiyaçlarının karşılanamaması, Öngörülen bütçenin aşılması ve Zamanında teslim edilememesi durumlarında başarısız olur;

Yapısal programlamadaki bu sıkıntıları gören yazılımcıların imdadına Simula ve Smalltalk programlama dillindeki yaklaşım yetişmiştir. Bu diller oldukça yavaş olmasına rağmen getirdiği çözümler oldukça yenilikçiydi.

Smalltalk, 1972 yılında Xerox Park şirketinde *Alan Kay* önderliğinde *Dan Ingalls*, *Adele Goldberg*, *Ted Kaehler*, *Diana Merry*, *Scott Wallace* ve *Peter Deutsch* tarafından geliştirilmiş bir dildir. Smalltalk dilinde nesneler (object) temel yapıtaşlarıdır. Nesneler;

- Durum (state) ve davranışlara (behavior) sahiptir.
- Programlama, nesnelerin birbirlerine ileti göndermesiyle (message-passing) yapılır.
- Nesnelerin kendi ya da bir başka nesnenin davranış ve durumlarını öğrenebilme yeteneği yani yansıma (reflection) özelliği vardır.

Emreden paradigma (imprative programming) aksine nesnelerin birbirlerine ileti göndermesi bakış açısıyla yapılan programlamaya nesne yönelimli programlama (object oriented programming) paradigması adı verilir.

DERLEYİCİLER VE ÇALIŞTIRMA ORTAMI

Genel amaçlı bilgisayarların çok kullanılmasıyla birlikte metin editörleri de (Windows Not Defteri, Mac TextEdit, Notepad++, Epsilon, EMACS, nano, vim veya vi) işletim sistemlerinin bir parçası olmuştur. Metin editörlerinde yazılan programlar ise derleyiciler (compiler) tarafından makine koduna çevrilerek icra edilebilir (executable) dosya olarak kaydedilebilmektedir.

C++ dilinde programlama öğrenmeye başlamak için ilk adım, programı yazabileceğiniz bir metin editörü kullanmak ve yazdığınız programa ilişkin kaynak kodları c uzantısı ile dosya olarak saklamaktır. İkinci adım ise işletim sisteminizde (operating system) çalışabilen bir icra edilebilir dosya oluşturan bir derleyici kurmaktır. Yani bilgisayarınızda iki yazılım aracına ihtiyacınız vardır; Birincisi C++ derleyicisi, ikincisi ise basit metin düzenleyicisidir.

Derleyici için girdi, kaynak kodu dosyasıdır. Kaynak dosyasında yazılan kaynak kodu, içerisinde C++ talimatları (statement) olan, insan tarafından okunabilen metin dosyasıdır. Kaynak kodda verilen talimatlara göre işlemcinizin programı çalıştırabilmesi için, bunun makine diline derlenmesi (compile) gerekir. Derleme, bir başka programlama dilinden sembolik isim (mnemonic) ya da makine koduna (instruction code) dönüştürme işlemidir. Bunu yapan programlara derleyici (compiler) adı verilir.

C++ Derleyicileri

Günümüzde birçok C derleyicisi mevcuttur. En çok kullanılanları aşağıda verilmiştir;

- GNU Derleyici Koleksiyonu (GCC) – GCC, popüler bir açık kaynaklı C derleyicisidir[9]. Windows, macOS ve Linux dahil olmak üzere çok çeşitli platformlarda kullanılabilir. GCC, çok çeşitli özellikleri ve çeşitli C standartlarına desteğiyle bilinir.
- Clang – Clang, LLVM projesinin bir parçası olan açık kaynaklı bir C derleyicisidir[10]. Windows, macOS ve Linux dahil olmak üzere çok çeşitli platformlarda kullanılabilir. Clang, hızı ve optimizasyon yetenekleriyle bilinir.
- Microsoft Visual C++ – Microsoft Visual C++, Microsoft tarafından geliştirilen tescilli bir C derleyicisidir[11]. Windows, macOS ve Linux dahil olmak üzere çok çeşitli platformlarda kullanılabilir. Visual C, Microsoft Visual Studio geliştirme ortamıyla entegrasyonuyla bilinir.

Bilgisayarınızın 64 bitlik Windows olduğu varsayılarak G++ derleyicisinin MINGW sürümü ilgili web sayfasından indirilerek kurulur[12]. Kurulumu tamamlandıktan sonra PATH değişkenine G++ derleyicisinin klasörü eklenir. Böylece kurulum tamamlanmış olur.

Komut satırı (DOS Prompt) açıldığında `g++ -v` komutu yazılarak derleyicinin düzgün kurulup kurulmadığı kontrol edilebilir.

```
C:\Users\ILHANOZKAN>g++ -v
Using built-in specs.
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=D:/msys64/ucrt64/bin/../lib/gcc/x86_64-w64-mingw32/13.2.0/lto-wrapper.exe
Target: x86_64-w64-mingw32
Configured with: ../gcc-13.2.0/configure --prefix=/ucrt64 --with-local-prefix=/ucrt64/local --
build=x86_64-w64-mingw32 --host=x86_64-w64-mingw32 --target=x86_64-w64-mingw32 --with-native-
system-header-dir=/ucrt64/include --libexecdir=/ucrt64/lib --enable-bootstrap --enable-
checking=release --with-arch=nocona --with-tune=generic --enable-
languages=c,lto,c++,fortran,ada,objc,obj-c++,jit --enable-shared --enable-static --enable-
libatomic --enable-threads=posix --enable-graphite --enable-fully-dynamic-string --enable-
libstdcxx-filesystem-ts --enable-libstdcxx-time --disable-libstdcxx-pch --enable-lto --enable-
libgomp --disable-libssp --disable-multilib --disable-rpath --disable-win32-registry --
```

[9] https://gcc.gnu.org/
[10] https://clang.llvm.org/
[11] https://visualstudio.microsoft.com/tr/vs/features/cplusplus/
[12] https://www.mingw-w64.org/downloads/

```
disable-nls --disable-werror --disable-symvers --with-libiconv --with-system-zlib --with-
gmp=/ucrt64 --with-mpfr=/ucrt64 --with-mpc=/ucrt64 --with-isl=/ucrt64 --with-pkgversion='Rev3,
Built by MSYS2 project' --with-bugurl=https://github.com/msys2/MINGW-packages/issues --with-
gnu-as --with-gnu-ld --disable-libstdcxx-debug --with-boot-ldflags=-static-libstdc++ --with-
stage1-ldflags=-static-libstdc++
Thread model: posix
Supported LTO compression algorithms: zlib zstd
gcc version 13.2.0 (Rev3, Built by MSYS2 project)
```

Daha sonra metin editörü ile ilk kaynak kodumuzu oluşturmak için komut satırında `notepad hello.c` komutu yazılır.

```
C:\Users\ILHANOZKAN>notepad hello.cpp
```

Eğer böyle bir dosya yoksa editör bu dosyayı oluşturmak için onay ister. Açılan metin editörü penceresine ilk C programı yazılır ve `hello.cpp` olarak dosyaya kaydedilir.

```
#include <iostream>

int main() {
    std::cout << "Hello World!";
    return 0;
}
```

Aşağıdaki komut ile `hello.cpp` olarak hazırlanan kaynak kodumuz derleyici tarafından icra edilebilir `hello.exe` dosyası olarak derlenir.

```
C:\Users\ILHANOZKAN>g++ hello.cpp -o hello.exe
```

Derlenen programı çalıştırmak için yine aynı komut satırında `hello.exe` yazarak programımız işletim sistemi tarafında işlemciye hedef gösterilerek çalıştırılır.

```
C:\Users\ILHANOZKAN>hello.exe
Hello, World!
C:\Users\ILHANOZKAN>
```

Burada programın yazılması `notepad` programında, derlenmesi `g++` programı ile ve derlenen `hello.exe` programının çalışması tarafımızdan yapılmıştır.

Mac OSX ve Linux işletim sitemlerinde de derleyici kurulumu tamamlandıktan sonra terminal yani konsol uygulaması açılarak aşağıdakine benzer şekilde derlemeler yapılabilir.

Entegre Geliştirme Ortamı

Şimdiye kadar anlatılan kod yazma (implementation), derleme (compile), icra (execute) ya da koşma (run) ve izleme (trace) gibi süreçlerin tamamını tek bir çatı altında yürütmemizi sağlayan programlar, entegre geliştirme ortamı (Integrated Development Environment-IDE) olarak adlandırılır. C++ programlama dili için en çok kullanılan entegre geliştirme ortamları aşağıda verilmiştir;

- Code::Blocks – Açık kaynaklı bir C/C++ geliştirme ortamı olup bir grafik kullanıcı ara yüzüne (Graphic User Interface-GUI) sahiptir. Windows, macOSX ve Linux işletim sitemleri üzerine kurulabilmektedir.
- Visual Studio – Microsoft tarafından geliştirilen, C dilinde yazılmış, güçlü, yüksek performanslı uygulamalar oluşturmak için kullanılabilen bir geliştirme ortamıdır. Yalnızca Windows'ta çalışır. Visual Studio, kod tamamlama (intellisense), kullanıcı ara yüzü (user interface-UI) hazırlama desteği ve hata ayıklama (debugger) ve birçok eklentisi (plug-in) olan muazzam özelliklere sahiptir. Bu geliştirme ortamının kurulumu dipnotta verilmiştir[13].

[13] https://www.programiz.com/c-programming/getting-started

- Visual Studio Code – Microsoft tarafından geliştirilen açık kaynaklı bir geliştirme ortamıdır. Windows, macOS ve Linux gibi işletim sistemlerinde çalışır. Git sürüm kontrol (version control) sistemleriyle çok iyi çalışır. Ayrıca, akıllı kod tamamlamanın dikkate değer özellikleriyle birlikte gelir.
- CLion – Jetbrains tarafından geliştirilmiştir ve C++ programcıları için en çok önerilen çapraz platformlu (CMake derleme sistemiyle entegre macOS, Linux ve Windows'u destekler) geliştirme ortamıdır. Ayrıca, yerel (local) ve uzaktan (remote) desteğe sahip birkaç IDE'den biri olarak kabul edilir. Bu da yerel bir makinede kod yazmanıza ancak uzak sunucularda derlemenize olanak tanır. Kaynak kodlarımız yönetmemizi sağlayan Concurrent Versions System-CVS ve Team Foundation Server-TFS ile entegre edilebilir.
- Eclipse – Java'da yazılmış ve IBM tarafından geliştirilmiş ücretsiz ve açık kaynaklı bir geliştirme ortamıdır. Yaklaşık otuz programlama dilini desteklediği için geniş topluluk desteğiyle iyi bilinir. C++ için Eclipse IDE, kod tamamlama, otomatik kaydetme, derleme ve hata ayıklama desteği, uzak sistem gezgini, statik kod analizi, profilleme (profiling) ve yeniden düzenleme (refactoring) gibi beklenebilecek tüm özelliklere sahiptir. Ayrıca çeşitli harici eklentileri entegre ederek işlevselliğini genişletebilirsiniz. Çok platformludur ve Windows, Linux ve macOS'ta çalışabilir.
- NetBeans – Apache Yazılım Vakfı – Oracle Corporation tarafından geliştirilen ücretsiz ve açık kaynaklı bir geliştirme ortamıdır. Öğrenciler veya başlangıç seviyesindeki C/C++ geliştiricileri için şiddetle tavsiye edilmesinin sebebi, Eclipse'e benzer şekilde daha iyi sürükle ve bırak işlevlerine sahip olmasıdır. Windows, Linux, MacOSX ve Solaris gibi birden fazla platformda çalışır.
- Xcode – MacOSX işletim sisteminde yazılım geliştirmek için kullanılan bir entegre geliştirme ortamıdır.

Entegre geliştirme ortamları metin dosyası olarak tutulan kaynak kodları programcıya renklendirerek gösterir. Bu da programcının kodu anlamasını daha da kolaylaştırır. Ancak kodu dosyaya yine sade metin olarak kaydeder.

Çevrimiçi Derleyiciler

Bütün bu entegre geliştirme ortamlarının yanında online olarak da derleme yapılan web uygulamaları da bulunmaktadır. Bunlardan birkaçı aşağıda verilmiştir;

- https://www.tutorialspoint.com/compile_cpp_online.php
- https://www.onlinegdb.com/online_c++_compiler
- https://www.programiz.com/cpp-programming/online-compiler/
- https://onecompiler.com/cpp

Derleme Zamanı

Derleyici programının derleme işlemine başlayıp bitirildiği sürece derleme zamanı (compile time) denir. Bu süreç başarısızlıkla da sonuçlanabilir ve eğer derleme işleminde hata meydana gelirse programcı hata mesajları ile uyarılır.

Bir derleyici program, kaynak dosyayı makine diline çevirme çabasında, kaynak dosyanın C dilinin sözdizimi kurallarına uygunluğunu da denetler. Eğer dilin kurallarına uyulmamışsa, derleyici bu durumu bildiren bir ileti de vermek zorundadır.

```
#include <iostream>

int main() {
    std::cout << "Hello World!";
    return;
}
```

Yukarıda hatalı yazılmış bir c programı derlendiğinde aşağıdaki hata alınacaktır.

```
C:\Users\ILHANOZKAN>g++ hello.cpp -o hello.exe
hello.cpp: In function 'int main()':
hello.cpp:5:5: error: return-statement with no value, in function returning 'int' [-
fpermissive]
```

```
    5 |     return;
      |     ^~~~~~
```

Kaynak kod içerisindeki metinlerde Türkçe karakter kullanılması halinde Windows altında derlenen programı çalıştırdığınızda aynı metni göremeyebilirsiniz. Bunu düzeltmek için komut satırında

```
C:\Users\ILHANOZKAN>Chcp 65001
```

Komutu verilerek komut satırının UTF-16 karakter kümesi ile işlem yapmasını sağlayabiliriz.

Hatalar

Programcı tarafından kodlama yapılırken genellikle aşağıdaki üç tip hata yapılır;

1. Derleme zamanı hataları (compile time error): Bu tip hatalar genelde kullanılan dilin yazım kurallarına (syntax) uyulmadığından, talimatların (statement) yanlış yazılmasından ya da programcının kod metninde uygun olmayan karakterlerin kullanılmasından kaynaklanır.
2. Koşma zamanı hataları (run time error): Kaynak kod, kurallara uygun olarak yazılmıştır ve herhangi bir yazım hatası bulunmaz. Bu tip hatalara en iyi örneklerden birisi sıfıra bölme hatasıdır. Taşma hataları da bu hatalardandır. Daha sonra değinilecektir.
3. Mantıksal hatalar (logical error): Programcının çözüm için gerekli adımların oluşturulmasında, çözüm yönteminin yanlış olmasından ya da yanlış işleçler (operator) kullanmasından kaynaklanır. Örneğin bir büyüktür (`>`) işleci yerine küçüktür (`<`) işleci kullanıldığında ne bir yazım hatası ne de bir çalışma zamanı hatası ortaya çıkar. Fakat program kendisinden istenilen davranışları yerine getirmez ve uygun çıktıları üretmez. Faktöriyel hesaplanırken `0!=1` yerine `0!=0` kabul etmek buna örnek verilebilir.

C++ Programlama Kullanım Alanları

C dili, başlangıçta sistem geliştirme çalışmaları için, özellikle işletim sistemini oluşturan programlar için kullanıldı. Montaj (assembly) dilinde yazılan kod kadar hızlı çalışan kod ürettiği için sistem geliştirme dili olarak benimsendi. Bilgisayar mühendislerinin bilmesi gereken bu dilin birçok kullanım alanı mevcuttur. C++ ise;

- Çok kullanılan ve popüler programlama dillerinden biridir. C++ işletim sistemleri, gömülü sistemler ve Grafiksel Kullanıcı Ara yüzleri yapımında kullanılır.
- Nesne yönelimli programlamayı destekler ve Soyutlama (abstraction), Sarma (encapsulaton) ve Miras alma (inheritance) gibi tüm OOP kavramlarını uygular, bu da programlara net bir yapı kazandırır ve kodun yeniden kullanılmasına olanak tanır, bu da geliştirme maliyetlerini düşürür ve güvenlik sağlar.
- Sözdizimi C diline benzediği için programcıların Java ve C# dillerine geçişini kolaylaştırır. C++ dili ara dil (intermediate language) kullanmaz de kodu doğrudan işlemcinin emir kodlarına derler. Bu nedenle C#, Python ve Java'dan daha hızlıdır.
- C'nin de bir seçenek olduğu sınırlı kaynak ortamlarında kullanışlıdır. Ayrıca yürütme hızına ihtiyaç duyduğumuz veya donanıma yakın çalışmamız gereken yerlerde de kullanılır.
- C ile karşılaştırıldığında C++, daha zengin kütüphanelere sahiptir, nesne yönelimli programlamayı destekler, şablonlar, istisna işleme ve daha birçok özelliği barındırır.

C++ PROGRAMLAMA DİLİNE GİRİŞ

En Basit C++ Programı

C programlama dili, her programcının ihtiyacını görecek genel amaçlı, emreden paradigmalı (kısa ve öz talimatlar birbiri ardına verilerek yapılan programlama), yapısal programlama dilidir. Yapısal programlamanın çerçevesi *Yapısal Programlama* altında anlatılmıştır. C programlama dilinin bir uzantısı olan C++, nesne yönelimli programlama mantığını programcıların yapabilmesi için geliştirilmiştir.

Yapısal programlamada programın başladığı yeri belirten bir ana fonksiyon (main function) tanımlanır. C dilinde bu fonksiyon `main()` fonksiyonudur. C++ dili de C dilinin uzantısı olduğundan ana fonksiyon yine `main()` fonksiyonudur. Yani içinde main fonksiyonu yazılmamış bir C++ programı çalışmaz. C dili gibi C++ dilinde de talimatlar gelişigüzel (free format) yazılır.

Aşağıda en basit C++ programı verilmiştir.

```
int main()
{
   return 0;
}
```

C++ dilinde her fonksiyon tırnaklı parantez karakteri ile başlayan biten bir kod bloğuna sahiptir ve fonksiyona ait bu blok, fonksiyon bloğu (function block) olarak adlandırılır. Ana fonksiyonunun başındaki `int`, ana fonksiyonun bir tamsayı (integer) geri döndüreceğini belirtir. Ana fonksiyon olan `main` fonksiyonu içinde kullanılan `return 0;` talimatı (statement) main fonksiyonundan sıfır geri döndürerek çıkıldığını söyleyen talimattır. Bu talimatın kullanılma zorunluluğu bulunmamaktadır. Ana fonksiyondan çıkıldığında programın sonlanıp işletim sistemine dönüleceğinden işletim sistemine `0` sayısını varsayılan (default) olarak gönderir. Sıfır dışında bir değerin işletim sistemine gönderilmesi return talimatıyla yapılır.

```
...Program finished with exit code 0
```

Bu durumda işletim sistemine 11 sayısını gönderen program aşağıdaki şekilde yazılır;

```
int main() {

   return 11;
}
```

Program çalıştırıldığında aşağıdaki çıktı elde edilir;

```
...Program finished with exit code 11
```

C++ Programlama Dili Söz Dizim Kuralları

C++ dili için oluşturulacak kaynak kod programcı tarafından metin dosyasına yazılırken belli söz dizim kullarına (syntax) uyar. Bunlar aşağıda başlıklar halinde verilmiştir.

Kaynak Kodumuzu Oluşturacak Karakterler

C Dilinde olduğu gibi C++ dilinde de kaynak kodu oluşturan karakterler aşağıda verilmiştir;

1. İngiliz alfabesindeki büyük harfler:
   `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
2. İngiliz Alfabesindeki küçük harfler:
   `a b c d e f g h i j k l m n o p q r s t u v w x y z`
3. Rakamlar: `0 1 2 3 4 5 6 7 8 9`
4. Özel Karakterler: `< > . , _ ( ) ; $ : % [ ] # ? ' & { } " ^ ! * / | - \ ~ +`
5. Metinde görünmeyen ancak metni biçimlendiren beyaz boşluk (white space) karakterleri:

a. ␣ Boşluk (space)
b. ↲ Yeni satır (new line)
c. ⇤ Satır başı (carriage return)
d. ⇦ Geri (backspace)
e. ⇥ Sekme (tab)

Talimatlar ve Anahtar Kelimeler

Anahtar kelimeler (keywords), programlamada kullanılan ve C++ derleyicisi tarafından özel anlama sahip, önceden tanımlanmış, ayrılmış (reserved) kelimelerdir. Başka hiçbir amaç için veya değişkenler veya fonksiyon adları gibi yerlerde kimliklendirme için kullanılamazlar. C++ sözdiziminin bir parçası olan önceden tanımlanmış sözcüklerdir. Kodun yapısını ve davranışını tanımlamaya yardımcı olurlar. Aslında programı oluşturan talimatlar (statement), bu kelimeler kullanılarak yazılırlar. Küçük harflerle yazılırlar. C++ dilinde, C dilinde olduğu gibi küçük-büyük harf ayrımı vardır.

A-C	D-P	R-X
`alignas` (C++11)	`decltype` (C++11) (2)	`reflexpr`
`alignof` (C++11)	`default` (1)	`register` (3)
`and`	`delete` (1)	`reinterpret_cast`
`and_eq`	`do`	`requires` (C++20)
`asm`	`double`	`return`
`atomic_cancel`	`dynamic_cast`	`short`
`atomic_commit`	`else`	`signed`
`atomic_noexcept`	`enum` (1)	`sizeof` (1)
`auto` (1) (3) (4) (5)	`explicit`	`static`
`bitand`	`export` (1) (4)	`static_assert` (C++11)
`bitor`	`extern` (1)	`static_cast`
`bool`	`false`	`struct` (1)
`break`	`float`	`switch`
`case`	`for` (1)	`synchronized`
`catch`	`friend`	`template`
`char`	`goto`	`this` (5)
`char8_t` (C++20)	`if` (3) (5)	`thread_local` (C++11)
`char16_t` (C++11)	`inline` (1) (3)	`throw` (3) (4)
`char32_t` (C++11)	`int` (1)	`true`
`class` (1)	`long`	`try`
`compl`	`mutable` (1)	`typedef`
`concept` (C++20)	`namespace`	`typeid`
`const`	`new`	`typename` (3) (4)
`consteval` (C++20) (5)	`noexcept` (C++11)	`union`
`constexpr` (C++11) (3)	`not`	`unsigned`
`constinit` (C++20)	`not_eq`	`using` (1) (4)
`const_cast`	`nullptr` (C++11)	`virtual`
`continue`	`operator` (1)	`void`
`co_await` (C++20)	`or`	`volatile`
`co_return` (C++20)	`or_eq`	`wchar_t`
`co_yield` (C++20)	`private` (4)	`while`
	`protected`	`xor`
	`public`	`xor_eq`

Tablo 6. Anahtar Kelime Listesi[14]

Talimatlar (statement), yapılaması gerekeni kısa ve net olarak belirten mantıksal cümlelerdir (logical sequences). C++ programlama dilinde her talimat, anahtar kelimeler içerecek şekilde yazılır ve noktalı virgül karakteri ile biter.

[14] https://en.cppreference.com/w/cpp/keyword-Burada; (1) C++11, (2) C++14, (3) C++17, (4) C++20 ve (5) C++23 uyarlamasında değişti veya eklendi anlamındadır.

Yüksek seviyeli bir dilde yazılmış bir talimat (statement), işlemciye belirtilen bir eylemi gerçekleştirmesini söyleyen cümledir. Yüksek seviyeli bir dildeki tek bir talimat, birkaç makine dili emir kodunu (instruction code) temsil edebilir. Talimatlar, yapılaması gerekeni kısa ve net olarak belirten mantıksal satırlardır (logical sequences).

Açıklamalar

Programcı kodu yazarken açıklama (comment) yapma gereği duyabilir. Bunu iki şekilde yapar;

1. Kod metninde bulunulan yerden sonra satırın sonuna kadar olan bir açıklama yapılmak istendiğinde, açıklama öncesinde iki adet bölü karakteri kullanılır.
2. Eğer birden çok satırı içeren bir açıklama yapılacak ise açıklama `/*` ile `*/` karakterleri arasına alınır.

Aşağıda açıklama eklenmiş bir ana fonksiyon örneği bulunmaktadır.

```
/*
Bu program, açıklama (comment) içeren basit bir ana fonksiyonu olan C++ programıdır.
İlhan ÖZKAN, Ocak 2025
*/
int main() // Ana Fonksiyon
{   //fonksiyon bloğu başlagıcı
    /*
     Ana Fonksiyon, programın başladığı yerdir.
     Ana fonksiyonu olmayan bir program çalışmaz.
    */
} //fonksiyon bloğu bitişi
```

C++ dilinde kodu yazarken kodun bir başka programcı tarafından anlaşılması için bir kod bloğu içindeki açıklama ve talimatları bloktan itibaren girintili olarak yazarız.

```
int main()
{
    //Ana fonksiyon bloğuna ait girinti
    {
        //iç bloğa ait girinti...
        return 0;
    }
}
```

Kodu bu şekilde yazmamızın derleyici açısından hiçbir önemi yoktur. Derlemenin ilk aşamasında bütün bu açıklamalar ve beyaz boşluk karakterleri metinden çıkartılır. Aşağıda en kısa C++ programı verilmiştir.

```
int main(){}
```

Değişken Tanımlama

C++ dilinde fonksiyonlar da yapısal programlama dilinde olduğu gibi tanımlanır. Veri yapıları (data structure) ve kontrol yapıları (control structure) birbirinden ayrıdır. Dolayısıyla veriyi işleyecek kontrol yapısı kodlanmadan önce veriyi taşıyan veri yapıları tanımlanmalıdır.

Değişkenler veri yapılarının temel öğeleridir. Değişkenin (variable) ne olduğu *Değişken* başlığı altında verilmiştir. Değişkenler, belleğin belli bir bölgesinde yer kaplayan ve çeşitli biçimde veri tutarlar. C++ dilinde verinin bellekte kapladığı yer ve biçimi için kullanılan ilkel veri tipleri (primitive data type) için anahtar kelimeler aşağıda verilmiştir. Aynı zamanda Temel veri tipi (basic data type) olarak da adlandırılırlar; C dilinden de bildiğimiz veri tipleri:

– 8 bit tamsayılar için `char`, 16/32 bit tamsayılar için `int`, 64 bit tamsayılar için `long` veri tipleri tamsayı veri tiplerini tanımlamamızı sağlayan anahtar kelimelerdir.
– 32 bit tek hassasiyetli kayan noktalı sayılar için `float`, 64 bit çift hassasiyetli kayan noktalı ayılar için `double` gerçek sayı veri tiplerini tanımlamamızı sağlayan anahtar kelimelerdir.

– `void` değer veya veri tipi olmadığını belirtir ve genellikle değer döndürmeyen işlevler için kullanılan anahtar kelimedir.

C++ diline eklenen veri tipleri:

– `bool` sadece doğru veya yanlış bilgisini tutabilen (boolean) veri tipini tanımlamak için kullanılır.
– `wchar_t` özellikle kodumuzu çok dilli çalıştırmak yani uluslararası uygulamalar için yararlı olan geniş karakter tipini temsil eden değişken tanımlamak için kullanılan anahtar kelimedir.

C++ programlama dilinde hiçbir değeri olmayan ve bellekte yer kaplamayan `void` veri tipi vardır. Bu veri tipini hiçbir değer döndürmeyen fonksiyon tanımlamalarında ileride çokça göreceğiz.

Değişkenlerin veri tipine bağlı olarak kimlik verilerek tanımlanması (definition) gerekir. Değişken tanımlaması yapılabilmesi için ilk önce veri tipinin (data type) belirlenmesi ve değişkenin kullanılacağı yerler göz önüne alınarak benzersiz bir kimlik (identifier) verilir. Bu işleme değişkenin bildirimi (variable declaration) adı da verilir.

```
/* Bu program değişken tanımlamak için oluşturulmuştur.*/
int main() {
    char yas; // veri tipi "char" ve kimliği "yas" olan değişken bildirimi
    int kat;  // veri tipi "int" ve kimliği "kat" olan değişken bildirimi
    float kilo; // veri tipi "float" vekimliği "kilo" olan değişken bildirimi
}
```

Yukarıda tanımlaması yapılan `yas`, `kat` ve `kilo` değişkenleri ana fonksiyon olan `main` fonksiyon bloğu içinde tanımlandığından, tanımlanan değişkenler sadece bu blok içinde geçerlidir. Aynı blok içinde bu değişken kimlikleri bir başka değişkene verilemez.

Bildirim (declaration) derleyiciye böyle bir değişken, fonksiyon veya nesne var demenin adıdır. Tanımlama (definition) derlemenin sorunsuz olabilecek şekilde eksiksiz yapılan bir işlemdir. Değişken bildirimi ve tanımlaması arasındaki fark için *Harici Depolama Sınıfı* başlığını inceleyebilirsiniz. *Kullanıcı Tanımlı Fonksiyonlar* başlığı altında fonksiyon bildirimi ve fonksiyon tanımlaması arasındaki farkı inceleyebilirsiniz. Değişkenler için ise

Bir de yukarıda verilen temel veri tiplerini değiştiren veri tipi değiştiricileri (data type modifier) vardır. Değiştiriciler için kullanılan anahtar sözcükler;

– Bir değişkenin değerinin başlatmadan sonra değiştirilemeyeceğini `const` belirtir.
– Bir veri tipinin hem pozitif hem de negatif değerleri tutabileceğini `signed` belirtir. Örneğin 8 bitlik `char` veri tipinin en anlamlı olan 8. bitini işaret biti olarak kullanılmasını sağlar. Böylece -127 ile +128 arasındaki sayılar `signed char` olarak tanımlanabilir.
– Bir veri tipinin yalnızca negatif olmayan değerleri tutabileceğini `unsigned` belirtir. Örneğin 8 bitlik `char` veri tipinin en anlamlı olan 8. bitini sayı olarak kullanılmasını sağlar. Böylece 0 ile 255 arasındaki sayılar `unsigned char` olarak tanımlanabilir.
– Tamsayı tipinin daha kısa bir sürümünü `short` belirtir. Yani 16 bit tamsayı tanımlamak için `short int` veri tipi kullanılabilir.
– Tamsayı tipinin daha uzun bir sürümünü `long` belirtir. Yani 64 bit tamsayı tanımlamak için `long int` veri tipi kullanılabilir.
– Bir değişkenin değerinin programcının hazırladığı mevcut kod dışında beklenmedik şekilde değişebileceğini (örneğin bir aygıt tarafından, bir kesme/interrupt tarafından, ...) ve derleyicinin bazı optimizasyonları yapmaması gerektiğini `volatile` belirtir. Derleyiciye bir değişkenin değerinin kodu yazılmış program tarafından değil de başka şekillerde değiştirilebileceğini söyler.

```
/* Bu program işaretsiz tamsayı değişken tanımlamak için oluşturulmuştur.*/
int main() {
    unsigned char ogrenciYasi; /* veri tipi "unsigned char" ve kimliği "ogrenciYasi" olan
                                  değişken bildirimi. Bu değişken 0 ile 255 arasında bir değer
                                  alabilir. */

    unsigned short asansorunBulunduguKat;  /* veri tipi "unsigned short" ve kimliği
                                              "asansorunBulunduguKat" olan değişken
                                              bildirimi. Bu değişken, 0 ile 65.535 arasında
```

```
                                                değer alabilen bir değişkendir */

    unsigned int pozitifTamsayi; /* pozitifTamsayi değişkeni
                                    0 ile 4.294.967.295 arasında
                                    değer alabilen bir değişkendir. */

    volatile int deviceStatus=0; /* Programcının yazacağı kod
                                    dışında bu değişken değiştirilebilir! */
}
```

Değişkenin kapsamı (variable scope); Değişkenin bildiriminin (variable declaration) yapılabileceği, değişkene erişilebileceği (access), değişkenin üzerinde çalışılabileceği program kodu olarak tanımlanır. Esas olarak iki türü vardır; Yerel değişkenler (local variable) bulunduğu kod bloğu içerisinde tanımlanıp kullanılabilen değişkenlerdir. Evrensel değişkenler (global variable) ise bir fonksiyon bloğu dışında tanımlanan ve tanımlandığı yerden sonra geri kalan tüm kod içerisinden erişilebilen değişkenlerdir.

Değişken Kimliklendirme Kuralları

C programlama dilinde değişkenlerin kimliklendirilmesinde (identifier definition) aşağıda verilen kurallara uyulur;

1. Kimliklendirmede anahtar kelimeler kullanılamaz!
2. Kimliklendirme rakamla başlayamaz!
3. Kimlikler en fazla 32 karakter olmalıdır! Daha fazlası derleyici tarafından dikkate alınmaz!
4. Kimliklendirmede ancak İngiliz Alfabesindeki Büyük ve Küçük harfler, rakamlar ile altçizgi karakteri kullanılabilir.

```
/* Bu program değişken tanımlama örneklerini içerir. */
int main() {
    //Tamsayı değişkenler:
    char yas;
    int tam_sayi;
    long uzun_tam_sayi;

    //Gerçek Sayı Değişkenler:
    float kilo;
    double reel_sayi;

    /*Aynı veri tipinde birden fazla değişkene aralarına virgül koyarak kimlik verilebilir. */
    int kat1,kat2,kat3;
    float olcek1,olcek2;
}
```

Bütün bu kuralların yanında değişkenlere kimlik verilirken yazılan kodun okunaklılarını artırmak için camel case olarak kimlik verilir. Bu kimliklendirme türünde ilk sözcük tamamıyla küçük, izleyen sözcüklerin ise baş harfi büyük yazılır. Bu kural zorunlu değildir ama koda bakım yapacak sonraki programcıların işini kolaylaştırmak için ahlaki olarak tercih edilir.

```
/* Bu program camelCase değişken kimliklendirme örneklerini içerir.*/
int main() {
    int sayac;
    int ogrenciBoyu;
    int asansorunOlduguKat;
    float asansorAgirligi;
    int ogrenciYasi;
    char ilkHarf;
    float ortalamaNot;
}
```

Değişmezler

Programcı, kodlama yaparken bazı sabitleri (constant) ister istemez kullanır. Bunların biri değişmez (literal), diğeri ise değeri değiştirilemeyen sabit değişkenlerdir (const variable). Derleme öncesinde de sonrasında da kaynak koddakinin kelimesi kelimesine aynısı olan değişmezler (literal) kullanılır. Değişkenlere verilen ilk değerler (initial value) ile katsayılar (coefficient) en çok kullanılan değişmezlerdir. Değişkenler tanımlanırken sahip olacağı ilk değerleri belirten değişmezler ile verilebilir. Değişkenlere ilk değer verme işlemine başlatma (initialization) adı verilir. Aşağıda bu değişmezlere ilişkin örnek verilmiştir.

```
/* Bu program değişkenlere ilk değer verme ve değişmez örneklerini içerir. */
int main() {
    int i,j=0; /* "i" ve "j" kimlikli "int" veri tipindeki değişken tanımlandı.
                  "j" değişkenine 0 değişmezi ile ilk değer verildi */

    char c=65; /* "c" kimlikli "char" veri tipindeki değişkene 65 değişmezi
                  ile ilk değer verildi. */
    float f=2.5; /* "f" kimlikli "float" veri tipindeki değişkene Türkçe 2,5
                    değişmezi ile ilk değer verildi. Kod ingilizce yazıldığından,
                    kodun içerisindeki gerçek sayı sabitleri için
                    ondalık ayracı NOKTA olarak yazılır. */
    char harf='A';  /* "harf" kimlikli "char" veri tipindeki değişkene
                       'A' karakter değişmezi ile ilk değer verildi.
                       Klavyeden basılan her bir harf de bellekte tamsayı
                       olarak tutulur;
                       'A' karakterinin kodu 65,
                       'B' karakterinin kodu 66'dır... Bu ANSI Standardıdır.
                       Bu nedenle bu talimat aşağıdaki şekilde de yapzılabilir;
                       char harf=65;
                       Bellekte 1 bayt yer kaplayan "char" 255 farklı
                       karakteri gösterebilir.
                       Kodu 0 olan karakter ise NULL karakter olarak adlandırılır.
                       Günümüzde ANSI yerine UTF-8 ve UTF-16 karakter kümesi
                       kullanılmaktadır.*/
    bool kosul=false; /* "kosul" kimlikli "bool" veri tipindeki değişkene
                         false değişmezi ile ilk değer verildi.
                         false yerine 0 true yerine 1 de yazılabilir.*/
}
```

Tamsayı değişmezleri her zaman onluk tabanda verilmeyebilir. Bazı durumlarda onlatılık (hexal), sekizlik (octal) veya ikili (binary) olarak verilmek istenebilir;

```
int onluk = 61;
int onlatilik = 0x4b; //Onluk olarak 4x16+11=75
int Onlatilik = 0X4B;
int sekizlik = 052; //Onluk olarak 5x8+2=42
int ikili = 0b101010 // C++14 ve sonrasında
```

Aşağıdaki kod örneğinde programcı, `3`, `3.14`, `3.14` ve `2.0` olmak üzere dört farklı değişmez kullanılmıştır. Bu sabitlerin her biri derleyici tarafından farklı olarak değerlendirilir.

```
int main() {
   int yariCap=3;
   float daireninAlani=3.14*yariCap*yariCap;
   float daireninCevresi=2.0*3.14*yariCap;
}
```

Değişmezlerin (literal) çokça kullanılması derleme sonrası üretilen icra edilebilir makine kodunu da büyütür. Bunun yerine çok kullanılan değişmezler bellekte 1 kez yer kaplasın diye const sıfatıyla (const qualifier) sabit değişkenler (const variable) tanımlanabilir. Sabit değişkenler etik olarak büyük harfle kimliklendirilir.

```
/* Bu program, const sıfatı (const qualifier) örneğidir. */
int main() {
   int yariCap=3;
   const float PI=3.14;
   float daireninAlani=PI*yariCap*yariCap;
   float daireninCevresi=2.0*PI*yariCap;
}
```

Böylece aynı sabit `PI` değişkeni birden çok yerde kullanılır ve her seferinde aynı bellek bölgesine erişildiğinden bellekten tasarruf edilmiş olunur. Yani `PI` değişkeninin çalıştırma anında sabit olduğu bilinir ve değiştirilmez. Tanımlama (definition) sırasında const sıfatı kullanılan değişkene ilk değer (initial value) bir değişmez (literal) ile verilmelidir. Daha sonra bu değişkene bir değer ataması olamaz!

Bazı durumlarda derleme anında da değişkenlerin sabit olduğu derleyiciye `constexpr` anahtar kelimesiyle söylenebilir. Böyle bir sabit aşağıdaki şekilde tanımlanabilir;

```
int main() {
   int constexpr birHaftadakiGunSayisi = 7;
   //...
}
```

Bir değişkeni sabit tanımlamanın üstünlükleri aşağıda sıralanmıştır;

- Gelişmiş Kod Okunabilirliği: Bir değişkene const sıfatı vermek, diğer programcılara değerinin değiştirilmemesi gerektiğini belirtir; bu da kodun anlaşılmasını ve sürdürülmesini kolaylaştırır.
- Gelişmiş Veri Tipi Güvenliği: const kullanılması, değerlerin yanlışlıkla değiştirilmemesini sağlayabilir ve kodumuzda hata ve eksiklik olma olasılığını azaltabilir.
- Gelişmiş Optimizasyon: Derleyiciler, değerlerinin program yürütme sırasında değişmeyeceğini bildikleri için const değişkenlerini daha etkili bir şekilde optimize edebilirler. Bu, daha hızlı ve daha verimli kodla sonuçlanabilir.
- Daha İyi Bellek Kullanımı: Değişkenlere const sıfatı vermek, değerlerinin bir kopyasını oluşturmayı engeller; bu da bellek kullanımını azaltabilir ve performansı artırabilir.
- Gelişmiş Uyumluluk: Değişkenlere const sıfatı vermek, kodunuzu const değişkenleri kullanan diğer kitaplıklar ve API'lerle daha uyumlu hale getirebilir.
- Gelişmiş Güvenilirlik: const kullanarak, değerlerin beklenmedik şekilde değiştirilmemesini sağlayarak kodunuzu daha güvenilir hale getirebilir ve kodunuzdaki hata ve eksiklik riskini azaltabilirsiniz.

Kod içerisinde değişmezleri (literal) tekrar tekrar yazmamanın bir yolu da `#define` ön işlemci yönergesi (preprocessor directive) ile makro tanımlamaktır. Ön işlemciler, kaynak kodları derlemeden önce derleyiciyi yönlendirerek kaynak kodlar üzerinde işlem yapmayı sağlar. Aşağıdaki örnekte derleyici, derlemeye geçmeden `PI` görülen yerleri `3.14` olarak değiştirmesi söylenir. Derleme bu ön işlemci işleminden sonra gerçekleşir. Sabit değerler makro ile tanımlanırken yine büyük harfle kimliklendirilir. Makrolar sonunda talimatler gibi noktalı virgül karakteri ile bitmez. Çünkü ön işlemci yönergeleri kodların tasnifi ve derleme önhazırlığını yaparlar.

```
/* Bu program, #define önişlmeci direktifi (preprocessor directive) örneğidir. */

#define PI 3.14  /*Bu makro, derleme öncesinde PI görülen yerlere 3.14 yazılmasını sağlar.*/

int main() {
   int yariCap=3;
   float daireninAlani=PI*yariCap*yariCap;
   float daireninCevresi=2.0*PI*yariCap;
   return 0;
}
```

Ön işlemci yönergesini yerine getiren derleyici aynı kodu aşağıdaki şekle çerirerek derleme işlemine geçer. Ayrıca ön işlemci yönergeleri yerine getirilmeden önce bütüm açıklamalar ve beyaz boşluklar (white space) kaynak koddan kaldırılır.

```
int main() {
   int yariCap=3;
   float daireninAlani=3.14*yariCap*yariCap;
   float daireninCevresi=2.0*3.14*yariCap;
   return 0;
}
```

`#define` ön işlemci yönergesi çok tekrarlanan sabitler için kullanılabileceği gibi aşağıdaki şekilde de kullanılabilir;

```
/* Bu program, #define önişlmeci yönergesinin bir başka kullanımı örneğidir. */

#define BEGIN int main() {
#define END }
#define PI 3.14

BEGIN
   const char ILKHARF='A';
   const float AVAGADROSAYISI=6.022e23;
   float yarimRadyan=PI/2;
   return 0;
END
```

Yazdığımız Kod Nasıl İcra Edilir?

Derleyici tarafından işlemcinin anlayacağı makine diline çevrilen programımız aslında yazdığımız talimatları (statement) sırasıyla çalıştırır. C dilinde her talimat noktalı virgül ile biter. Yapısal programlamada program, ana fonksiyondan başlar. Bu nedenle icra, `main` fonksiyonu içindeki ilk talimatla başlar ve solunda verilen sırayla icra edilir.

		boy	kilo	bki	boyKare
	`int main() {`				
1	`   float boy=1.80;`	1.80	?	?	?
2	`   float kilo=100.0;`	1.80	100.0	?	?
3	`   float bki;`	1.80	100.0	?	?
4	`   float boyKare;`	1.80	100.0	?	?
5	`   boyKare=boy*boy;`	1.80	100.0	?	3.24
6	`   bki=kilo/boyKare;`	1.80	100.0	30.86419753	3.24
7	`   boy=1.50;`	1.50	100.0	30.86419753	3.24
	`}`				

Tablo 7. Örnek bir C++ Programının İcra Sırası

Tablonun sağında ise her icra sonrası değişkenlerin değerleri gösterilmiştir. Her talimatın icrasında neler olduğu aşağıda verilmiştir;

1. `boy` değişkenine `1.80` değeri atanır.
2. `kilo` değişkenine `100.0` atanır.
3. `bki` değişkeni tanımlanır.
4. `boyKare` değişkeni tanımlanır. Daha sonraki birkaç satır açıklama olduğundan icra edilecek bir şey yoktur.
5. `boy` ile `boy` değişkeni çarpılır ve sonuç `boyKare` değişkenine atanır. Artık `boyKare` değişkeninin değeri bu noktadan sonra `3.24` olmuştur.
6. `kilo` değişkeni `boyKare` değişkenine bölünür ve sonuç `bki` değişkenine atanır. Artık `bki` değişkeninin değeri bu noktadan sonra `30.86419753` olmuştur.
7. `boy` değişkenine `1.50` atanmıştır. Bu atama sonrası `boyKare` ve `bki` değişkenlerinin değeri değişmemiştir. Çünkü bu değişkenleri değiştiren 5. ve 6. talimatlar icra edilmiştir.

İşleçler

İşleçler (operator) matematikten bildiğimiz işleçlerdir.

$$y = 3 + 2$$

Yukarıda verilen cebirsel ifadede toplama işleci (`+`) iki tarafına bulunan argümanları toplar. Bu argümanlara işlenen (operand) adı verilir. Yüksek düzey dillerde de aritmetik işleçler ve bunun yanında birçok işleç de bulunur. En çok kullanılan işleç, atama (`=`) işlecidir. Atama işleci, sağdaki işleneni soldakine atar. Bu nedenle kodlama yapılırken `2+2=x` veya `a+b=x` yazılamaz!

Aritmetik İşleçler

C++ programlama dilinde aritmetik işleçlerin (aritmetic operator) çalışma şekli işlenenlerin (operand) veri tipine göre değişir. Bu konu detaylı olarak *Üstü Kapalı Tip Dönüşümleri* başlığında detaylı anlatılmıştır. Aritmetik işlemler ve atama işleci için kurallar;

1. İşlenenlerden birinin bellekte kapladığı yer küçük ise ikisi aynı olacak şekilde bellekte daha fazla yer kaplayanına dönüştürülür ve işlem sonra yapılır ve sonuç dönüştürülen veri tipinde üretilir. Örneğin `char` veri tipindeki değişken ile `long` veri tipindeki bir değişken toplanmaya çalışıldığı zaman `char` veri tipindeki değişkenin değeri `long` veri tipine otomatik dönüştürülür. Sonuç `long` veri tipinde üretilir.
2. İşlenenlerden biri tamsayı diğeri kayan noktalı sayı ise işlem yapılmadan önce işlenenler kayan noktalı sayıya dönüştürülür. Örneğin `int` veri tipindeki değişken ile `float` veri tipindeki bir değişken toplanmaya çalışıldığı zaman `int` veri tipindeki değişkenin değeri `float` veri tipine otomatik dönüştürülür. Sonuç `float` veri tipinde üretilir.

İşleç	İşlenenler (operand) üzerinde yapılan işlem
`+`	Sol ve sağındaki yer alan işlenenleri toplar.
`-`	Soldaki işlenenden sağdakini çıkarır.
`*`	Sol ve sağındaki yer alan işlenenleri çarpar.
`/`	Soldaki işleneni sağdakine böler. Eğer işlenenlerin her iki de tamsayı ise sadece kalan kısmı atılır. Bu durumda son uç yine tamsayı olarak üretilir.
`%`	Kalan (modulus) işleci, sadece tamsayıları işlenen olarak kabul eder. Soldaki işlenenin sağdakine bölümünden kalanı verir. Kalan yine tamsayıdır.

Tablo 8. Aritmetik İşleçler

Aritmetik işlemler ve atama işlecine ilişkin yukarıda belirtilen kuralların çalıştığını aşağıdaki örnekte görebilirsiniz.

```
int main() {
   int a = 9, b = 4, res;
   float fa=6.6, fb= 2.2, fres;

   res = a + b; // işlem sonucu res=13
   res = a - b; // işlem sonucu res=5
   res = a * b; // işlem sonucu res=36
   res = a / b; // işlem sonucu res=9/4=2 tam 1/4=2
   /*
    toplama işlecinin işlenenleri tamsayı olduğundan,
    bölme sonucundaki tam kısım bölmesonucudur.
   */
   res = a / 4.3 ; /* işlem sonucu tamsayı olan res değişkeninde tutulacağından
                      res=9/4.3=9.0/4.3=2.0930= 2 tam 0.0930=2 */
   res = a % b; // işlem sonucu res=9%4= 2 tam 1/4=1
   fres= fa / fb; // işlem sonucu res=6.6/2.2=3.00
}
```

C programlama dilinde tamsayı bölme işlemi işlemcinin en basit yetenekleriyle çözülür. Bu nedenle C++ dışında diğer programlama dillerinden ayrılır. Aşağıda buna ilişkin örnek verilmiştir.

```
int main () {
   int a = 19 / 2 ; /* a = 2*9+1 = 9 */
   int b = 18 / 2 ; /* b = 9 */
   int c = 255 / 4; /* c = 4*63+3 = 63 */
```

```
   int d = 45 / 4 ; /* d = 4*11+1 = 11 */
   double aa = 19 / 2.0 ; /* aa = 9.5 */
   double bb = 18.0 / 2 ; /* bb = 9.0 */
   double cc = 255 / 2.0; /* cc = 127.5 */
   double dd = 45.0 / 4 ; /* dd = 11.25 */
   double ee = 45 / 4 ;   /* ee = 11 çünkü bölme tamsayılar arasında yapılmıştır */
}
```

Tekli İşleçler

Tekli işleçler (unary operator) sadece bir işlenen üzerinde işlem yapar. Tekli olarak adlandırılmasının sebebi tek bir işlenenle çalışmasıdır.

İşleç	İşlenenler (operand) üzerinde yapılan işlem
-	Kendisinden sonra gelen işlenenin işaretini değiştirir. İşlenen pozitif ise negatif yapar, negatif ise pozitif yapar.
+	Kendisinden sonra gelen işlenenin işaretini pozitif yapar.
++	Artırım (increment) işleci, işlenenin sağında veya solunda kullanılabilir. İşlenen solda ise işlenenin değeri kullanılmadan önce 1 artırılacaktır (pre-increment). İşlenen sağda ise işlenenin değeri kullanılmadan sonra 1 artırılacaktır (post-increment).
--	Eksiltme (decrement) işleci, işlenenin sağında veya solunda kullanılabilir. İşlenen solda ise işlenenin değeri kullanılmadan önce 1 eksiltilecektir (pre-decrement). İşlenen sağda ise işlenenin değeri kullanılmadan sonra 1 eksiltilecektir (post-decrement).
!	Mantıksal DEĞİL işleci işlenenden önce kullanılır. İşleneninin mantıksal durumunu tersine çevirmek için kullanılır.
sizeof()	`sizeof()` işleci kendisinden sonra gelen işlenenin bellek boyutunu bayt cinsinden döndürür.
&	Referans işleci (reference operator), kendisinden sonra gelen işlenenin bellek adresini döndürür.

Tablo 9. Tekli İşleçler

Aşağıda tekli işleçlerin kullanımına ilişkin örnek programı lütfen inceleyiniz.

```
int main() {
   int a = 5, b = 5, res;
   res= -a; // tekli eksi (unary minus): res=-5;
   res = ++a+2;
   /* ++a işlemi önce-artırım (Pre-Increment) olarak adlandırılır.
      a değişkeni işleme girmeden önce artırılır. İşlem sonucu res=8, a=6 */
   res = 2+b++;
   /* b++ işlemi sonra-artırım (Post-Increment) olarak adlandırılır.
      b değişkeni işleme girdikten sonra artırılır. İşlem sonucu res=7, b=6 */
   a=5; b=5; /* iki talimat(statement) yan yana yazılmıştır.
                Daha fazla da yazılabilir. */
   res = --a+2; //Pre-Decrement: işlemden sonra res=6, a=4
   res = 2+b--; //Post-Decrement: işlemden sonra res=7, b=4
   a=1; b=0;
   res=!a; // işlem sonucu res=0
   res=!b; // işlem sonucu res=1
   res=sizeof(char); /* char veri tipinin bellek miktarını öğrenme:
                        işlem sonucu res=1 */
   res=sizeof a; /* a değişkeninin bellek miktarını öğrenme:
                    işlem sonucu res=4 */
}
```

İlişkisel İşleçler

İlişkisel işleçler (relational operator), işlenenlerin arasındaki ilişkiyi verirler. Bu işleçler, karşılaştırma doğruysa `true`, değilse `false` değerini döndürür.

İşleç	İşlenenler (operand) üzerinde yapılan işlem
==	Eşit mi? İşleci: İki işlenenin değeri birbirine eşit ise 1, değilse 0 verir
!=	Farklı mı? İşleci: İki işlenenin değeri birbirinden farklı ise 1, değilse 0 verir

>	Büyük mü? İşleci: soldaki işlenenin değeri soldakinden fazla ise 1, değilse 0 verir
<	Küçük mü? İşleci: soldaki işlenenin değeri soldakinden az ise 1, değilse 0 verir
>=	Büyük veya eşit mi? İşleci: soldaki işlenenin değeri soldakinden fazla ya da iki işlenen eşit ise 1, değilse 0 verir
<=	Küçük veya eşit mi? İşleci: soldaki işlenenin değeri soldakinden az ya da iki işlenen eşit ise 1, değilse 0 verir

Tablo 10. İlişkisel İşleçler

Aşağıda verilen ilişkisel işleçlerin kullanımına ilişkin örnek programı lütfen inceleyiniz.

```
int main() {
    int a = 5, b = 6;
    res = a<b; // işlem sonucu res=true
    res = a<=b; // işlem sonucu res=true
    res = a>b; // işlem sonucu res=0
    res = a>=b; // işlem sonucu res=0
    res = a==b; // işlem sonucu res=0
    res = a!=b; // işlem sonucu res=true
}
```

Bit Düzeyi İşleçler

Bit düzeyi işleçler (bitwise operators) özellikle ikilik tabandaki tamsayılarda karşılıklı bitler üzerinde yapılan işlemlerdir. Aslında bu işlemler çarpma, toplama, bölme ve çıkarma işlemleri için kullanılır.

İşleç	İşlenenler (operand) üzerinde yapılan işlem
&	Bit düzeyi VE: Tamsayının karşılıklı bitleri VE işlemine tabi tutulur. Sonuç tamsayıda karşılaştırılan iki bit `1` ise sonuç bit `1`, değilse `0` olacak şekilde işlem yapar.
|	Bit düzeyi VEYA: Tamsayının karşılıklı bitleri VEYA işlemine tabi tutulur. Sonuç tamsayıda karşılaştırılan iki bitin herhangi biri `1` ise sonuç bit `1`, değilse `0` olacak şekilde işlem yapar.
^	Bit düzeyi ÖZEL VEYA: Tamsayının karşılıklı bitleri ÖZEL VEYA işlemine tabi tutulur. Sonuç tamsayıda karşılaştırılan iki bitin her ikisi farklı ise sonuç bit `1`, değilse `0` olacak şekilde işlem yapar.
~	Bit düzeyi NOT: Tekli işleç olup sonrasında gelecek sayının bitlerini `1` ise `0`, `0` ise `1` olacak olacak şekilde işlem yapar.
<<	Bit düzeyi sola kaydırma: Tekli işleç olup sonrasında gelecek sayının bitlerini bir bit sola kaydırır. En anlamlı bit taşar. Taşan bu bit `1` ise işlemcinin durum kaydedicisinde taşma bayrağı `1` olur.
>>	Bit düzeyi sağa kaydırma: Tekli işleç olup sonrasında gelecek sayının bitlerini bir bit sağa kaydırır. En anlamsız bit taşar. Taşan bu bit `1` ise işlemcinin durum kaydedicisinde taşma bayrağı `1` olur.

Tablo 11. Bit Düzeyi İşleçler

Aşağıda tamsayılar üzerinde yapılan bit düzeyi işlemler gösterilmiştir;

```
#include <iostream>
using namespace std;

int main() {
    int a = 6; //  0110
    int b = 2; //  0010

    int bitwise_and = a & b; // 0010
    int bitwise_or = a | b;  // 0110
    int bitwise_xor = a ^ b; // 0100
    int bitwise_not = ~b;    // 1101
    int left_shift = b << 2; // 0100
    int right_shift = b >> 1;// 0001
}
```

Mantıksal İşleçler

Mantıksal işleçler (logical operator), mantıksal ifadelerle çalışır. Mantıksal ifadeler doğru (true) ve yanlış (false) olabilen ifadelerdir. VE(AND), VEYA(OR), ÖZEL VEYA(XOR) ve DEĞİL (NOT) gibi mantıksal ifadelerin işlenmesi için aşağıdaki işleçler kullanılır. C++ dilinde bu işleçlere işlenen olarak giren ifadeler öncelikle `bool` veri tipine (data type) dönüştürülür. Yani bu işleçler için beklenen işlenen (operand) bir mantıksal ifadedir.

İşleç	İşlenenler (operand) üzerinde yapılan işlem
&&	Şartlı Mantıksal VE: Eğer iki işlenen `true` ise `true` verir. Soldaki işlenen `false` ise sağdakine bakılmaz.
||	Şartlı Mantıksal VEYA: Eğer iki işlenenden biri `true` ise `true` verir. Soldaki işlenen `true` ise sağdakine bakılmaz.
!	Mantıksal DEĞİL: İşlenenden önce kullanılır. Tekli işleçlerde verilmiştir. İşleneninin mantıksal durumunu tersine çevirmek için kullanılır. Eğer işlenen `true` ise `false`, değilse `true` verir.

Tablo 12. Mantıksal İşleçler

Aşağıda verilen ilişkisel ve mantıksal işleçlerin kullanımına ilişkin örnek programı lütfen inceleyiniz.

```
int main() {
    int a = 5, b = 6;
    bool res;
    res = a&&b; // işlem sonucu res=true
    res = a||b; // işlem sonucu res=true
    res = !a; // işlem sonucu res=false
}
```

Atama işleçleri

Atama işleçleri (assignment operator) en çok kullanılan işleçlerdir. Atama işleçleri her zaman sağdaki değeri sola atar! Dolayısıyla atama yapılacak uygun veri tipinde (data type) bir değişken (variable) olmak zorundadır.

İşleç	İşlenenler (operand) üzerinde yapılan işlem
=	Sağdaki işleneni sola atar.
+=	Soldaki işleneni sağdaki işlenen üzerine toplar ve sonucu sola atar.
-=	Soldaki işlenenden sağdaki işlenenden çıkarır ve sonucu sola atar.
*=	Soldaki işleneni sağdaki işlenenle çarpar ve sonucu sola atar.
/=	Soldaki işleneni sağdaki işlenene böler ve sonucu sola atar. Eğer işlenenler tamsayı ise kesirli kısım göz ardı edilir. Bu durumda sonuç tamsayıdır.
%=	Bu işleç tamsayılarla çalışır. Soldaki işleneni sağdaki işlenene böler ve kalanı sola atar.
<<=	İki işlenen tamsayının ikincisi kadar biti sola kaydırılır ve sonucu sola atar.
>>=	İki işlenen tamsayının ikincisi kadar biti sağa kaydırılır ve sonucu sola atar.
&=	İki işleneni bit düzeyi VE işlemine tabi tutar ve sonucu sola atar.
|=	İki işleneni bit düzeyi VEYA işlemine tabi tutar ve sonucu sola atar.
^=	İki işleneni bit düzeyi ÖZEL VEYA işlemine tabi tutar ve sonucu sola atar.

Tablo 13. Atama İşleçleri

Aşağıda verilen atama işleçlerin kullanımına ilişkin örnek programı lütfen inceleyiniz.

```
int main() {
  int a = 10; /* a değişkenine 10 değişmezi = işleciyle atanıyor.*/
  a += 10; /* a=a+10; ile eşdeğer: a değişkenine 10 ekleniyor. */
  a -= 10; /* a=a-10; ile eşdeğer: a değişkeninden 10 çıkarılıyor. */
  a *= 10; /* a=a*10; ile eşdeğer: a değişkeni 10 kat yapılıyor.*/
  a /= 10; /* a=a/10; ile eşdeğer: a değişkeni 1/10 kat yapılıyor.*/
  a %= 10; /* a=a%10; ile eşdeğer: a değişkeninin 10 a kalanı
              bulunup kendisine atanıyor. */
  int b=12;
  //b-5=a; // HATA: Atama sağdan sola yapıldığından böyle bir talimat yazılamaz!
```

```
}
```

İşleç Öncelikleri

Cebirsel ifadelerde nasıl ki önce çarpma ve bölme sonra toplama işlemleri yapılıyor. C programlama dilinde yazılan ifadelerde de işleç öncelikleri (operator precedence) vardır. İfadelerde bir işlemi öncelikli hale getirmek için parantez `()` içerisine alınır. En içteki parantez en öncelikli olarak gerçekleştirilir. Başka işleçler de bulunmaktadır bunlar ilerleyen bölümlerde yeri geldikçe anlatılacaktır.

Öncelik	İşleç Grubu	İşleçler	Gruptaki Öncelik
1	Sonek (postfix)	`() ++ --`	Soldan Sağa
2	Tekli (unary)	`+ - ! ++ -- sizeof()`	Sağdan sola
3	Çarpım (multiplicative)	`* / %`	Soldan Sağa
4	Toplam (additive)	`+ -`	Soldan Sağa
5	İlişkisel (relational)	`< <= > >=`	Soldan Sağa
6	Eşitlik (equality)	`== !=`	Soldan Sağa
7	Bit düzeyi VE	`&`	Soldan Sağa
8	Bit düzeyi ÖZEL VEYA	`^`	Soldan Sağa
9	Bit Düzeyi VEYA	`|`	Soldan Sağa
10	Mantıksal VE	`&&`	Soldan Sağa
11	Mantıksal VEYA	`||`	Soldan Sağa
12	Eşitlik (assignment)	`= += -= *= /= %= <<= >>= &= ^= |=`	Sağdan sola

Tablo 14. İşleçlerin İşlem Öncelikleri

Aşağıda işleçlerin önceliklerine ilişkin örnek programı lütfen inceleyiniz.

```
int main() {
   int a=10, b=5;
   a=a-b+2; // Öncelik Sırası +, -, = İşlem sonunda a=7
   a= a/b+2; // Öncelik Sırası /, +, = İşlem Sonunda a=7/5+2=3
   a= 2*10-20/2+3+(12-10); // Öncelik Sırası: (-) * / - + +
   /*
    İfadenin çözüm sırası:
    > 2*10-20/2+3+2
    > 20-20/2+3+2
    > 20-10+3+2
    > 10+3+2
    > 13+2
    > 15
   */
}
```

Kayan Noktalı Sayı Hesapları

Kayan noktalı sayıların IEEE-754 standardında ikili tabanda tutulduğu *Değişken* başlığında anlatılmıştı. Kayan noktalı sayılar olarak değişkenlere verdiğimiz onluk tabanda değerler aslında arka planda ikilik tabanda tutulur ve bu durum yazdığımız kodlarda garip davranışların ortaya çıkmasına sebep olur. Buna örnek olarak; hemen hemen her programcının yaptığı ilk hata, aşağıdaki kodun amaçlandığı gibi çalışacağını varsaymaktır;

```
float total = 0;
for(float a = 0; a != 2; a += 0.01f) {
    total += a;
}
```

Acemi programcı bunun 0, 0.01, 0.02, 0.03, gibi artarak, 1.97, 1.98, 1.99 aralığındaki her bir sayıyı toplayacağını ve 199 sonucunu, yani matematiksel olarak doğru cevabı vereceğini varsayar. Bu kodda doğru yürümeyen iki şey olur: Birincisi yazıldığı haliyle program asla sonuçlanmaz. `a` asla 2'ye eşit olmaz ve döngü asla sonlanmaz. İkincisi ise döngü mantığını `a < 2` şartını kontrol edecek şekilde

yeniden yazarsak, döngü sonlanır, ancak toplam 199'dan farklı bir şey olur. IEEE754 uyumlu makinelerde, genellikle bunun yerine yaklaşık 201'e ulaşır. Bunun olmasının nedeni, Kayan noktalı sayılarının onluk tabanda kendilerine atanan değerlerin, ikilik tabanda yaklaşık değerlerini temsil etmesidir. Bu duruma aşağıdaki klasik örnek verilir;

```
#include <iostream>
int main()
{
    double a = 0.1;
    double b = 0.2;
    double c = 0.3;
    if(a + b == c)
        cout<<"Bu satır İcra Edilmez!\n" << endl; //IEEE754 standardında işlem yaparsak.
    else
        cout<< "Kayan Noktalı Sayılar İkili Sayı Sisteminde"+
               " Tutulduğundan  Yaklaşık Olarak Hesaplanır." << endl;
    return 0;
}
```

Programcı olarak gördüğümüz şey onluk tabanda yazılmış üç sayı olsa da derleyicinin (ve altta yatan donanımın) gördüğü şey ikili sayılardır. 0.1, 0.2 ve 0.3, ona mükemmel bölmeyi gerektirdiğinden (ki bu onluk sayı sisteminde oldukça kolaydır), ancak ikili sayı sisteminde imkansızdır (bu sayılar, onluk sayı sistemindeki 1/3 sayısı gibi 0.333333333333333… şeklinde kesin olmayan biçimde depolanması gerektiğindendir);

```
#include <iostream>
int main()
{
    /*64 bitlik kayan noktalı sayılar, tam sayı kısmı dahil olmak üzere
      53 basamaklı hassasiyete sahiptir */
    double a = 0011111110111001100110011001100110011001100110011001100110011010;
    //onluk sistemdeki 0.1 ikili sistemde kusurlu olarak saklanır
    double b = 0011111111001001100110011001100110011001100110011001100110011010;
    //onluk sistemdeki 0.2 ikili sistemde kusurlu olarak saklanır
    double c = 0011111111010011001100110011001100110011001100110011001100110011;
    //onluk sistemdeki 0.3 ikili sistemde kusurlu olarak saklanır
    double a + b = 0011111111010011001100110011001100110011001100110011001100110100;
    //c değişkeni ile a+b nin onluk sistemde 0,3'e tam olarak eşit olmadığını unutmayın!
    return 0;
}
```

BASİT GİRİŞ ÇIKIŞ İŞLEMLERİ

Modüler Programlama

C++ programlama dili, yapısal olmasının yanında aynı zamanda modüler bir programlama dilidir. Yapısal programlamanın çerçevesi *Yapısal Programlama* altında anlatılmıştır. Yapısal programlamada yazılan fonksiyonlar gruplar halinde başka kod dosyalarına konulur ve bun bileşenler gerektikçe projemize `#include` ön işlemci yönergeleriyle (preprocessor directive) dahil edilir.

Modüllere ayırmanın soyutlama (abstraction), değişim yönetimi (change management) ve yeniden kullanım (reusing) gibi üstünlükleri vardır. Modüllere ayrılan bir kodun bakımı da daha kolaydır. Ancak daha fazla fonksiyonun çağrılması, işlemciyi yorar. Çünkü her fonksiyon çağrısında işlemcinin mevcut durumu ve kaydediciler belleğe itilir ve fonksiyondan geri dönüldüğünde eski duruma dönmek içi bu veriler tekrar geri çekilir. İşlemciye yüklenen bu çağırma yükü (calling overhead) günümüz bilgisayarlarında oldukça ihmal edilebilir seviyededir. Ancak milisaniyeler bazında yapılması gereken bir iş varsa böyle zaman kritik işler gerçekleştirmek için bu durum göz önüne alınmalıdır.

C++ programlama dilinde en çok kullanacağımız modüllerden biri de giriş çıkış işlemlerinin (input output-IO) tanımlı olduğu `iostream` başlık (header) dosyasıdır. C dilinde `stdio.h` olarak verilen başlık dosyası yerine nesnelerle tasarlanmış `iostream` başlık dosyası kullanılır. Bu modülü kodumuza dahil etmek (include) için kodu yazdığımız metin editörünüzde kaynak kodun başına `#include` ön işlemci yönergesini (preprocessor directive) aşağıdaki şekilde ekleriz.

```
#include <iostream>
```

Böylece konsola (konsol kelimesi UNIX/Linux işletim sitemindeki terminal ya da Windows işletim sistemindeki komut satırını ifade eder) bir şey yazmak için cin veya klavyeden bir şey okumak ve bir değişkene atamak için ise `cout` nesnelerini kullanabilir hale geliriz.

C programlama dilinde her işletim siteminde çalışan standart birçok başlık dosyası bulunmaktadır. Ancak bunların dışında çalıştığı işletim sistemine özel olan başlık dosyaları da bulunmaktadır;

Başlık Dosyası	Açıklama
`<iostream>`	`std::cout`, `std::cin` nesneleri kullanılarak, konsol ve klavye gibi akış (stream) nesneleri ile giriş ve çıkış işlemleri için nesne ve `std::endl` gibi
`<cmath>`	`sqrt()` , `log2()` , `pow()` gibi matematiksel işlemleri yapmak için kullanılan sınıfı içerir.
`<cstdlib>`	`malloc()` ve `free()` gibi bellek tahsisi ve sistemle ilgili `exit()` ve `rand()` gibi işlevleri içerir.
`<vector>`	Dinamik diziler (vektörler) için konteyner (container) sınıf tanımlarını içerir.
`<string>`	Dizgi işleme için sınıf ve işlevler sağlar `std::string`
`<iomanip>`	Giriş çıkış işlemlerinde değişkenlerdeki ondalık basamak sayısını sınırlamak için `set()` ve `setprecision()` fonksiyonlarına erişmek için kullanılır.
`<cerrno>`	`errno()`, `strerror()`, `perror()` gibi istisna işleme işlemlerini gerçekleştirmek için kullanılır.
`<time>`	`setdate()` ve `getdate()` gibi `date()` ve `time()` ile ilgili fonksiyonlarla işlem yapmak için kullanılır. Ayrıca sistem tarihini değiştirmek ve CPU zamanını almak için de kullanılır.

Tablo 15. En çok kullanılan başlık dosyaları

Konsolda Veri Okuma ve Yazma

C++ dilinde `std::cin` nesnesi, varsayılan olarak klavye ile ilişkilendirilen standart giriş akışı (input stream) olan `stdin` akışından gelen girdiyi kabul etmek için kullanılan `istream` sınıfında yer alır. Akışları, C dilindeki dosyalar olarak düşünebiliriz.

Girdi olarak kullanılan akışlarda, akıştan veri çıkarma işleci (stream extraction operator) (`>>`) ile verileri akıştan çıkarmak ve verilen değişkene eklemek için `cin` nesnesi ile birlikte kullanılır.

```
#include <iostream>
int main() {
   int yas;
   float agirlik;
   std::cin >> yas;
   std::cin >> agirlik;
}
```

Benzer şekilde C++ dilinde `std::cout` nesnesi, varsayılan olarak konsola ile ilişkilendirilen standart çıkış akışı (output stream) olan `stdout` akışına gönderilen çıktıyı kabul etmek için kullanılan `ostream` sınıfında yer alır. Çıktı olarak kullanılan akışlarda, akışa veri ekleme işleci (stream insertion operator) (`<<`) ile verileri verilen değişkeni kullanarak akışa yazmak için `cout` nesnesi ile birlikte kullanılır.

```
#include <iostream>
int main() {
   int yas;
   float agirlik;
   std::cout << "Yaş Giriniz:";
   std::cin >> yas;
   std::cout << "Ağırlık Giriniz:";
   std::cin >> agirlik;
   std::cout << "Girilen Yaş:"<< yas <<" ve Ağırlık:" <<agirlik;
}
/* Program çalıştırıldığında:
Yaş Giriniz:12
Ağırlık Giriniz:75.6
Girilen Yaş:12 ve Ağırlık:75.6

...Program finished with exit code 0
*/
```

Dahil edilen `iostream` başlığı birçok nesne ve bildirim içerir. Bunlardan standart giriş nesnesi olan `cin` ve standart çıkış nesnesi olan `cout` nesnesi `std` adlı aynı isim uzayında (namespace) yer alırlar. Programda bu isim uzayını hep yazmamak için `using namespace std;` talimatı yazılabilir. Aksi halde `cout` veya `cin` nesnesine ulaşmak için yukarıdaki örnekte olduğu gibi iki tane iki nokta üst üste karakteri olan kapsam çözümleme işleci (scope resolution operator) kullanılır. İsim uzayları sonradan anlatılacaktır. Bu durumda aynı kod aşağıdaki gibi kısaltılabilir;

```
#include <iostream>
using namespace std;
int main() {
   int yas;
   float agirlik;
   cout << "Yaş Giriniz:";
   cin >> yas;
   cout << "Ağırlık Giriniz:";
   cin >> agirlik;
   cout << "Girilen Yaş:"<< yas <<" ve Ağırlık:" <<agirlik;
}
```

Benzer şekilde yarıçapı gerçek sayı olarak klavyeden alınan bir çemberin ve çevresini hesaplayan program aşağıda verilmiştir.

```
#include <iostream>
#define PI 3.1415
using namespace std;
int main() {
   float cemberinCevresi;
   cout << "Çemberin Çevresini Giriniz:";
```

```
   cin >> cemberinCevresi;
   float cemberinYariCapi=cemberinCevresi/2/PI;
   cout << "Çevresini Girdiğiniz Çemberin Yarıçapı:"<<cemberinYariCapi;
}
```

Uluslararası Metinler

C++ dilinde `wchar_t`, desteklenen en geniş karakter setinin (UTF-16) tüm karakterlerini temsil edebilecek kadar büyük bir tamsayı türüdür. Normalde, `char` karakter türünden daha büyük bir boyuta sahip olduğundan, ASCII 255 üzerinde karakterleri depolamanız gerektiğinde kullanılır.

```
#include <iostream>
using namespace std;
int main()
{
    const wchar_t message_chinese[] = L"你好，世界";// Chinese for "hello, world"
    const wchar_t message_hebrew[] = L" שלום עולם "; //Hebrew for "hello, world"
    const wchar_t message_russian[] = L"Привет мир"; //Russian for "hello, world"
    const wchar_t message_turkish[] = L"Merhaba Dünya"; //Türkçe "Merhaba Dünya"
    const wchar_t message_turkish2[] = L"ıİüÜğĞiİşŞçÇöÖ"; //Türkçe "Diğer Karakterler
    std::locale::global(std::locale(""));
    std::wcout.imbue(std::locale());
    setlocale(LC_ALL, "Chinese");
    wcout << message_chinese << endl;
    setlocale(LC_ALL, "Hebrew");
    wcout << message_hebrew << endl;
    setlocale(LC_ALL, "Russian");
    wcout << message_russian << endl;
    setlocale(LC_ALL, "Turkish");
    wcout << message_turkish << endl;
    wcout << message_turkish2 << endl;
}
/*Program Çıktısı:
你好，世界
 שלום עולם
Привет мир
Merhaba Dünya
ıİüÜğĞiİşŞçÇöÖ
*/
```

std::iomanip

Bu başlık `std::cin` ve `std::cout` ile ileride göreceğimiz akış nesnelerinde girdi ve çıktıyı biçimlendirmek için kullanılır; Aşağıdaki örnekte `std::setprecision` ile çıktıya gönderilecek hane sayısı belirleniyor;

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <limits>
int main()
{
    const long double pi = std::acos(-1.L);
    std::cout << "default precision (6): " << pi << '\n'
    << "std::precision(10): " << std::setprecision(10) << pi << '\n'
    << "max precision: "
    << std::setprecision(std::numeric_limits<long double>::digits10 + 1)
    << pi << '\n';
}
/*Program Çıktısı:
default precision (6): 3.14159
std::precision(10): 3.141592654
```

```
max precision: 3.141592653589793239
*/
```

Aşağıda `std::setw` ile yazdırılacak genişlik ve `std::setfill` ile doldurulacak karakter örneği verilmiştir;

```
#include <iostream>
#include <iomanip>
int main()
{
    int val = 10;
    std::cout << val << std::endl;
    std::cout << std::setw(10) << val << std::endl;
    std::cout << "default fill: " << std::setw(10) << 42 << '\n'
    << "setfill('*'): " << std::setfill('*')
    << std::setw(10) << 42 << '\n';
}
/* Program Çıktısı:
10
        10
default fill:         42
setfill('*'): ********42
*/
```

Giriş çıkış işlemlerinde `out<<setiosflags(mask)` veya `in>>setiosflags(mask)` kullanıldığında akışın tüm biçim bayraklarını maskenin belirttiği şekilde dışarı veya içeri ayarlar. Tüm `std::ios_base::fmtflags` listesi:

- `dec` - tamsayı G/Ç için ondalık taban kullan
- `oct` - tamsayı G/Ç için sekizlik taban kullan
- `hex` - tamsayı G/Ç için onaltılık taban kullan
- `basefield` - `dec`|`oct`|`hex`|`0` maskeleme işlemleri için kullanışlıdır
- `left` - sol ayarlama (sağa dolgu karakterleri ekler)
- `right` - sağ ayarlama (sola dolgu karakterleri ekler)
- `internal` - dahili ayarlama (dahili olarak belirlenmiş noktaya dolgu karakterleri ekler)
- `adjustfield` - `left`|`right`|`internal`. Maskeleme işlemleri için kullanışlıdır
- `scientific` - bilimsel gösterim veya `fixed` ile birleştirildiğinde `hex` gösterimi kullanarak kayan nokta türleri oluşturur
- `fixed` - `fixed` gösterimi veya `scientific` ile birleştirildiğinde `hex` gösterimi kullanarak kayan nokta türleri oluşturur
- `floatfield` - `scientific`|`fixed`|(`scientific`|`fixed`)|`0`. Maskeleme işlemleri için kullanışlıdır
- `boolalpha` – alfa sayısal biçimde `bool` türünü ekler ve çıkarır
- `showbase` - tamsayı çıktısı için sayısal tabanı belirten bir önek oluşturur, para birimi göstergesini gerektirir
- `showpoint` - kayan nokta sayı çıktısı için koşulsuz olarak ondalık nokta karakteri oluşturur
- `showpos` - negatif olmayan sayısal çıktı için + karakteri oluşturur
- `skipws` - belirli girdi işlemlerinden önce öndeki boşlukları atlar
- `unitbuf` her çıktı işleminden sonra çıktıyı temizler
- `uppercase` - belirli çıktı çıktılarında belirli küçük harfleri büyük harf eşdeğerleriyle değiştirir

```
#include <iostream>
#include <string>
#include<iomanip>
int main()
{
    int l_iTemp = 47;
    std::cout<< std::resetiosflags(std::ios_base::basefield);
    std::cout<<std::setiosflags( std::ios_base::oct)<<l_iTemp<<std::endl; //57
    std::cout<< std::resetiosflags(std::ios_base::basefield);
    std::cout<<std::setiosflags( std::ios_base::hex)<<l_iTemp<<std::endl; //2f
    std::cout<<std::setiosflags( std::ios_base::uppercase)<<l_iTemp<<std::endl; //2F
```

```
    std::cout<<std::setfill('0')<<std::setw(12);
    std::cout<<std::resetiosflags(std::ios_base::uppercase);
    std::cout<<std::setiosflags( std::ios_base::right)<<l_iTemp<<std::endl; //00000000002f
    std::cout<<std::resetiosflags(std::ios_base::basefield|std::ios_base::adjustfield);
    std::cout<<std::setfill('.')<<std::setw(10);
    std::cout<<std::setiosflags( std::ios_base::left)<<l_iTemp<<std::endl; //47........
    std::cout<<std::resetiosflags(std::ios_base::adjustfield)<<std::setfill('#');
    std::cout<<std::setiosflags(std::ios_base::internal|std::ios_base::showpos);
    std::cout<<std::setw(10)<<l_iTemp<<std::endl; //+#######47
    double l_dTemp = -1.2;
    double pi = 3.14159265359;
    std::cout<<pi<<" "<<l_dTemp<<std::endl; //+3.14159 -1.2
    std::cout<<std::setiosflags(std::ios_base::showpoint)<<l_dTemp<<std::endl; //-1.20000
    std::cout<<setiosflags(std::ios_base::scientific)<<pi<<std::endl; //+3.141593e+00
    std::cout<<std::resetiosflags(std::ios_base::floatfield);
    std::cout<<setiosflags(std::ios_base::fixed)<<pi<<std::endl; //+3.141593
    bool b = true;
    std::cout<<std::setiosflags(std::ios_base::unitbuf|std::ios_base::boolalpha)<<b; //true
    return 0;
}
```

KONTROL İŞLEMLERİ

Ardışık İşlem ve Kontrol İşlemleri

Yapısal Programlama başlığı altında programın ana fonksiyondan başlayacağı ve programlamanın ise birbirini çağıran fonksiyonlarla yapıldığı anlatılmıştı. Yapısal programlamada, ana fonksiyon dahil tüm fonksiyonlarda ilk önce işlenecek verileri taşıyan veri yapıları tanımlanır ve ardından bu verileri işleyen kontrol yapılarına ilişkin talimatlar yazılır.

Şu ana karar örneğini verdiğimiz kodlarda veri yapısı olarak sadece değişkenler kullanılmıştır. Sonrasında ise giriş çıkış işlemleri talimatlar içeren programlar yazılmıştır. Programın icrası ilk talimattan başlar ve sırasıyla program bitene kadar devam eder. İşte programın icrasını değiştirmeyen bu tür talimatlara ardışık işlem (sequential operation) adı verilir. Ardışık işlemler aşağıdaki üç tür talimattan (statement) oluşur.

1. Değişkenlerin kimliklendirildiği değişken tanımlamaları (identifier definition):

```
int yariCap=3;
const float PI=3.14;
float daireninAlani,daireninCevresi;
```

2. İfadelerden (expression) yani sabit, değişken ve operatör içeren sözdizimleri:

```
daireninAlani=PI*yariCap*yariCap;
float daireninCevresi=2.0*PI*yariCap;
```

3. Klavyeden veri okume ve konsola veri yazma gibi giriş çıkış işlemleri (input output operation):

```
printf("Kapasite Oranını (0.00-1.00) Giriniz:");
scanf("%f",&kapasiteOrani);
```

Kontrol işlemleri (control operation) ise programın icra sırasını değiştiren kontrol yapıları olup bu talimatlar (statement) olup üç türü vardır;

1. Duruma göre seçimler (conditional choice): `if`, `if-else` ve `switch` talimatları.
2. İlişkisel döngü (relational loop): `while`, `do-while` ve `for` talimatları.
3. Dallanmalar (jump): `continue`, `break`, `goto` ve `return` talimatları.

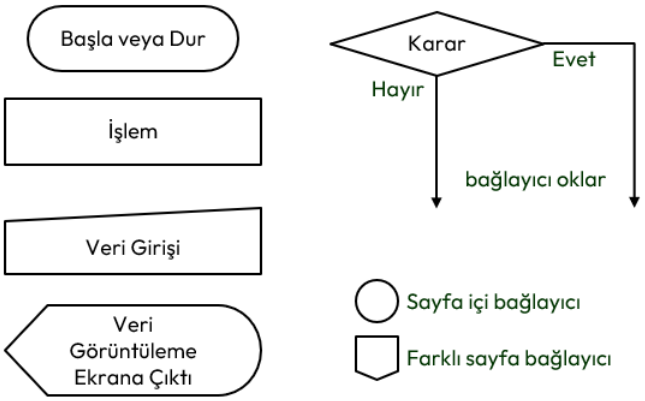
Kontrol işlemlerinde; talimat olarak yazılmış mantıksal satırlar (logical sequence) ile fiziksel olarak icra edilen satırlar (physical sequence) birbirinden farklıdır.

Akış Diyagramları

Kontrol işlemi içeren programın icrasını anlamak için akış diyagramlarını da anlamak gerekir. Akış diyagramı, adımların grafiksel gösterimleridir. Algoritmaları ve programlama mantığını temsil etmek için bir araç olarak bilgisayar biliminden ortaya çıkmıştır. Ancak diğer tüm işlem türlerinde kullanılmak üzere genişletilmiştir.

Günümüzde akış diyagramları, bilgileri göstermede ve akıl yürütmeye yardımcı olmada son derece önemli bir rol oynamaktadır. Karmaşık süreçleri görselleştirmemize veya sorunların ve görevlerin yapısını açık hale getirmemize yardımcı olurlar. Bir akış diyagramı, uygulanacak bir süreci veya projeyi tanımlamak için de kullanılabilir. Akış diyagramı çizilirken aşağıdaki kurallara uyulur;

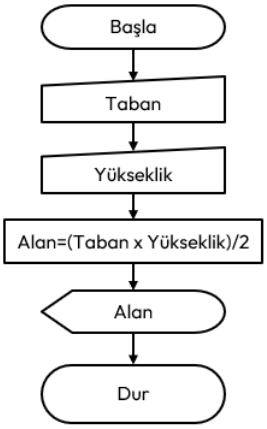
- Akış şemalarında tek bir başlangıç simgesi olmalıdır
- Bitiş simgesi birden çok olabilir.
- Karar simgesinin haricindeki simgelere her zaman tek giriş ve tek çıkış yolu bulunur.
- Bağlaç simgesi sayfanın dolmasından ötürü parçalanan akış şemasının öğelerini birleştirmede kullanılır.
- Simgeler birbirleri ile tek yönlü okla bağlanırlar.
- Okların yönü algoritmanın mantıksal işlem akışını tanımlar.



Şekil 4. Akış Diyagramları Genel Şekilleri

Yukarıda tüm programlama dillerinde akış diyagramları için ortak kullanılan temel şekiller verilmiştir.

Aşağıda taban ve yüksekliği klavyeden girilen bir üçgenin alanını hesaplamak için bir akış diyagramı örneği verilmiştir.



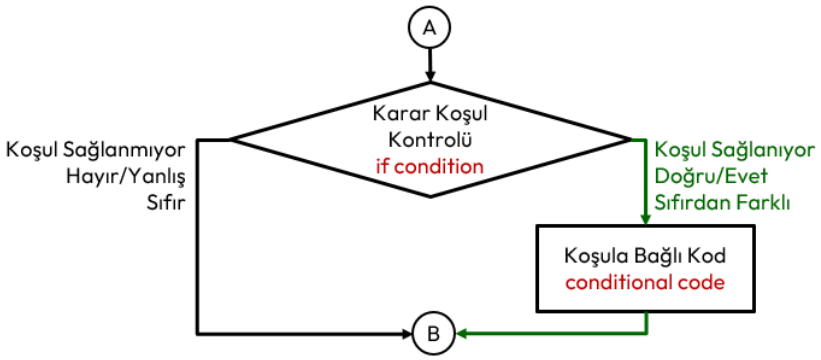
Şekil 5. Örnek Akış Diyagramı

Duruma Göre Seçimler

Duruma göre seçimler (conditional choice) programın akışını bir koşula göre değiştiren talimatlardır.

If Talimatı

If talimatı, karar vermeye (decision-making) ilişkin bir talimat olup bir koşula bağlı olarak programın icrasını değiştirir. Koşul (condition) ifadesi `true` ise koşul kodu (conditional code) icra edilir değilse icra edilmez. Koşul ifadesi (expression) test edilir ve sıfırdan farklı ise `true`, aksi halde `false` kabul edilir.



Şekil 6. If Talimatı İcra Akışı

Sözde kodu aşağıda verilmiştir;

```
if (koşul)
    koşul-kodu;
```

		yas	cinsiyet
	`#include <iostream>`		
	`using namespace std;`		
	`int main() {`		
1	`int yas=18;`	18	?
2	`char cinsiyet='K';`	18	'K'
3	`if (yas<30)`	18	'K'
4	`cout << "Genç";`	18	'K'
5	`if (cinsiyet=='E')`	18	'K'
	`cout << "Erkek";`	18	'K'
	`}`		

Tablo 16. If Talimatı İçeren Bir C Programı İcra Sırası

Programın icrası, `main` fonksiyonu içindeki ilk talimatla başlar ve solda verilen sırayla icra edilir. Sağda ise her icra sonrası değişkenlerin değerleri gösterilmiştir. Her talimatın icrasında neler olduğu aşağıda verilmiştir;

1. `yas` değişkenine `18` değeri atanır.
2. `cinsiyet` değişkenine `'K'` atanır.
3. `if` talimatındaki koşul (condition) test edilir. `yas<30` yani `18<30` testinde küçüktür işleci `true` verir. Bu nedenle izleyen koşul kodu (conditional code) icra edilir.
4. `Cout << "Genç";` koşul kodu olduğundan icra edilir.
5. `if` talimatındaki koşul test edilir. `cinsiyet=='E'` yani `'E'=='K'` testinde eşit mi işleci `false` verir. Dolayısıyla izleyen koşul kodu icra edilmez.

Bu durumda programın çalışması sonucu aşağıdaki çıktı elde edilir.

```
Genç

...Program finished with exit code 0
```

`if` talimatında koşul kodu (conditional code) her zaman ardışık işlem (sequential operation) olmaz. `if` gibi bir kontrol işlemi (control operation) de olabilir. Aşağıda kademeli (compound/cascade) if kullanımına ilişkin kod örneğinde ikinci `if`, birinci `if` talimatının koşul kodudur.

```
    if (yas<30)
        if (yas<7)
            cout << "Çoçuk ";
```

Koşul kodu (conditional code) birden fazla talimattan oluşacak ise kod bloğu `{ }` içine alınır. Aşağıda verilen kod örneğinde ilk `if` talimatında `yas<30` ile test edilen koşul doğrulandığında kod bloğunun içindeki tüm talimatlar icra edilir.

```
    if (yas<30) {
        if (yas<7)
           cout << "Çoçuk ";
        if (yas<18)
           cout << "Genç ";
        if (yas>60)
           printf("Yaşlı ");
    }
```

Bahsedilen iki durum iç içe de olabilir. Buna ilişkin kod örneği de aşağıda verilmiştir.

```
    if (cinsiyet=='E') {
        if (yas<7)
           cout << "Erkek Çocuk ";
        if (yas<18)
           cout << "Genç Delikanlı ";
        if (yas>60)
           cout << "Yaşlı Adam ";
    }
```

```
    if (cinsiyet=='K') {
        if (yas<3)
           cout << "Kız Bebek ";
        if (yas<7)
           cout << "Kız Çoçuk ";
        if (yas<18)
           cout << "Genç Kız ";
        if (yas>60)
           cout << "Yaşlı Kadın ";
    }
```

If talimatında koşul (condition) her zaman tek bir test ifadesinden oluşmayabilir. Bahsedilen duruma ilişkin kod örneği de aşağıda verilmiştir.

```
    if ((cinsiyet=='E') && (yas<18))
       cout << "Genç Delikanlı ";
    if ((cinsiyet=='K') && (yas>60))
       cout << "Yaşlı Kadın ";
    // if (10>rakam>5) // HATA!. Böyle bir şart yazılamaz.
       cout << "rakam 10 ile 5 arasında ";
    // if (rakam>10>rakam>5) // HATA!. Böyle bir şart yazılamaz.
       cout << "rakam 10 ile 5 arasında ";
```

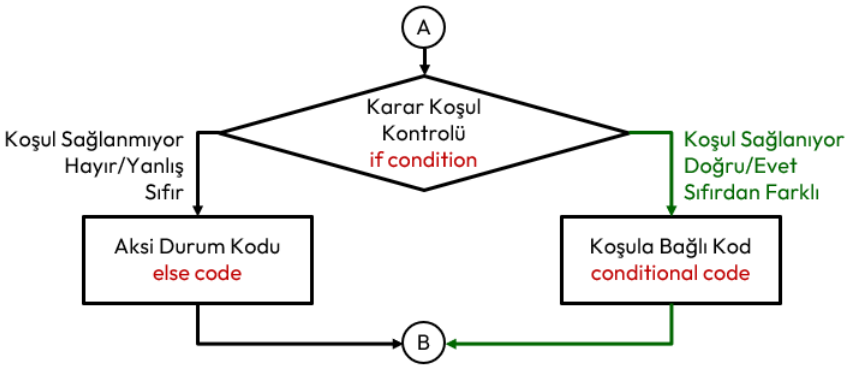
Bazen programcı tarafından aşağıdaki gibi `if` talimatları yazabilir.

```
    if (1)
      cout << "Bu metin konsola her zaman yazılır.";
    if (0)
      cout << "Bu metin konsola hiçbir zaman yazılmaz!";
```

If-Else Talimatı

If-Else Talimatı, bir başka karar verme (decision-making) talimatı olup bir koşula bağlı olarak programın icrasını değiştirir. If talimatına benzer şekilde koşul (condition) ifadesi `true` ise koşul kodu (conditional code) icra edilir, değilse aksi durum kodu (else code) icra edilir.

```
if (koşul)
    koşul-kodu;
else
    aksi-durum-kodu;
```



Şekil 7. If-Else Talimatı Sözde Kodu ve İcra Akışı

		yas	cinsiyet
	`#include <iostream>`		
	`using namespace std;`		
	`int main() {`		`?`
1	`int yas=65;`	65	`?`
2	`char cinsiyet='K';`	65	`'K'`
3	`if (yas<50)`	65	`'K'`
	`cout << "Genç ";`	65	`'K'`
4	`else`	65	`'K'`
5	`cout << "Yaşlı ";`	65	`'K'`

	}		

Tablo 17. If-Else Talimatı İçeren Bir C Programı İcra Sırası

Programın icrası, `main` fonksiyonu içindeki ilk talimatla başlar ve solda verilen sırayla icra edilir. Sağda ise her icra sonrası değişkenlerin değerleri gösterilmiştir. Her talimatın icrasında neler olduğu aşağıda verilmiştir;

1. `yas` değişkenine `65` değeri atanır.
2. `cinsiyet` değişkenine `'K'` atanır.
3. if talimatındaki koşul (condition) test edilir. `yas<50` yani `65<50` testinde küçüktür işleci `false` verir. Test sonucu `false` olduğundan aksi durum kodu (else code) icra edilir.
4. Programın icrası `else` ifadesinden devam eder.
5. `Cout <<"Yaşlı ";` aksi durum kodu olduğundan icra edilir.

Bu durumda programın çalışması sonucu aşağıdaki çıktı elde edilir.

```
Yaşlı

...Program finished with exit code 0
```

If talimatında olduğu gibi bu talimatta da koşul kodu (conditional code) ya da aksi durum kodu (else code) birden fazla talimattan oluşacak ise kod bloğu `{ }` içine alınır. Buna ilişkin kod örneği aşağıda verilmiştir.

```
    if (cinsiyet=='E') {
        if (yas<7)
           cout << "Erkek Çocuk ";
        if (yas<18)
           cout << "Genç Delikanlı ";
        if (yas>60)
           cout << "Yaşlı Adam ";
    } else {
        if (yas<3)
           cout << "Kız Bebek ";
        if (yas<7)
           cout << "Kız Çoçuk ";
        if (yas<18)
           cout << "Genç Kız ";
        if (yas>60)
           cout << "Yaşlı Kadın ";
    }
```

Aşağıdaki program örneği incelendiğinde else, hangi if talimatına aittir? Böyle bir kod programcı tarafından yazılabilir ve derleyici buna hiçbir sıkıntı çıkarmaz.

```
    if (cinsiyet=='E') if (yas<18) cout << "Delikanlı"; else cout << "Adam";
```

Bu durumda kodu aşağıdaki gibi okunaklı hale getirdiğimizde ilk `if` talimatının koşul kodunun (conditional code) ikinci if-else talimatı olduğu görülmektedir. Bu problem sarkan else (dangling else) problemi olarak bilinir. Kod bloğu kullanılmadan yazılan bu tip kodlarda else her zaman kendinden bir önceki if talimatına aittir.

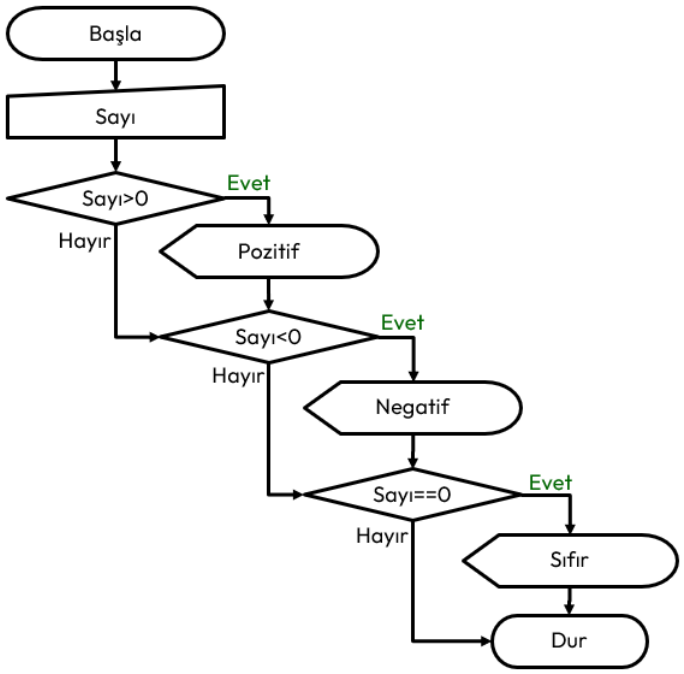
```
    if (cinsiyet=='E')
        if (yas<18)
            cout << "Delikanlı";
        else
            cout << "Adam";
```

Örnek olarak klavyeden girilen bir sayının sıfırdan küçük mü? Büyük mü? ya da sıfıra eşit mi? Olduğunu bulan bir C++ programı yazalım.

```
#include <iostream>
using namespace std;
```

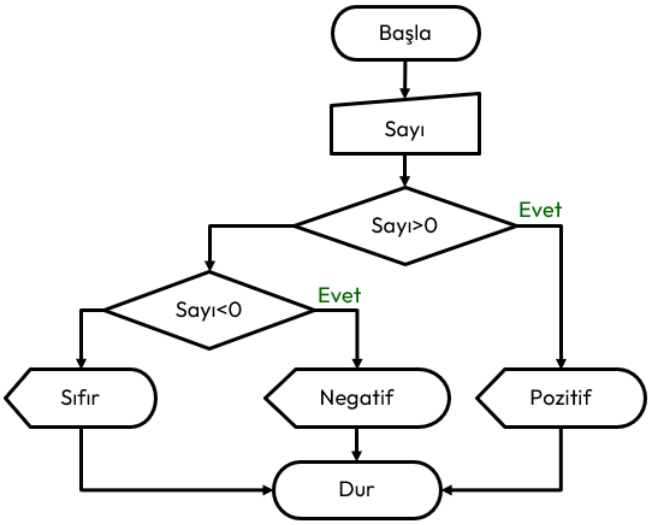
```
int main() {
    int sayi;
    cout << "Sayi Giriniz:";
    cin >> sayi;
    if (sayi>0) // sayının pozitif olup olmadığı test ediliyor.
        cout << "Pozitif"; // Burada sayının pozitif olduğu biliniyor.

    if (sayi<0) // sayının negatif olup olmadığı test ediliyor.
        cout << "Negatif"; // Burada sayının negatif olduğu biliniyor.

    if (sayi==0) // sayının sıfır olup olmadığı test ediliyor.
        cout << "Sıfır"; // Burada sayının sıfır olduğu biliniyor.
}
```

Programın icra akışı aşağıdaki şekilde olacaktır.



Şekil 8. Programın If Talimatlarıyla Yazılması Halinde İcra Akışı

Halbuki ilk if talimatında yapılan test doğru çıkarsa geri kalan testlerin yapılmasına gerek kalmaz. Benzer şekilde ilk iki test geçilirse üçüncü if talimatındaki teste gerek yoktur.



Şekil 9. Programın If-else Talimatlarıyla Yazılması Halinde İcra Akışı

Bu durumda aynı program if-else talimatlarıyla aşağıdaki şekilde yeniden yazabiliriz.

```
#include <iostream>
using namespace std;
int main() {
    int sayi;
    cout << "Sayi Giriniz:";
    cin >> sayi;
    if (sayi>0) { // sayının pozitif olup olmadığı test ediliyor.
        cout << "Pozitif"; // Burada sayının pozitif olduğu biliniyor.
    } else {  // Burada sayının pozitif olmadığı biliniyor.
        if (sayi<0) { // sayının negatif olup olmadığı test ediliyor.
            cout << "Negatif"; // Burada sayının negatif olduğu biliniyor.
        } else  { // Burada sayının pozitif ve negatif olmadığı test biliniyor.
            cout << "Sıfır"; // Burada sayının sıfır olduğu biliniyor.
        }
    }
}
```

Program incelendiğinde sayının pozitif girilmesi halinde sadece ilk if talimatındaki test geçecek ve else sonrası aksi durum koduna ilişkin blok icra edilmeyecektir. Sayının negatif girilmesi halinde ilk if talimatının else bloğuna girilecek ve blok içindeki ilk if talimatındaki test geçecek ve bu if talimatının else kısmı çalıştırılmayacaktır. Sayı sıfır girilmiş ise blok içindeki else kodu çalıştırılacaktır. Bunların dışında blok içinde tek bir if-else talimatı bulunmaktadır. Dolayısıyla blok içine almaya gerek de yoktur. Bu durumda kodun nihai hali aşağıda verilmiştir.

```
    if (sayi>0)
        cout << "Pozitif";
    else if (sayi<0)
        cout << "Negatif";
    else
        cout << "Sıfır";
```

Sarkan Else

Aşağıdaki program örneği incelendiğinde else, hangi if talimatına aittir? Böyle bir kod programcı tarafından yazılabilir ve derleyici buna hiçbir sıkıntı çıkarmaz.

```
if (cinsiyet=='E') if (yas<18) std::cout<<"Delikanlı"; else std::cout<<"Adam";
```

Bu durumda kodu aşağıdaki gibi okunaklı hale getirdiğimizde ilk `if` talimatının koşul kodunun (conditional code) ikinci if-else talimatı olduğu görülmektedir. Bu problem sarkan else (dangling else) problemi olarak bilinir. Kod bloğu kullanılmadan yazılan bu tip kodlarda else her zaman kendinden bir önceki if talimatına aittir.

```
    if (cinsiyet=='E')
        if (yas<18)
            std::cout<<"Delikanlı";
        else
            std::cout<<"Adam";
```

Switch Talimatı

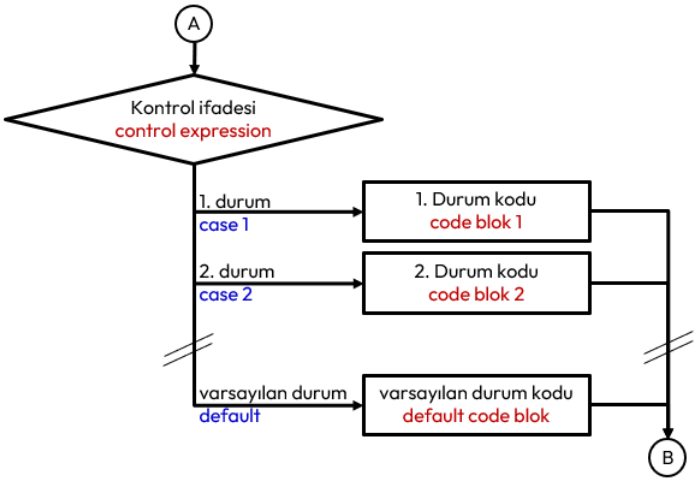
Switch talimatı, bir kontrol ifadesi (expression) sonucunda birden fazla alternatif arasında seçim yapılmasını sağlar. Sözde kodu aşağıda verilmiştir;

```
switch (kontrolifadesi) {
   case alternatif-1:
      alternatif-1-kodu;
      break;
   case alternatif-2:
      alternatif-2-kodu;
      break;
   // ...
```

```
   default:
      varsayılan-alternatif-kodu;
}
```

Switch talimatına ilişkin kurallar;

1. Kontrol ifadesi (control expression), sonucu tamsayı olan ifadedir ve bir kez değerlendirilir.
2. Switch bloğu zorunludur. Yani bloğu olmayan bir switch talimatı yazılamaz!
3. Her bir alternatif tamsayı değişmezdir (integer literal) izleyen ve iki nokta ile biten bir etiket (label) olarak tanımlanır. `case 1:`, `case 0:`, `case 'A':` gibi. Alternatif, bir değişken olamaz.
4. Blok için yazılan kod sonuna `break;` talimatı yazılarak switch bloğu dışına çıkılır. Zorunlu değildir, yazılmaz ise bir sonraki durum için yazılan kod icra edilir.
5. Tüm tamsayılar içerecek şekilde alternatiflerin tümü yazılmayabilir.
6. Blok sonunda yer alacak şekilde üzerinde yer alan alternatifler dışında kalan tüm alternatifler için `default:` etiketli alternatif yazılabilir. Zorunlu değildir.



Şekil 10. Switch Talimatı Sözde Kodu ve İcra Akışı

Aşağıda menü seçimi için yazılmış bir program örneği verilmiştir.

```
#include <iostream>
using namespace std;
int main() {
    int menu;
    cout << "Menü İçin Rakam Giriniz:";
    cin >> menu;

    switch(menu) {
    case 1:
        cout << "1 Numaralı Menüyü Seçtiniz.";
        cout << "Hamburger ve Ayran Hazırlanak.";
        break;
    case 2:
        cout << "2 Numaralı Menüyü Seçtiniz.";
        cout << "Patates Kızartması ve Kola Hazırlanacak.";
        break;
    case 3:
    case 4:
        cout << "3 veya 4 Numaralı Menüyü Seçtiniz.";
        break;
    default:
        cout << "1,2, 3 veya 4 Dışında Menü Seçtinniz.";
        cout << "Böyle bir Menümüz Yok!.";
    }
}
```

Programı aşağıdaki durumlar için çalıştırabiliriz;

1. Programda klavyeden `1` girildiğinde switch talimatında kontrol ifadesi test edilir ve sonucu `1` olduğuna karar verilir. Bu durumda `case 1:` etiketine gidilir ve bu alternatife ilişkin kodlar icra edilir. Yani ilk önce `cout << "1 Numaralı Menüyü Seçtiniz.";` icra edilir ve bir sonraki talimata geçilir. `cout << "Hamburger ve Ayran Hazırlanak.";` icra edilir ve bir sonraki talimata geçilir. `break;` talimatı bizi switch bloğunun sonundan dışına çıkarır.
2. Programda klavyeden `2` girildiğinde switch talimatında kontrol ifadesi test edilir ve sonucu `2` olduğuna karar verilir. Bu durumda `case 2:` etiketine gidilir ve bu alternatife ilişkin kodlar icra edilir. Yani ilk önce `cout << "2 Numaralı Menüyü Seçtiniz.";` icra edilir ve bir sonraki talimata geçilir. `cout << "Patates Kızartması ve Kola Hazırlanacak.";` icra edilir ve bir sonraki talimata geçilir. `break;` talimatı bizi switch bloğunun sonundan dışına çıkarır.
3. Programda klavyeden `3` girildiğinde switch talimatında kontrol ifadesi test edilir ve sonucu `3` olduğuna karar verilir. Bu durumda `case 3:` etiketine gidilir ve bu alternatife ilişkin icra edilecek kod yoktur. Sonrasında bulunan ilk talimat olan `cout << "3 veya 4 Numaralı Menüyü Seçtiniz.";` talimatıdır ve icra edilir ve bir sonraki talimata geçilir. `break;` talimatı bizi switch bloğunun sonundan dışına çıkarır.
4. Programda klavyeden `4` girildiğinde switch talimatında kontrol ifadesi test edilir ve sonucu `4` olduğuna karar verilir. Bu durumda `case 4:` etiketine gidilir ve bir üst maddede belirtilen icra gerçekleşir.
5. Programda klavyeden `-1` , `0`, `12` ve `300` gibi bir sayı girildiğinde switch talimatında kontrol ifadesi test edilir ve sonucu `1`,`2`,`3`,`4` alternatiflerinden biri olmadığına karar verilir. Bu durumda `default:` etiketine gidilir. Bu etiket sonrası ilk önce `cout << "1,2, 3 veya 4 Dışında Menü Seçtinniz.";` icra edilir ve bir sonraki talimata geçilir. `cout << "Böyle bir Menümüz Yok!.";` icra edilir. Zaten blok sonuna ulaşılmıştır. Bloktan çıkılır.

Aşağıda klavyeden girilen bir sayının 4 rakamına bölünmesinde kalanı gösteren bir program verilmiştir.

```
#include <iostream>
using namespace std;
int main() {
    int sayi;
    cout << "Bir Sayı Giriniz:";
    cin >> sayi;
    switch(sayi%4) { //konrol ifadesi bir işlem içerir
    case 3:
        cout << "Sayı 4 rakamına bölündüğünde kalan 3 dir.";
        break;
    case 2:
        cout << "Sayı 4 rakamına bölündüğünde kalan 2 dir.";
        break;
    case 1:
        cout << "Sayı 4 rakamına bölündüğünde kalan 1 dir.";
        break;
    default: // Başka alternatif yoktur.
        cout << "Sayı 4 Rakamına TAM Bölünür";
        break;
    }
}
```

Üçlü Şart İşleci

Daha önce *İşleçler* başlığı altında işlenenlere ek olarak, ardışık işlemlerde (sequential operation) if talimatlarına gerek kalmadan bir koşula göre ifadeler (expression) işlenecek ise üçlü şart işleci (conditional tenary operator) kullanılır. Aşağıda üçlü işlecin iki sözdizimi verilmiştir;

```
koşul ? doğruifadesi : yanlışifadesi;
değişken = koşul ? doğruifadesi : yanlışifadesi;
```

Aşağıda oy kullanma durumu bu işleçle işlenmiştir;

```
#include <iostream>
using namespace std;
int main() {
    int yas;
    cout << "Yaşınız?: ";
    cin >> yas;
    (yas >= 18) ? cout << "Oy Kullanabilirsin." : cout << "Oy Kullanamazsın!";
}
```

Aşağıda başka kullanımlarına ilişkin kod örneği verilmiştir;

```
    int sayi,tek,cift;
    cout << "Bir Sayı Giriniz: ";
    cin >> sayi;
    tek= (sayi%2) ? 1 : 0;
    cift= tek ? 0 : 1;
    int sonuc1=tek ? sayi+30 : sayi+40;
    int sonuc2=cift ? sayi*30 : sayi*40;
```

İlişkisel Döngüler

İşlemciler iyi bir sayıcıdır. CPU içindeki kaydediciler (registers) sayma işlevini birçok matematiksel işlemi yapmak için de kullanılır. Program akışında belli problemlerin çözümünde, gerçekleştirilen belli adımların tekrarlanması sonucu gidilir. Bu tip problemler şimdiye kadar olan yöntemlerle yapılırsa, tekrar tekrar aynı kodu yazmak zorunda kalırız. Yazdığımız kodları tekrar tekrar icra edebilmek için ilişkisel döngü (relational loop) talimatlarını kullanırız.

Konsola 10 defa ismimizi yazdıran bir program örneğini ele alalım. Burada konsolda satır başı (new line) yapmak için konsola `std::endl` gönderilir.

```
#include <iostream>
using namespace std;
int main() {
    int yas;
    cout << "ILHAN" << endl;
    cout << "ILHAN" << endl;
    cout << "ILHAN" << endl;
    cout << "ILHAN" << endl;
    cout << "ILHAN" << endl;
    cout << "ILHAN" << endl;
    cout << "ILHAN" << endl;
    cout << "ILHAN" << endl;
    cout << "ILHAN" << endl;
    cout << "ILHAN" << endl;
}
```

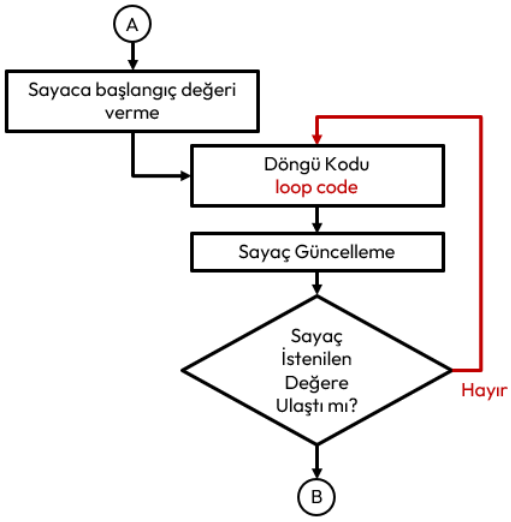
Program incelendiğinde aynı `cout` talimatının tekrar tekrar yazıldığını görürüz. Bu istenmeyen bir durumdur. Bunu yapmamak için sayaç olarak kullanılan bir değişken kullanırız.

Sayaç Kontrollü Döngüler

Belirlenen bir sayaç (counter) değişkeninin istenilen bir değere ulaşıp ulaşmadığı kontrol ederek kodun tekrar tekrar icra edilmesi sağlanır.

C ce C++ programlama dili ara seviye bir dil olduğundan `goto` talimatı desteklenir. 1968 Yılında *Edsger W. Dijkstra* GOTO/JUMP TO ifadelerini zararlı olarak ilan edilmiştir[15]. GOTO kullanmamak için;while, do-while ve for kontrol talimatları yapısal programlamaya eklenmiştir.

[15] https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf



Şekil 11. Sayaç Kontrollü Döngü İcra Akışı

Yapısal programlama öncesinde bu döngü `goto` talimatıyla sağlanır. İcra sırasını etiketlenen bir yer olarak değiştirmek için `goto` talimatı kullanılır. Etiketlere kimlik verilirken yine değişken kimliklendirme (identifier definition) kuralları uygulanır.

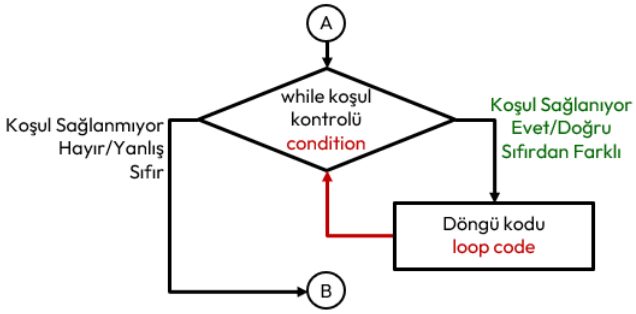
```
#include <iostream>
using namespace std;
int main() {
    int sayac=0; //Sayaca İlk Değer 0 Olarak Verildi
    Etiket: //Tekrar Edilecek Kodun Başı ETİKETLENİR. Etilet kimliği "Etiket"
        cout << "ILHAN" << endl;
        sayac++; //Sayaç Güncelleme
    if (sayac<10) //Sayaç Kontrolü
        goto Etiket; // İstenilen değere ulaşılmamış ise etikete git.
}
```

Yukarıdaki örnek incelendiğinde işaretli döngü bloğu üç adet talimat içeren mantıksal satır (logical sequence) olmasına karşın 30 adet fiziksel satır (physical sequence) icra edilmiştir.

While Talimatı

While döngüsü, koşul (condition) `true` olduğu sürece tekrarlanacak kodu (loop code) icra eder. Tekrarlanacak kod, tek bir talimat (statement) olabileceği gibi bir kod bloğu da olabilir. Sözde kodu aşağıda verilmiştir;

```
while (koşul)
    döngü-kodu;
```



Şekil 12. While Talimatı İcra Akışı

Sayaç Kontrollü Döngüler başlığında verilen döngü, While talimatı ile aşağıdaki şekilde yazılabilir.

```
#include <iostream>
using namespace std;
int main() {
    int sayac=0; //Sayaca İlk Değer 0 Olarak Verildi
```

```
    while (sayac<10) // While koşul kontrolü
    {  // döngü bloğu
        cout << "ILHAN" << endl;
        sayac=sayac+1; //Sayaç Güncelleme
    }
}
```

Sayaç güncelleme ifadesi koşul ifadesine eklenerek program daha da kısaltılabilir. Her koşul kontrolü sonrası sayaç artırılacaktır.

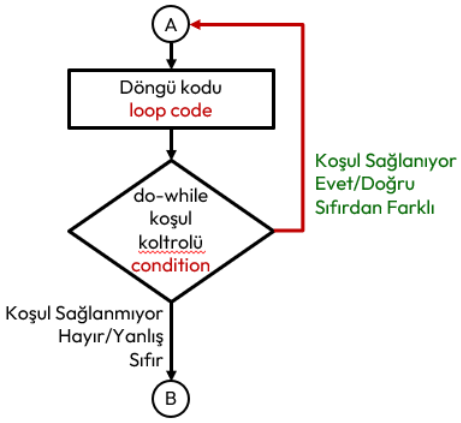
```
#include <iostream>
using namespace std;
int main() {
    int sayac=0; //Sayaca İlk Değer 0 Olarak Verildi
    while (sayac++<10) // hem sayaç güncelleme hem de koşul kontrolü
        cout << "ILHAN" << endl; // tek talimat olduğundan blok kullanılmadı
}
```

Yukarıdaki örnek incelendiğinde işaretli döngü bloğu bir adet talimat içeren mantıksal satır (logical sequence) içermesine rağmen 10 adet fiziksel satır (physical sequence) icra edilmiştir.

Do-While Talimatı

Do-While döngüsü, `do` ile `while` saklı kelimeleri arasındaki tekrarlanacak kodu (loop code) koşul (condition) `true` olduğu sürece tekrarlayarak icra eder. Tekrarlanacak kod tek bir talimat (statement) olabileceği gibi bir kod bloğu da olabilir. While döngüsünde koşul kontrolü en başta yapılır, bu döngüde ise döngü bloğu en az bir kez çalıştırıldıktan sonra koşul kontrolü yapılır. Sözde kodu aşağıda verilmiştir;

```
do
    döngü-kodu;
while (koşul);
```



Şekil 13. Do-While Talimatı ve İcra Akışı

Sayaç Kontrollü Döngüler başlığında verilen döngü, Do-While talimatı ile aşağıdaki şekilde yazılabilir.

```
#include <iostream>
using namespace std;
int main() {
    int sayac=0; //Sayaca İlk Değer 0 Olarak Verildi
    do {  // döngü bloğu
        cout << "ILHAN" << endl;
        sayac=sayac+1; //Sayaç Güncelleme
    } while (sayac<10); // Do-While koşul kontrolü
}
```

Sayaç güncelleme ifadesi koşul ifadesine eklenerek program daha da kısaltılabilir.

```
#include <iostream>
```

```
using namespace std;
int main() {
    int sayac=0; //Sayaca İlk Değer 0 Olarak Verildi
    do
        cout << "ILHAN" << endl;
    while (sayac++<10); // Do-While koşul kontrolü ve sayaç güncelleme
}
```

Yukarıdaki örnek incelendiğinde işaretli döngü bloğu 1 adet talimat içeren mantıksal satır (logical sequence) içermesine rağmen 10 adet fiziksel satır (physical sequence) icra edilmiştir.

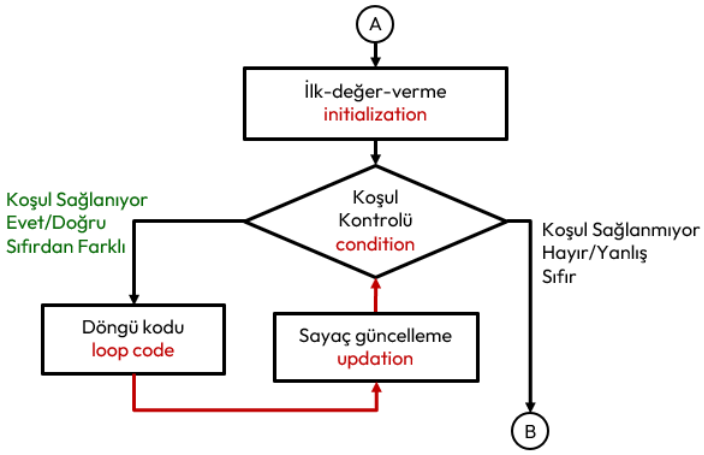
For Talimatı

For döngüsü özellikle tekrar edilen işlemlerin sayısı belli olduğunda kullanılan bir döngü yapısıdır. Sayaç kontrollü döngüler için en iyi seçimdir. Sözde kodu aşağıda verilmiştir.

```
for (ilk-değer-verme; koşul-kontrolü; sayaç-güncelleme)
    döngü-kodu;
```

For talimatı aşağıdaki şekilde icra edilir;

1. İlk değer verme ifadesi bir kez icra edilir.
2. Sonra koşul kontrolü yapılır. Eğer test sonucu `true` ise döngü kodu icrasına geçilir. Aksi halde döngüden çıkılır.
3. Döngü kodu icra edilir.
4. Döngü kodu icrası bitince sayaç güncellemeleri yapılır ve tekrar ikinci adımdaki koşul kontrolüne dönülür.



Şekil 14. For Talimatı İcra Akışı

Sayaç Kontrollü Döngüler başlığında verilen döngü, For talimatı ile aşağıdaki şekilde yazılabilir.

```
#include <iostream>
using namespace std;
int main() {
    int sayac;

    for (sayac=1; sayac<10; sayac++)
        cout << "ILHAN" << endl;
}
```

Buradaki döngü mantıksal olarak iki satır talimattan oluşmaktadır. Ancak fiziksel olarak 10 satır icra edilmiştir. Aşağıda konsola yazdığımız her bir satırın başına satır numarası koyan bir program gösterilmektedir. Ayrıca sayaç sadece for bloğunda kullanılacak ise ilk değer verme ifadesinde tanımlanabilir.

```
#include <iostream>
using namespace std;
int main() {
```

```
    for (int sayac=1; sayac<=10; sayac++)
        cout << sayac << ". ILHAN" << endl;
}
/* Program çalıştırıldığında:
01-ILHAN
02-ILHAN
03-ILHAN
04-ILHAN
05-ILHAN
06-ILHAN
07-ILHAN
08-ILHAN
09-ILHAN
10-ILHAN

...Program finished with exit code 0
*/
```

Yukarıdaki örnek incelendiğinde işaretli döngü bloğu bir adet talimat içeren mantıksal satır (logical sequence) içermesine rağmen 10 adet fiziksel satır (physical sequence) icra edilmiştir.

For döngüsünde bir sayaç ile çalışılabileceği gibi birden fazla sayaç ile de çalışılabilir. Hem ilk değer verme hem de sayaç güncellemede birden fazla sayaca ilişkin işlem yapılabilir. Bunun için virgül işleci (comma operator) kullanılır.

```
#include <iostream>
using namespace std;
int main() {
    int sayac1,sayac2;

    for (sayac1=0,sayac2=10; sayac1<10; sayac1++,sayac2--)
        cout << sayac1 << "-" <<sayac2 <<"- ILHAN" << endl;
}
/* Program çalıştırıldığında:
0-10- ILHAN
1-9- ILHAN
2-8- ILHAN
3-7- ILHAN
4-6- ILHAN
5-5- ILHAN
6-4- ILHAN
7-3- ILHAN
8-2- ILHAN
9-1- ILHAN

...Program finished with exit code 0
*/
```

Aşağıda klavyeden girilen iki sayı aralığında tek ve çift sayıların toplamını bulan ve ekrana yazan programı verilmiştir.

```
#include <iostream>
using namespace std;
int main() {
   int sayac, sayi1,sayi2;
   int tekToplam=0,ciftToplam=0;
   cout << "İki Sayı Giriniz: ";
   cin >> sayi1 >> sayi2;
   if (sayi2<sayi1) { //sayi2, sayi1 den büyük olmalı.
      int temp=sayi1; //küçük ise yer değiştiriyoruz.
      sayi1=sayi2;
      sayi2=temp;
```

```
   }
   for (sayac=sayi1; sayac<=sayi2; sayac=sayac+1) {
      if (sayac%2==1) //sayac tek mi?
         tekToplam+=sayac;
      else
         ciftToplam+=sayac;
   }
   cout << "Tek Toplam:" << tekToplam << " Çift Toplam:" << ciftToplam;
}
/* Program çalıştırıldığında:
İki Sayı Giriniz: 9 17
Tek Toplam: 65 Çift Toplam: 52

...Program finished with exit code 0
*/
```

İç İçe Döngü Kodu

Döngüler içinde yer alan döngü kodu (loop code) bir başka döngüyü içerebilir. Örneğin ekrana satır ve sütun içeren bir şekil oluşturmak istediğimizde iç içe döngü kullanırız. Aşağıda buna ilişkin bir örnek verilmiştir. İlk for döngüsünün tekrarlanan kodu ikinci for döngüsüdür.

```
#include <iostream>
using namespace std;
int main() {
  int satir,sutun;
  for (satir=0; satir<5; satir++)
  {
     for (sutun=0; sutun<4; sutun++)
        cout << "*";
     cout << endl;
  }
}
/* Program çalıştırıldığında:
****
****
****
****
****

...Program finished with exit code 0
*/
```

Bir başka örnek olarak elemanları, satır ve sütun çarpımları olan 4x5 boyutlarındaki matrisi ekrana yazdıran program;

```
#include <iostream>
using namespace std;
int main() {
  int satir,sutun;

  //Başlığı Yazdıran Kısım
  cout << "  Sutun: ";
  for(sutun=0; sutun<3;sutun++)
   cout << sutun+1 << " ";
  cout << endl;

  //Matrisi Yazdıran Kısım
  for (satir=0; satir<5; satir++) {
    cout << satir+1 <<".Satir:" << " ";
    for(sutun=0; sutun<3;sutun++)
      cout << sutun << " ";
```

```
    cout << endl;
  }
}
/* Program çalıştırıldığında:
  Sutun: 1 2 3
1.Satir: 0 1 2
2.Satir: 0 1 2
3.Satir: 0 1 2
4.Satir: 0 1 2
5.Satir: 0 1 2

...Program finished with exit code 0
*/
```

Gözcü Kontrollü Döngüler

Döngüler her zaman bir sayaca bağlı çalıştırılmaz. Bir döngüden çıkış koşulu döngü kodu içerisinde üretilen bir değere bağlı ise bu değere gözcü değeri (sentinel value) adı verilir. Buradaki gözcü değeri beklenen bir değer dışındaki bir değerdir. Buna örnek olarak klavyeden `0` girilene kadar girilen rakamları toplayan bir program verilmiştir.

```
#include <iostream>
using namespace std;
int main() {

   int sayi; /* okunan tamsayı */
   int toplam=0; /* girilen sayıların toplamı. başlangıçta 0*/
   do {
      cout << "Bir sayı giriniz (Bitirmek için 0): ";
      cin >> sayi;
      toplam+=sayi; // sayı 0 girilse bile toplam etkilenmez
   } while (sayi != 0); //sentinel value 0

   cout << "Girilen Sayıların Toplamı:" << toplam;
}
```

Aşağıda pozitif sayı girilmesini sağlayan bir kod örneği verilmiştir.

```
#include <iostream>
using namespace std;
int main() {
    int sayi,pozitifSayi=0;
    do { /* Pozitif girilmesini zorluyoruz */
        cout << "Lütfen pozitif sayı giriniz: ";
        cin >> sayi;
        if (sayi<=0)
            cout << "HATA:Pozitif Sayı GİRMEDİNİZ!" << endl;
    } while (sayi <= 0);
    pozitifSayi=sayi;
    cout <<"Girilen pozitif tamsayı: " << pozitifSayi;
}
```

Aynı örnek bir başka şekilde aşağıdaki gibi kodlanabilir.

```
#include <iostream>
using namespace std;
int main() {
    int sayi,pozitifSayi=0;
    cout << "Lütfen pozitif sayı giriniz: ";
    cin >> sayi;
    while (sayi <= 0) /* Pozitif girilmesini sağlıyoruz */
    {
          cout << "HATA:Negatif sayı giriniz!" << endl;
```

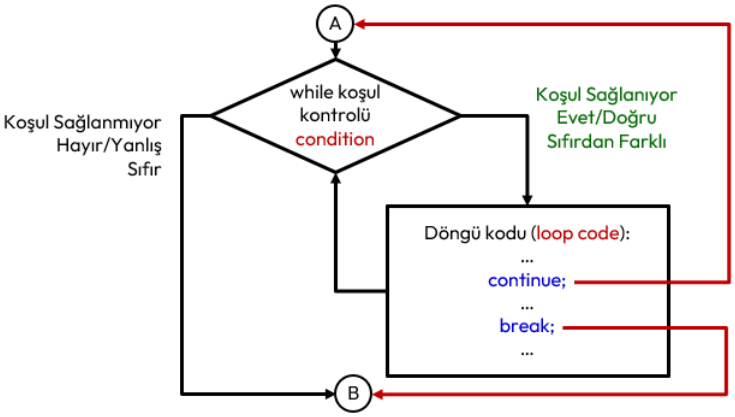
```
        cout << "Lütfen pozitif sayı giriniz: ";
        cin >> sayi;
    }
    pozitifSayi=sayi;
    cout <<"Girilen pozitif tamsayı: " << pozitifSayi;
}
```

Dallanmalar

Dallanmalar (jump) programın akışını bir başka talimata yönlendiren talimatlardır.

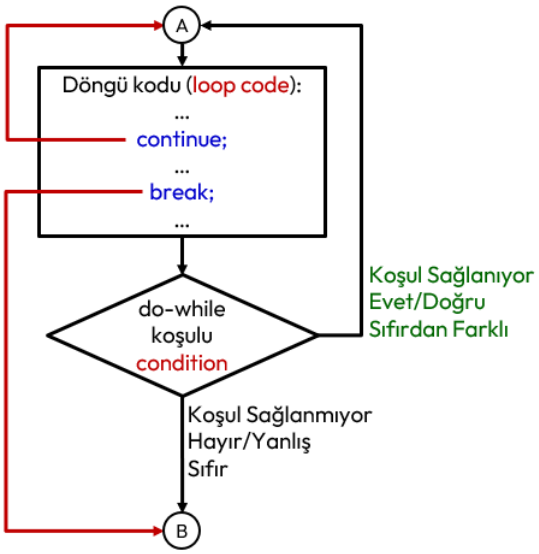
Continue ve Break Talimatları

Döngüdeki `continue` talimatı, geçerli yinelemeyi (iteration) sonlandırır ve bir sonraki yinelemeye devam edilir. Dolayısıyla sadece döngü kodları (loop code) içinde kullanılabilir. `break` Talimatı döngüyü sonlandırır ve program icrası döngü sonrası ilk talimattan devam eder. Bunu daha önce switch talimatında görmüştük. Bu talimat, aynı şekilde `while`, `do-while` ve `for` döngüleri ile kullanılabilir. Bu talimatlarının döngü kodlarında kullanılması halinde icra akışı aşağıdaki şekilde verilmiştir.



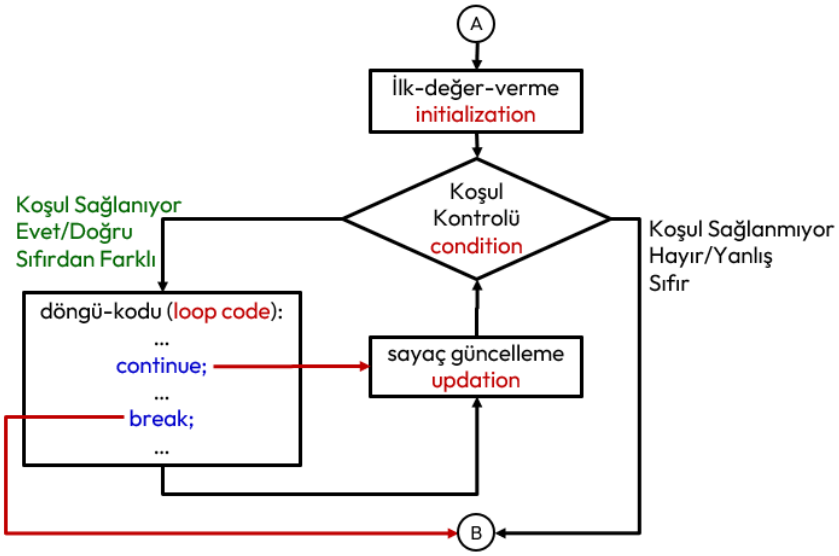
Şekil 15. While Döngü Kodunda Break ve Continue Talimatları İcra Akışı

While döngüsünde, döngü bloğunda `continute` kullanmadan önce koşulu değiştirecek bir talimat vermeliyiz. Aksi halde sonsuz döngüye girmiş oluruz. Do döngüsünde eğer `continue` talimatı bir şarta bağlanmalıdır. Yoksa sonsuz döngüye girilebilir.



Şekil 16. Do-While Döngü Kodunda Break ve Continue Talimatları İcra Akışı

Benzer durum for döngüsü için de geçerlidir. Ancak for döngüsünde `continue` talimatı sonrasında bir sonraki yineleme için önce sayaç güncelleme ifadeleri icra edilir ve sonra koşul testi yapılır. Bu durum göz önüne alınarak bu talimat kullanılmalıdır. Bu talimatlar döngü kodu içinde amacımıza uygun olarak birden çok kullanabiliriz.



Şekil 17. For Döngü Kodunda Break ve Continue Talimatları İcra Akışı

Aşağıda `0` ile `15` arasındaki sayıların beşe bölünenleri ve `10` dışındaki sayılar ekrana yazdıran bir program örneği verilmiştir.

```
#include <iostream>
using namespace std;
int main() {
    int i;
    for (i=0; i<=15; i=i+1)
    {
        if (i<7) //i'nin değeri 7 den küçük ise
            continue;  //bir sonraki yinelemeye (iteration) geç

        if (i==10) //i'nin değeri 10 ise
            continue;  //bir sonraki yinelemeye (iteration) geç
        if (i%5==0) // i'nin değeri 5 in katı ise
            continue;  //bir sonraki yinelemeye (iteration) geç
        cout << "i=" << i << endl;
   }
}
/*Program Çıktısı:
i = 7
i = 8
i = 9
i = 11
i = 12
i = 13
i = 14

...Program finished with exit code 0
*/
```

Yukarıdaki örnek incelendiğinde işaretli döngü bloğu üç adet talimat içeren mantıksal satır (logical sequence) olmasına karşın 42 adet fiziksel satır (physical sequence) icra edilmiştir. Bazı `continue` talimatları `cout` talimatının icrasını engellemiştir.

Aşağıda pozitif sayı girilmesini zorlayan sonsuz döngü içeren program verilmiştir. Programda döngü, pozitif sayı girildiğinde break talimatı ile kırılmaktadır.

```
#include <iostream>
using namespace std;
int main() {
    int sayi;
    int pozitifSayi=0;
```

```
    while (1) /* Sonsuz döngü */
    {
        cout << "Lütfen pozitif sayı giriniz: ";
        cin >> sayi;
        if (sayi<=0)
            cout << "HATA:Pozitif Sayı GİRMEDİNİZ!" << endl;
        else
            break; //döngüden çık
    }
    pozitifSayi=sayi;
    cout << "Girilen pozitif tamsayı: " << pozitifSayi;
}
```

Return Talimatı

Fonksiyonlarda çokça kullanacağımız `return` talimatı bir fonksiyonun üreteceği ya da belirlenen değeri geri döndürür. Kullanıldığı yerde belirlenen değer ile içinde bulunduğu fonksiyon bloğu dışına çıkılır. *FONKSİYONLAR* başlığı altında detaylı incelecek olup bir örneği *En Basit C++ Programı* başlığında bir örneği verilmiştir.

Goto Talimatı

Tanımlı bir etikete program akışını yönlendiren `goto` talimatıdır. *Sayaç Kontrollü Döngüler* başlığında örneği verilmiştir.

FONKSİYONLAR

Fonksiyon

Yapısal Programlama mantığında çözülecek problem genel olarak fonksiyonların birbirini çağırmasıyla yapıldığı anlatılmıştı. Yani kodlaması yapılacak işi birbirini çağıran fonksiyonlar yazarak ana fonksiyondan bu fonksiyonları çağırarak yaparız.

Ayrıca yazılacak programda birbiriyle ilgili fonksiyonlar bir araya toplanarak ayrı bir dosyada tutulur. Bu durum *Modüler Programlama* başlığı altında anlatılmıştı.

Bu bölüme kadar gördüğümüz kodlarda çözümü oluşturan tüm tasarım parçaları ana fonksiyon olan `main()` fonksiyonu içerisine çözülmüştür. Bir bilgisayar programı, genel olarak, tek başına tüm görevleri yerine getiren tek bir main fonksiyonundan oluşmaz. Bu durumda ana problemin büyüklüğüne göre ana fonksiyon bloğumuz uzar, okunabilirliği azalır, müdahale etmek zorlaşır. Bu tip büyük programları kendi içerisinde, her biri özel bir işi çözmek için tasarlanmış parçacıklarından oluşturmak daha mantıklı olacaktır. Yani böl ve yönet (divide and conquer) tekniği uygulanır. Çünkü büyük bir problemin toptan çözülmesi, problemin parçalar halinde çözülmesinden her zaman daha yorucu ve zordur.

Yazılım geliştirmede problemi büyük ana parçalara ayırırız. Daha sonra bu büyük parçaları daha küçük parçalara ayırarak çözeriz. Buna yukarıdan aşağıya (top down) yaklaşım adı verilir. C++ dilinde fonksiyonlar;

- Fonksiyonlar, belirli bir görevi gerçekleştiren bir kod bloğudur.
- Parametre alabilir, girdilerle ya da girdi olmadan bir şeyler yapar ve ardından cevabı verebilir. Kısaca bilgiyi fonksiyona argümanlar ile aktarırız.
- Her fonksiyonun bir kimliği vardır. (*Değişken Kimliklendirme Kuralları* burada da geçerlidir.)
- Bir fonksiyon program içerisinde istenildiği kadar farklı yerlerden ulaşılarak çalıştırılabilir ya da çağrılabilir. Bir başka deyişle tekrar kullanılabilir.
- Bir fonksiyon, çağıran koda bir değer geri döndürebilir (return).
- Bir fonksiyon, bir başka fonksiyondan çağrılabilir.

C++ dilinde iki tip fonksiyon bulunmaktadır; başlık dosyalarında bize sunulan hazır fonksiyonlar ve programcının tanımlayacağı kullanıcı tanımlı fonksiyonlar (user defined function).

Hazır Fonksiyonlar

C++ geliştiricileri, programcıların kullanmaları için, önceden yazılmış olan hazır fonksiyonlar hazırlamışlardır. Hazır fonksiyonlar teknik olarak C++ dilinin parçası değillerdir. Yalnızca standart hale getirilmişlerdir.

Programcı, kullanmak istediği hazır fonksiyonları; hangi modülde ise o modüle ilişkin başlık (header) dosyasını `#include` ön işlemci yönergesi (preprocessor directive) ile kendi projesine dahil eder. Bunu Modüler Programlama başlığı altında incelemiştik. Bir modül olan bu başlık dosyalarında;

- Hazır fonksiyonların sadece prototipleri (prototype) bulunur.
- Fonksiyonların kendileri yani derlenmiş halleri her işletim sistemi için ayrı oluşturulmuş kütüphane (library) dosyaları içerisinde bulunur.
- Derleme işlemi sırasında bağlayıcı (linker) tarafından bu fonksiyonlar icra edilebilir dosyaya (executable file) dahil edilir.

Cmath Başlık Dosyası

Matematiksel iş ve işlemleri gerçekleştirmek için standart hale getirilmiş bir başlık dosyasıdır. Bu başlık `abs`, `sin`, `cos`, `floor`, `sqrt` gibi birçok matematiksel fonksiyonu barındırır.

Örnek olarak kenarları verilen bir üçgenin alanını hesaplayan bir program aşağıda verilmiştir.

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    int kenar1,kenar2,kenar3;
    double alan,kenarlarToplamininYarisi;
    cout << "Üçgenin Kenarlarını Giriniz:";
    cin >> kenar1 >> kenar2 >> kenar3;
    if ((kenar1+kenar2 <= kenar3) ||
       (kenar2+kenar3 <= kenar1) ||
       (kenar1+kenar3 <= kenar2) ) {
        cout << "Böyle bir üçgen olamaz." << endl;
        return 1;
    }
    kenarlarToplamininYarisi=(kenar1+kenar2+kenar3)/2.0;
    alan=sqrt( kenarlarToplamininYarisi*
               (kenarlarToplamininYarisi-kenar1)*
               (kenarlarToplamininYarisi-kenar2)*
               (kenarlarToplamininYarisi-kenar3) );
    cout << "Üçgenin alanı: " << alan ;
}
```

Cstdlib Başlık Dosyası

Bu başlık içerisinde; birçok fonksiyonun yanında rastgele sayı üretme fonksiyonları `rand()` ve `srand()` bulunur.

Aşağıda bir zar için rastgele sayı üreten bir program verilmiştir.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
    int rastgeleZar;
    rastgeleZar=1 + rand() %6;
    /*
     std::rand() fonksiyonunun üreteceği sayıyı kalan işleci ile
     6 sayısına bmleriz ki 0 ile 5 arasında bir sayı elde edelim.
     Buna da 1 ekler isek zarın rakamları ortaya çıkar.
    */
   cout << "Atılan Zar:" << rastgeleZar;
}
```

Fakat programın her çalışmasında `srand::rand()` aynı başlangıç değerini kullandığından aynı rastgele değerleri üretir. Bu nedenle aynı zar rakamı gelmiş olur. Bunu değiştirmek için `std::srand()` fonksiyonu kullanılır.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
    unsigned randBaslangicDegeri;
    cout << "std::rand() için başlangıç değeri olabilecek bir sayı giriniz:";
    cin >> randBaslangicDegeri;
    srand(randBaslangicDegeri);
    /*
     std::rand() fonksiyonunun üreteceği sayı her zaman aynı başlangıç değeriyle
     başladığından her program çalıştığında üterilecek ilk rastgele sayı
     ve sonrasındaki sayılar hep olur.
     Bu hesaplamanın bir başka başlangıçla başlaması üterilecek ilk rastgele sayı
     ve sonraski üretilecekleri değiştirir. Bunu değiştirmenin yolu std:srand()
     fonksiyonunu kullanmaktır.
```

```
    */
    int rastgeleZar;
    rastgeleZar=1 + rand() %6;
    cout << "Atılan Zar:" << rastgeleZar;
}
```

Rastgele sayı her seferinde kullanıcıdan alınan sayıya göre hesaplandığından her seferinde farklı bir zar rakamı üretilmiş olacaktır. Ancak başlangıç değerinin her seferinde kullanıcıdan alınması bir başka problemdir. Bunun üstesinden gelmek için bilgisayarın saatini sayı olarak veren `ctime` başlık dosyasındaki `time()` fonksiyonu `nullptr` parametresiyle kullanılabilir.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main() {
    unsigned randBaslangicDegeri=time(nullptr);
    /*
     std:srand() fonksiyonu ile std::rand() fonksiyonu ilk değer verilir.
     Verilecek ilk değer std::time() fonksiyonuna nullptr verilerek sistem zamanından
     gelen değer kullanılır.
     Böylece her çalıştırmada farklı rastgele sayı üretilmesi sağlanacaktır.
    */
    srand(randBaslangicDegeri); // srand(time(nullptr)); olarak da yazılabilir.
    int rastgeleZar;
    rastgeleZar=1 + rand() %6;
    cout << "Atılan Zar:" << rastgeleZar;
}
```

Kullanıcı Tanımlı Fonksiyonlar

Programcılar kendi fonksiyonlarını oluşturarak kodlarını daha kullanışlı hale getirirler. Bu tür fonksiyonlara kullanıcı tanımlı fonksiyonlar (user defined function) denilir. Kullanıcı tanımlı fonksiyonu oluşturmak ve kullanmak için bu üç unsuru bilmemiz gerekir;

- Fonksiyon Bildirimi: Bir fonksiyonu çalıştıran koda, çağıran program adı verilir. Çağıran program kullanılacak fonksiyonu bilmeli. Bunun için çağıran program öncesinde fonksiyon bildirimi (function declaration) yapılmalı veya prototipi (function prototype) tanımlanmalıdır. Bildirim derleyiciye böyle bir fonksiyon var demenin yoludur.
- Fonksiyon Tanımı: Fonksiyon tanımı (function definition), bir fonksiyonun girdileri, yaptığı işlem ve döndüreceği değeri içerecek şekilde başlık ve gövdesinin kodlanmasından oluşur. Bildirimi yapılan fonksiyonun derleme yapılabilmesi için eksiksiz bir şekilde tamamlar. Bildirime benzer başlık ve tırnaklı parantez `{ }` kod bloğu ve arasında yer alan kodlardan oluşur. Bildirim yapılmadan da fonksiyon tanımlanabilir.
- Fonksiyon Çağırma: Fonksiyonu çağırmak (function call/invoke function) için fonksiyonun kimliğini ve ardından parantez içindeki argüman listesini yazmanız yeterlidir.

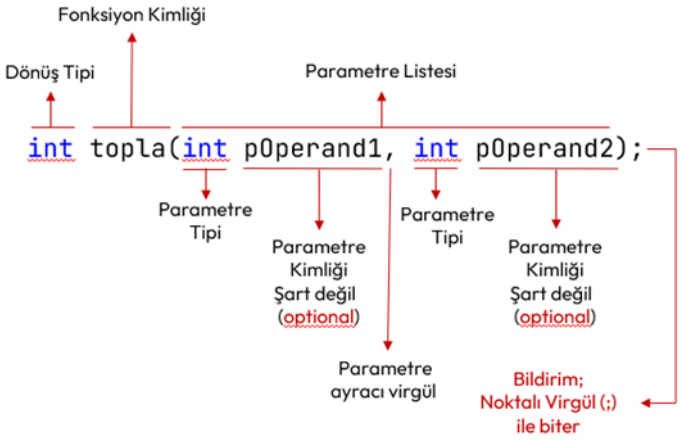
Fonksiyon Bildirimi Nasıl Yapılır?

Fonksiyon bildiriminde (function declaration); Dönüş tipi, işlevin döndürdüğü değerin veri tipidir. Bazı işlevler, herhangi bir değer döndürmeden istenen işlemleri gerçekleştirir. Bu durumda dönüş tipi `void` anahtar sözcüğüdür ve bu fonksiyonlar yordam (procedure) ya da alt rutin (subroutine) olarak adlandırılır.

```
/* Fonksiyon Bildirimleri */
int ikiSayiyiTopla(int,int); /* ikiSayiyiTopla kimlikli bir fonksiyon olduğu ve
                                bu fonksiyonun iki tamsayı parametre aldığı ve
                                bir tamsayı geri döndürdüğü derleyiciye bildiriliyor. */
void sayiyiKonsolaYaz(int);  /* sayiyiKonsolaYaz kimlikli bir fonksiyon olduğu ve
                                bu fonksiyonun bir tamsayı parametre aldığı ve
```

```
                                geri değer döndürmediği derleyiciye bildiriliyor. */
int konsoldanTamsayiOku();   /* konsoldanTamsayiOku kimlikli bir fonksiyon olduğu ve
                                bu fonksiyonun bir parametre almadığı ve
                                bir tamsayı geri döndürdüğü derleyiciye bildiriliyor. */
int main() {
   sayiyiKonsolaYaz(13);
   int toplam= ikiSayiyiTopla (10,20);
   //...
}
```

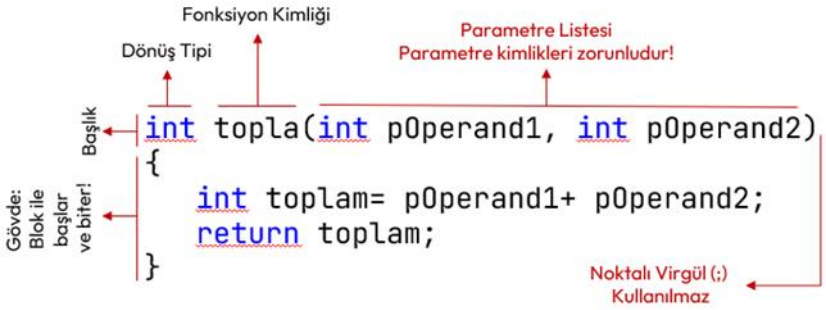
Fonksiyon adı, fonksiyonun benzersiz kimliğidir ve kimliklendirme, değişken kimliklendirme kuralları burada da geçerlidir. Bağımsız değişkenler parametre olarak adlandırılır. Bir fonksiyon çağrıldığı anda parametreye gerçek parametre ya da argüman adı verilir. Parametre listesi, bir işlevin parametrelerinin veri tipini, sırasını ve sayısını belirtir. Parametreler isteğe bağlıdır; yani bir fonksiyon hiçbir parametre içermeyebilir.



Şekil 18. Fonksiyon Bildirimi Örneği

Fonksiyona ilişkin bir bildirim yapıldığında bir yerlerde böyle bir fonksiyon olduğu derleyiciye iletilmiş olur. Bildirim sonrasında artık onu çağırabiliriz. Tanımı daha sonradan yapılabilir. Bildirim yapılmaz ise fonksiyon tanımlanmalıdır.

Fonksiyon Tanımı Nasıl Yapılır?



Şekil 19. Fonksiyon Tanım Örneği

Bir fonksiyon bir değer döndürebilir. Dönüş tipi, işlevin döndürdüğü değerin veri tipidir.

Fonksiyon adı, fonksiyonun benzersiz kimliğidir ve kimliklendirmede, değişken kimliklendirme kuralları geçerlidir. Eğer daha önce fonksiyon bildirimi (function declaration) yapılmış ise aynı kimlik kullanılır.

Bağımsız değişkenler parametre olarak adlandırılır. Parametre listesi, bir işlevin parametrelerinin veri tipini, sırasını ve sayısını belirtir. Parametreler isteğe bağlıdır; yani bir fonksiyon hiçbir parametre içermeyebilir. Bildirimden farklı olarak her bir parametreye kimlik verilmelidir.

Fonksiyon Gövdesi, fonksiyonun ne yaptığını tanımlayan talimatları içerir. Yapısal programlama yapıldığından ilk önce değişkenler tanımlanmalı daha sonra bu değişkenleri işleyecek talimatlar yazılmalıdır. Fonksiyonda istenilen sonuca ulaşmak için kullanacağı adımlara karar verilir yani algoritması belirlenir.

Dönüş tipine uygun olarak bir değer, return talimatı ile geri döndürülmelidir. Return talimatı çoğunlukla fonksiyonundaki son talimat olarak görünür.

Return talimatı, bir fonksiyonun yürütülmesini sonlandırır ve kontrolü çağrıldığı fonksiyona geri verir. Yani programın icrasına devam etmek için çağrıldığı yere dönülür. Bir fonksiyonun tanımında yer alan dönüş tipi (örnekte `int`) ile `return` talimatına ilişkin ifade sonucu (örnekte `toplam`) aynı veri tipinde olmalıdır.

Fonksiyonun çağrıldığı yerde, fonksiyonun döndürdüğü değerin tipi ile eşleşen bir değişkene dönüş değeri atanır.

Bildirimi yapılan bir fonksiyon, C++ derleyicisi tarafından dahil edilen başlık dosyalarını içerecek şekilde derlemeye konu olacak dosyaların tamamında yalnızca bir kez tanımlanabilir. Buna tek tanım kuralına (one definition rule) adı verilir.

```
int topla(int a,int b) {
    return a+b;
}
int topla(int x,int y) { // error: redefinition of ‘int topla(int, int)’
    return x+y;
}
```

Kullanıcı Tanımlı Fonksiyon Örnekleri

Aşağıda topla adlı bir kullanıcı tanımlı fonksiyona ilişkin program verilmiştir.

```
#include <iostream>
using namespace std;
int topla(int,int); // kullanıcı tanımlı "topla" kimlikli fonksiyon bildirimi
int main () {
   int sonuc;
   /* ÇAĞRI ORTAMI: main fonksiyonu içinde "topla" kimlikli
      Fonksiyonu çağrılacak. */
   topla(1,2); /* topla fonksiyonu çağrılıyor.
                  Ama geri döndürdüğü değer kullanılmıyor. */
   cout << "1. toplam:" <<  topla(2,3) << endl; /* topla fonksiyonu çağrılıyor.
                                                   Geri döndürdüğü değer printf
                                                   fonksiyonuna argüman olarak
                                                   giriyor. */
   sonuc= topla(3,4);  /* topla fonksiyonu çağrılıyor.
                          Geri döndürdüğü değer sonuc değişkenine atanıyor. */
   cout << "2. toplam:" << sonuc << endl;
   sonuc= topla(1,2)+topla(3,4); /* topla fonksiyonu iki kez çağrılıyor.
                                    Her bir çağrıdaki geri dönüş değeri
                                    gecici bir yerde saklanıp toplamı
                                    sonuc değişkenine atanıyor. */
   cout << "3. toplam:" << sonuc << endl;
   sonuc= topla(topla(1,2),3)+topla(4,5); /* ... */
   cout << "4. toplam: " << sonuc << endl;
}

// kullanıcı tanımlı topla fonksiyonu tanımı (definition)
int topla(int pOperand1,int pOperand2) //Fonksiyon başlığı
{ //Fonksiyon bloğu başlangıcı
   int toplam=pOperand1+ pOperand2;
   return pOperand1+ pOperand2;
} //Fonksiyon bloğu bitişi
```

```
/* Program çalıştırıldığında:
toplam: 5
toplam: 7
toplam: 10
toplam: 15

...Program finished with exit code 0
*/
```

Aşağıda pozitif sayı girilmesini garanti ettikten sonra sayının 10 sayısına bölünebilirliğini bulan fonksiyon içeren program verilmiştir.

```
#include <iostream>
using namespace std;
int pozitifSayiOku(); // parametre almayan bir fonksiyon bildirimi.
int onaBolunurMu(int);
// int veri titpinde parametre alan bir başka fonksiyon bildirimi
int main () {
    int birPozitifSayi=pozitifSayiOku();
    if (onaBolunurMu(birPozitifSayi))
        cout << "Girilen Pozitif Sayı 10 ile bölünebilir.";
    else
        cout << "Girilen Pozitif Sayı 10 ile tam bölünmez.";
}
int pozitifSayiOku() {
    int okunan;
    do {
        cout << "Lütfen pozitif sayı giriniz: ";
        cin >> okunan;
        if (okunan<=0)
            cout << "HATA:Pozitif Sayı GİRMEDİNİZ!" << endl;
    } while (okunan <= 0);
    return okunan;
}
int onaBolunurMu(int p){
    return (p%10)==0;
}
```

Örnek programda görüldüğü üzere yapısal programa kapsamında bir programın iskeleti oluşmuştur. Ana program içinde problemin çözümü, çeşitli fonksiyonların birbirini çağırması ile yapılmıştır. Her bir fonksiyon içinde işlenecek veriye ilişkin değişkenler (en basit veri yapısı) tanımlanmış ve bu veriler ile bu veri yapılarını işleyen kontrol işlemleri birbirinden ayrılmıştır. Okunaklılık çok yüksek olan bir kod elde edilmiştir.

Bunların dışında hiçbir değer geri döndürmeyen fonksiyonlar da tanımlayabiliriz. Bunlara diğer dillerde prosedür (procedure) adı da verilir. Bu durumda değeri olmayan veri tipi olan `void` anahtar kelimesi kullanılır. Bu tür fonksiyonların da geri dönüş talimatı yalnızca `return;` şekilde yazılır.

```
#include <iostream>
using namespace std;

void printMenu(); // geri değer döndürmeyen bir fonksiyon bildirimi.

int main() {
    int secim;
    do {
        printMenu();
        cin >> secim;
        switch(secim) {
            case 1:
                cout << "Verileri Sakladım." << endl;
                break;
```

```
            case 2:
                cout << "Verileri Okudum." << endl;
                break;
            case 0:
                cout << "Programdan Çıkılıyor..." << endl;
                break;
            default:
                cout << "Tekrar Seçim Yapınız!" << endl;
        }
    } while (secim!=0);
}

void printMenu() {
    cout << endl;
    cout << "1-Verileri Yaz." << endl;
    cout << "2-Verileri Oku." << endl;
    cout << "0-ÇIKIŞ." << endl;
    return;
}
```

Çağrı Kuralı

Çağrı kuralı (calling convention), bir fonksiyon çağrısıyla karşılaşıldığında argümanların fonksiyona hangi sıraya göre iletileceğini belirtir. İki olasılık vardır; Birincisi argümanlar soldan başlanarak sağa doğru fonksiyona geçirilir. İkincisi ise C dilinde kullanılır ve argümanlar sağdan başlanarak sola doğru fonksiyona geçirilir.

```
#include <iostream>
using namespace std;
int topla(int, int, int);
int main () {
    int toplam;
    toplam=topla(10,20,30);
    cout << "toplam:" << toplam << endl;
}
int topla(int x, int y, int z) {
    cout << "x:"<< x << " y:" << y  << " z:" << z<< endl;
    return x+y+z;
}
/* Program çalıştırıldığında:
x:10 y:20 z:30
toplam:60
*/
```

Yukarıdaki örnek incelendiğinde argümanların hangi sırada fonksiyona geçirildiğinin bir önemi yoktur. Ancak aşağıdaki verilen örnekteki durumu inceleyelim;

```
#include <iostream>
using namespace std;
int topla(int, int, int);
int main () {
    int a = 1,toplam;
    toplam = topla(a, ++a, a++);
    cout << "toplam:" << toplam << endl;
}
int topla(int x, int y, int z) {
    cout << "x:"<< x << " y:" << y  << " z:" << z<< endl;
    return x+y+z;
}
/* Program çalıştırıldığında:
x:3 y:3 z:1
```

```
toplam:7
*/
```

Bu kodun çıktısı `1 2 3` olarak beklenir ancak çağrı kuralından ötürü çıktı `3 3 1` olur. Bunun nedeni C++ dilinin çağrı kuralının sağdan sola olmasıdır. Bunu şöyle açıklayabiliriz;

1. Fonksiyona sağdan ilk argüman olarak a değişkeni değeri 1 geçirilir.
2. Ardından a++ ifadesi ile a değişkeni 2'ye yükseltilir.
3. Sonra ++a ifadesi ile a değişkeni 3'e yükseltilir.
4. Fonksiyona ikinci argüman olarak 3 geçirilir.
5. Fonksiyona son olarak a'nın son değeri olan 3 geçirilir.

Depolama Sınıfları

Depolama sınıfları (storage classes) bir değişkenin ömrünü (lifetime), görünürlüğünü (visibility), bellek konumunu (memory location) ve başlangıç değerini (initial value) belirlemek için kullanılır. Bu özellikler temel olarak bir programın çalışma süresi boyunca belirli bir değişkenin varlığını izlememize yardımcı olan kapsam (scope), görünürlüğü ve yaşam süresini içerir.

Yapısal programlama ve fonksiyon kavramının bilgisayarlarda kullanılmasıyla birlikte işlemciler de de değişiklikler olmuştur. *İşlemcinin Emri Alma, Çözme ve İcra Döngüsü* başlığı altında belirtilen işlemci kaydediciler dışında yeni kaydediciler de eklenmiştir;

– Verilerle icra edilen kod farklı bellek bölgesinde bulunur. Bu nedenle verileri işaret eden veri bellek adreslerini tutan kaydedicisi (data memory register).
– Yığın bellek kaydedicisi (stack register), fonksiyon çağrıları sırasında mevcut işlemci durumu ve yerel değişkenler gibi bazı işlemleri depolamak için kullanılan belleği gösteren (stack pointer) adresi tutan kaydedicidir.
– Bayrak kaydedicisi (flag register), işlemcinin durumunu veya sıfır (zero) bayrağı, taşıma (carry) bayrağı, işaret (sign) bayrağı, taşma (overflow) bayrağı, eşlik (parity) bayrağı, yardımcı taşıma (auxiliary carry) bayrağı ve kesme etkinleştirme (interrupt enable) bayrağı gibi çeşitli işlemlerin sonucunu tutmak için kullanılan özel bir kayıt türüdür.
– Genel amaçla kullanılan kaydediciler (general purpose registers).

Eklenen bu kaydediciler ile derlenen her bir program dört bellek bölgesinden (segment) oluşur;

1. Kod bellek ya da kaynak koda ithafla metin bellek (code segment / text segment) icra edilecek emirleri içeren bellek.
2. Programın işleyeceği verileri tutan veri belleği (data segment).
3. Fonksiyonların çağırmadan (call) önce mevcut işlemci durumu ve yerel değişkenler gibi bazı işlemleri depolamak için ve fonksiyondan dönülünce (return) işlemciyi ve çağrı ortamını eski hale getirmek için kullanılan yığın bellek (stack segment). Bu bellek bu nedenle son giren veri ilk çıkar (last in first out-LIFO) mantığıyla çalışır.
4. Programın icrası sırasında dinamik olarak eklenip çıkarılan veriler için hurdalık gibi kullanılan öbek bellek (heap segment). İleride *Dinamik Hafıza* başlığında anlatılacaktır.

Depolama sınıfları tanımlanan değişkenleri hangi bellek bölgesinde yer alacağını belirler. Beş tür depolama sınıfı vardır; `auto`, `extern`, `static`, `register` ve `mutable`. Mutable depolama sınıfı nesnelerle ilgili olduğundan *Mutable Depolama Sınıfı* başlığında anlatılacaktır.

Auto Depolama Sınıfı

Otomatik (auto) depolama sınıfı, bir fonksiyon veya blok içinde kimliklendirilmiş tüm değişkenler için ön tanımlı (default) depolama sınıfıdır. C++ dilinde bu depolama sınıfı için değişkenlerde hiçbir sıfat kullanılmaz. Otomatik değişkenlere; Yalnızca blok içinden erişilebilir, Değişkenin kapsamı içinde bulunduğu blok ile sınırlıdır ve yığın bellekte (stack segment) tutulur.

Otomatik değişkenlere ilişkin örnek aşağıda verilmiştir;

```
#include <iostream>
```

```
using namespace std;
int main() {
    int a = 32;
    int b=20;
    /* a ve b değişkenleri bu fonksiyon bloğu ve iç bloklarda geçerlidir.
       Yani faaliyet alanları (scope) bu fonksiyon bloğu ile sınırlıdır.*/
    if (a==32){
        int b=10; // Ana fonksiyon bloğunda tanımlı b değişkenine artık ulaşılamaz.
        int c=50;
        /* if bloğunda ve tanımlanabilecek iç bloklarda geçerlidir. */
        a=16;// Bu blokta a da geçerlidir.
        cout << "a:" << a << " b:" << b << " c:" << c << endl;
             // Çıktı: a:16, b:10 c:50
   }
   /*if bloğu dışında sadece ana fonksiyon bloğunda
     tanımlı değişkenlere ulaşılabilir. */
   cout << "a:" << a << " b:" << b; // Çıktı: a:16, b:20
}
```

Static Depolama Sınıfı

Statik (static) depolama sınıfı yaygın olarak kullanılan statik değişkenleri kimliklendirmek için kullanılır. Statik değişkenler, kapsam (scope) dışına çıktıktan sonra bile değerlerini koruma özelliğine sahiptir! Statik değişkenler;

- Kapsamları yani geçerli olduğu yer, tanımlandıkları fonksiyonla ya da blokla sınırlıdır.
- Kapsamlarındaki son kullanımlarının değerini korur.
- Program genelinde geçerli olan evrensel (global) statik değişkenlere programın herhangi bir yerinden erişilebilir.
- Veri bellekte (data segment) yer alır. Derleme sırasında bu bellek hep sıfırla doldurulduğundan ilk değerler (initial value) hep sıfırdır.

Statik (static) olarak tanımlanan değişken, program başladığında veri bellekte oluşturulup program bitene kadar aynı bellek bölgesini kullanan değişkendir.

```
#include <iostream>
using namespace std;
void func() {
    int sayac = 0;
    for (sayac = 1; sayac < 10; sayac++)
    {
        static int statikOlan = 5;
        int statikOlmayan = 10;
        statikOlan++;
        statikOlmayan++;
        cout << "sayaç: " << sayac << ", staticOlan: " << statikOlan << endl ;
        cout << "sayaç: " << sayac << ", statikOlmayan: " << statikOlmayan << endl;
   }
   //Burada statikOlan değişkene erişilemez.
}
int main() {
    func();
}
/* Program çalıştırıldığında:
Sayaç: 1, staticOlan: 6
sayaç: 1, statikOlmayan: 11
sayaç: 2, staticOlan: 7
sayaç: 2, statikOlmayan: 11
sayaç: 3, staticOlan: 8
sayaç: 3, statikOlmayan: 11
sayaç: 4, staticOlan: 9
```

```
sayaç: 4, statikOlmayan: 11

...Program finished with exit code 0
*/
```

Programı çalıştığında `statikOlan` değişken yalnızca `for` bloğu içinde geçerli olduğu ve her kullanımda değerini koruduğu görülmektedir.

Harici Depolama Sınıfı

Harici (extern) depolama sınıfı bize basitçe değişkenin kullanıldığı blokta değil başka bir yerde tanımlandığını söyler. Harici değişkenler;

- Bu değişkenlere değerler tanımlandığı yerde değil farklı bir blokta atanır ve üzerinde değişiklik yapılabilir.
- Bir global değişken, `extern` anahtar sözcüğünü kimliklendirme önüne yerleştirerek harici de yapılabilir.
- Bu değişkenler de veri bellekte (data segment) yer alır. Derleme sırasında bu bellek hep sıfırla doldurulduğundan ilk değerler hep sıfırdır.

Bir harici (extern) değişkene bildirim (declaration) yapıldığında bir başka dosyada tanımlanmış (definition) yapıldığı kabul edilir. Bunun için iki farklı kaynak kod dosyası gereklidir. Birinci kaynak kod dosyası (`extern.c`) aşağıda verilmiştir.

```
//EXTERN.CPP
int externDegisken; //Diğer dosyalarda kullanılacak değişken tanımlandı.
void funcExtern() //Diğer dosyalarda kullanılacak fonksiyon tanımlandı.
{
    externDegisken++;
}
```

Bu dosyadaki değişken ve fonksiyonları kullanan bir başka kaynak kod (`main.cpp`) aşağıda verilmiştir;

```
#include <iostream>
using namespace std;
extern void funcExtern(); // bir başka dosyada tanımlandı. Buada bildirim yapıldı.
int main() {
   extern int externDegisken; // bir başka dosyada tanımlandı. Burada bildirim yapıldı.
   funcExtern();
   cout << "extern: " << externDegisken << endl;
   externDegisken = 2;
   cout << "extern: " << externDegisken << endl;
}
```

Bu iki dosyayı birlikte derlemek için `g++ extern.cpp main.cpp -o main.exe` komutu ile derleyiciye iki farklı dosya parametre olarak verilir. Çevrimiçi derleyicimizde ise bir projeye birden çok kaynak kod işaretli butona basılarak eklenebilir ve derleme yapılabilir.

Kaydedici Depolama Sınıfı

Kaydedici (register) depolama, otomatik değişkenlerle aynı işlevselliğe sahiptir. Tek fark, derleyicinin, eğer boş bir işlemci kaydedicisi mevcutsa değişken olarak o kaydedicinin kullanılmasıdır. Genel amaçlı kaydediciler bu amaçla kullanılır. Bu da bu değişkenlerin, bellekte saklanan değişkenlerden çok daha hızlı olmasını sağlar.

Eğer boş bir CPU kaydedicisi mevcut değilse, değişken yalnızca bellekte saklanır. `register` anahtar sözcüğüyle tanımlanan değişken nitelendirilir. `register` Anahtar sözcüğü ile sadece bloklar içinde tanımlanan yerel değişkenler (local variable) nitelendirilebilir, evrensel değişkenler (global variable) bu anahtar sözcük ile kullanılamaz.

Aşağıda kaydedici (register) depolama sınıfın ilişkin örnek verilmiştir. Kuradaki sayma ve toplama işlemleri uygun olan işlemci kaydedicilerinde yapılır.

```
#include <iostream>
using namespace std;
int main() {
    register int k, sum;
    for(k = 1, sum = 0; k < 1000; sum += k, k++);
        //döngü bloğu olmayan for talimatı örneği
        cout << sum << endl ; //Çıktı:499500
}
```

Öncelikle C++ standardı tek bir değişkenle birden fazla depolama sınıfının kullanılmasını yasaklar. Aynı değişkene hem `static` hem `extern` niteleyicisi kullanılamaz. Tablodaki çöp (garbage) kavramı tahsis edilen bellek içeriği o anda ne ise değişken o değeri alır anlamındadır. Depolama sınıfları aşağıdaki tabloda özetlenmiştir;

Depolama sıfatı	Yer aldığı bellek bölgesi (segment)	Başlangıç Değeri	Kapsamı (scope)	Ömür (lifetime)
Sıfat kullanılmaz!	Yığın Bellek (Stack Segment)	Çöp	Bulunduğu blok ve bu blok içindeki bloklar	Bulunduğu blok sonunda biter
`extern`	Veri Bellek (Data Segment)	Sıfır	Evrensel (tüm dosyalarda)	Program sonunda biter
`static`	Veri Bellek (Data Segment)	Sıfır	İçinde bulunduğu blok	Program sonunda biter
`register`	İşlemci kaydedicileri (CPU Registers)	Çöp	Bulunduğu blok ve bu blok içindeki bloklar.	Bulunduğu blok sonunda biter

Tablo 18. Depolama Sınıfları Özet Tablosu

Evrensel ve Yerel Değişken

C++ dilinde bir kaynak koda ait dosya içinde bir değişkene her yerden erişilmek istenirse evrensel değişken (global variable) olarak tanımlanır. Bu değişkenler bir fonksiyon bloğu içinde tanımlanmazlar. Tanımlandıkları yerden sonra kaynak kodun her yerinde kullanılabilirler. Bu nedenle genellikle dosyanın başında tanımlama yapılır.

```
#include <iostream>
using namespace std;
int evrenselDegisken=10; /* evrenselDeğişken bu noktadan sonra
                            her yerde kullanılabilir. */
void fonksiyon();
int main() {
    int yerelDegisken=20; /* Bu yerel değişken sadece main bloğu
                             içinde kullanılabilir.*/
    cout << "evrensel:" << evrenselDegisken << ", yerel: "
         << yerelDegisken << endl;
    evrenselDegisken++; //evrensel değişken değiştiriliyor.
    yerelDegisken--; //yerel değişken değiştiriliyor.
    cout << "evrensel:" << evrenselDegisken << ", yerel:"
         << yerelDegisken << endl;
    fonksiyon();
}
void fonksiyon() {
   int fonksiyonYerelDegiskeni=30;
   evrenselDegisken++;
   cout << "evrensel:"<< evrenselDegisken << ", fonsiyonYerel:"
        << fonksiyonYerelDegiskeni << endl;
   if (fonksiyonYerelDegiskeni==30) {
       int evrenselDegisken=100; //Artık evrensel değişkene bu blokta ulaşılamaz.
       cout << "evrensel:"<< evrenselDegisken << ", fonsiyonYerel:"
            << fonksiyonYerelDegiskeni << endl;
   }
}
```

```
/* Program Çıktısı:
evrensel:10, yerel:20
evrensel:11, yerel:19
evrensel:12, fonsiyonYerel:30
evrensel:100, fonsiyonYerel:30

...Program finished with exit code 0
*/
```

Bu program incelendiğinde if bloğu içindeki değişken her ne kadar `evrenselDegişken` olarak kimliklendirilse de yerel değişkendir ve sadece bu if bloğu içinde geçerlidir. Yani tanımlamada mantıksal hata (logical error) yapılmıştır.

Yerel değişkenler (local variable) ise bir fonksiyon bloğu ya da bir kod bloğu içinde tanımlanan değişkenlerdir. Bu değişkenlere evrensel değişkenler ile aynı kimlik verilirse yukarıdaki örnekteki gibi evrensel değişkene artık ulaşılamaz. Böyle bir durumda evrensel değişkene ulaşmak için iki tane iki nokta üst üste karakteri olan kapsam çözümleme işleci (scope resolution operator) kullanılır.

```
#include<iostream>
using namespace std;
int ogrenciSayisi = 5; // evrensel değişken
void ogrenciSayisiniYaz() {
    cout<< ogrenciSayisi << endl; // evrensel değişken konsola yazılıyor.
}
int main() {
   int ogrenciSayisi=10; // yerel değişkene evrensel değişkenin kimliği veriliyor
   ogrenciSayisiniYaz();
   ogrenciSayisi++; // yerel değişken değiştiriliyor.
   ::ogrenciSayisi++; // evrensel değişken değiştiriliyor.
   ogrenciSayisiniYaz();
}
```

Fonksiyon Çağırma Süreci

Bir fonksiyon çağrıldığında (call function/invoke function) nelerin yapıldığı aşağıdaki programdan anlaşılabilir;

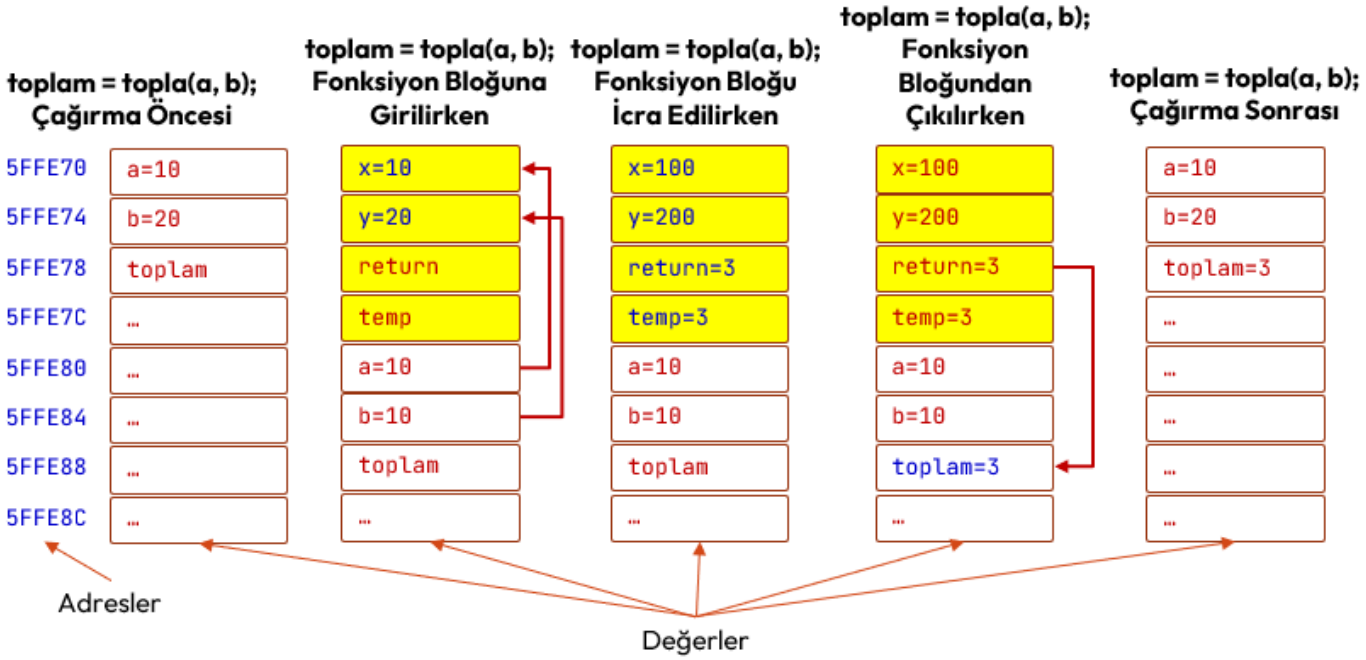
```
#include <iostream>
using namespace std;
int topla(int, int); /* İki tamsayıyı toplayan fonksiyon bildirimi/prototipi */
int main () {
    /* main() fonksiyonu içinde geçerli yerel (local) değişkenler*/
    int a = 10;
    int b = 20;
    int toplam = 0;
    cout << "çağrı öncesi a:" << a << ", b:" << b << ", toplam:" << toplam << endl;
    toplam = topla(a, b);
    cout << "çağrı sonrası a:"<< a << ", b:" <<b << ", toplam:" << toplam << endl;
}
int topla(int x, int y) { /* iki tamsayıyı toplayan fonksiyon tanımı*/
    /* Topla fonksiyonu yerel değişkenler */
    int temp= x+y;
    x=100; y=200;
    /*
      Burada değiştirilenler parametre değişkenleridir.
      main() içindeki yerel değişkenler değildir.
    */
    cout << "çağrı içinde x:" << x << ", y:" << y  << endl;
    return temp;
}
```

```
/* Program çalıştırıldığında:
çağrı öncesi a:10, b:20 toplam:0
çağrı içinde x:100, y:200
çağrı sonrası a:10, b:20 toplam:30

...Program finished with exit code 0
*/
```

Yukarıdaki kod incelendiğinde;

- İcra sırası `topla(a,b)` fonksiyonuna geldiğinde fonksiyon çağrılmadan önce main fonksiyonu yerel değişkenleri a, b ve toplam değişkenleri yığın (stack) belleğe itilir (push).
- `topla(a,b)` fonksiyonu çağrıldığı anda fonksiyon bloğuna başlamadan parametrelere sırasıyla a ve b değişkenlerinin değerleri kopyalanarak fonksiyona geçirilir.
- `topla(a,b)` fonksiyonu icra edilirken fonksiyon yerelindeki `temp` ve parametre değişkenleri `x`, ve `y` istenildiği gibi değiştirilebilir. Geri dönüş değeri de belirlenir.
- `toplam=topla(a,b)` fonksiyon bloğundan çıkılırken geri dönüş değeri çağrı ortamındaki toplam değişkenine atanır ve fonksiyon için yığın (stack) belleğe itilen değerler geri çekilir.

Aşağıda örneği verilen programda fonksiyon çağırma sürecindeki yığın bellek (stack segment) değişimi gösterilmektedir.



Şekil 20. Fonksiyon çağırma sürecinde yığın bellek

Özyinelemeli Fonksiyonlar

Özyineleme (recursion), bir fonksiyonun kendi çağrısını yapma tekniğidir. Bu teknik, karmaşık sorunları çözülmesi daha kolay basit sorunlara ayırmanın bir yolunu sağlar. Özyineleme, yineleme (iteration), döngüler (loop) veya tekrar yerine geçebilecek çok güçlü bir tekniktir. Özyinelemeli (recursive) algoritmalarda, tekrarlar fonksiyonun kendi kendisini kopyalayarak çağırması ile elde edilir. Bu kopyalar işlerini bitirdikçe kaybolur. Bir problemi özyineleme ile çözmek için problem iki ana parçaya ayrılır.

1. Cevabı kesin olarak bildiğimiz temel durum (base case)
2. Cevabı bilinmeyen ancak cevabı yine problemin kendisi kullanılarak bulunabilecek durum.

Faktöriyel örneğini inceleyelim; Matematikte, negatif olmayan bir tam sayı olan n'nin faktöriyeli, $n!$ ile gösterilir ve n 'den küçük veya ona eşit olan tüm pozitif tam sayıların çarpımıdır.

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

Negatif olmayan bir tam sayı olan n'nin faktöriyeli aynı zamanda n'nin bir sonraki daha küçük faktöriyelle çarpımına eşittir. Yani problemin tanımı yine kendisini içerir:

$$n! = n \cdot (n-1)!$$

Örneğin;

$$5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

Aşağıda faktöriyel için girilen sayıdan sonra tüm sayıların çarpımıyla hesaplayan bir fonksiyon yazılmıştır.

```
#include <iostream>
using namespace std;
unsigned int faktoriyel(unsigned int n) { //Fonksiyon tanımı yapıldı. Bildirim yapılmadı.
    int sonuc=1,indis;
    for (indis=n;indis>0;indis--)
        sonuc=sonuc*indis;
    return sonuc;
}
int main() {
    unsigned int sayi,sonuc; //Pozitif tamsayı yanımı yapılmış değişkenler.
    cout << "Pozitif Bir Sayi Giriniz:";
    cin >> sayi;
    sonuc=faktoriyel(sayi);
    cout << sayi << "!=" << sonuc;
}
```

Sıfır sayısı da pozitif bir sayıdır ama sonucum sıfır çıkmaması için 0 sayısının faktöriyeli 1 kabul edilir. Bu aslında faktöriyel problemi için temel durumdur (base case) ve boş ürün (emty product) kuralı olarak bilinir. Kısaca faktöriyel fonksiyonu aşağıdaki şekilde gösterilir;

$$n \in \mathbb{N} \; olmak \; üzere \; f(n) = n! = \begin{cases} 1, & n = 0 \\ n \cdot f(n-1), & n > 0 \end{cases}$$

Özyinelemeli (recursive) olarak ise bu fonksiyon aşağıdaki şekilde kodlanabilir;

```
unsigned int faktoriyel(unsigned int n)
{
    if (n==0)
        return 1;
    else
        return n*faktoriyel(n-1);
}
```

Girilen taban ve üs ile kuvvet hesaplama fonksiyonu özyinelemeli olarak aşağıda verilmiştir;

```
#include <iostream>
using namespace std;
int Kuvvet(int taban, int us) {
    if (taban == 0) { return 0; } //Taban 0 ise 0 dön
    if (us == 0) { return 1; } // Üs 0 ise 1 dön
    return taban* Kuvvet(taban, us - 1);
}
int main(){
    int taban,us;
    cout << "Tabanı Giriniz:";
    cin >> taban;
    cout << "Üssü Giriniz:";
    cin >> us;
    cout << taban <<" uzeri "<< us <<" = " << Kuvvet(taban,us);
}
/*
Tabanı Giriniz:3
Üssü Giriniz:7
```

```
3 uzeri 7 = 2187

...Program finished with exit code 0
*/
```

Satır İçi Fonksiyonlar

Fonksiyon Çağırma Süreci başlığında anlatıldığı üzere fonksiyonlar çağrılırken sürekli yığın belleğe (stack segment) it-çek yapılır. Bazı durumlarda bu işlem performans gereği istenmez. Bu durumda fonksiyonlar, satır içi fonksiyon (inline function) olarak tanımlanır. 2017 C sürümü ile gelen bu özellik, fonksiyona inline sıfatı (inline specifier) kullanılır.

```
#include <iostream>
using namespace std;
inline int topla (int op1, int op2) {
    int toplam= op1+op2;
    return toplam;
}
int main() {
    int x=10, y=15, sonuc;
    sonuc = topla(x,y);
    cout << x << "+" << y << "=" << sonuc;
}
```

Bu kod aşağıdaki şekle çevrilmiş gibi derlenir;

```
#include <stdio.h>
int main() {
    int x=10, y=15, sonuc;
    {
        int toplam= op1+op2;
        sonuc = toplam;
    }
    cout << x << "+" << y << "=" << sonuc;
}
```

Bir fonksiyon aşağıda belirtilen durumlarda satır içi fonksiyon olarak derlenmez;

- Bir döngü içeriyorsa.
- Statik değişkenler içeriyorsa.
- Özyinelemeli ise.
- Dönüş türü `void` haricinde olup return ifadesi işlev gövdesinde mevcut değilse.
- Switch veya `goto` talimatı içeriyorsa.

Bu tip fonksiyon tanımlamadaki amacımız, fonksiyon çağırma yükünüm azaltılması için işlemci kaydedicilerinin daha çok kullanılmasıdır. Bu da icra hızını yükseltir.

Varsayılan Argümanlar

Varsayılan argüman (default argument), bir fonksiyon bildiriminde bir parametre için sağlanan ve çağıran fonksiyon bu parametreler için bir değer sağlamazsa derleyici tarafından otomatik olarak atanan bir değerdir. Bu parametreye çağrı sırasında bir değer geçirilirse, varsayılan değer dikkate alınmaz.

```
#include <iostream>
using namespace std;
void yaz(int a = 10) { //a parametresi için varsayılan argüman 10 dur.
    cout << a << endl;
}
int main() {
    yaz(); // 10
    yaz(200); //200
```

```
}
```

Varsayılan argüman bildirimde farklı, tanımda farklı verilemez;

```
#include <iostream>
using namespace std;
void yaz(int a = 10); //a parametresi için varsayılan argüman 10 dur.
int main() {
    yaz(); // 10
    yaz(200); //200
}
void yaz(int a = 20); // Derleme hatası olur. 10 olarak verilmelidir.
{
    cout << a << endl;
}
```

Birden fazla parametreye varsayılan argüman atanacak ise sağdan sola doğru hepsine atanmalıdır;

```
void yaz(int x, int y = 20); // Geçerli Bildirim.
void yaz(int x = 10, int y); /* Derleme hatası olur. GEÇERSİZ Bildirim:
                                sağdaki y parametresine varsayılan
                                argüman atanmadı */
```

DİZİLER

Dizi Nedir?

Şu ana kadar en basit veri yapıları (data structure) olan değişkenlerle karşılaştık. Programlamada kullanılan bir başka veri yapısı da dizilerdir (array).

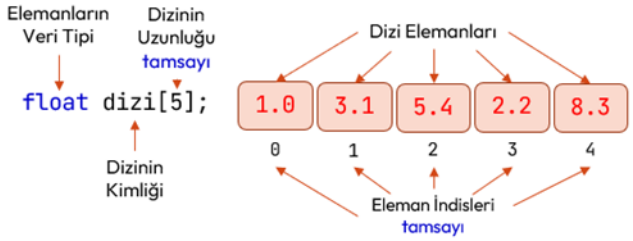
Matematikte diziler; bir sıralı elemanlardan oluşan bir listedir. Sıralı elemanların sayısına dizinin uzunluğu (array size) denir. Elemanların sırası indis (index) olarak adlandırılır. Kümenin aksine aynı elemanlar dizide farklı konumlarda birkaç kez bulunabilir. Elemanlara terim adı da verilir.

Örneğin Elemanları 1'den 10'a kadar sayıların karesinden oluşan bir dizi; $(a_k)_{k=1}^{10}, a_k = k^2$ veya $(a_1, a_2, a_3, \ldots, a_{10})$ veya $(1,4,9,\ldots,100)$ olarak gösterilir. Örneğin;

- $('K','İ','T','A','P')$ Dizisi ile $('P','A','T','İ','K')$ dizisi birbirinden farklıdır. Aynı elemanlardan oluşmasına rağmen elemanların sıraları farklıdır.
- $(1,3,5,1,2,1,7)$ Dizisinde birden farklı indiste aynı 1 elemanı vardır. Bu dizinin geçersiz olduğu anlamına gelmez.
- Matematikte sonsuz elemanı olan sonsuz diziler tanımlanabilir. Ama programlamada hep sonlu elemanlı diziler yani uzunluğu (size) belirli diziler kullanılır.

Tek boyutlu Diziler

C++ Dilinde sonlu elemanlı diziler aşağıdaki gibi tanımlanır;



Şekil 21. Tek Boyutlu Dizi Tanımlama

1. Dizi içinde, benzer veri tipindeki (data type) veri öğelerini bulundurur ve bu öğeler bitişik bellek bölgesini paylaşırlar.
2. Dizi elemanları, ilkel veri tipleri (int, float, char) veya daha sonra göreceğimiz kullanıcı tanımlı tip olan yapı (struct) veya gösterici (pointer) olabilir.
3. Aynı değer birkaç farklı yerde dizi içinde bulunabilir.
4. Dizinin veri tipi, dizideki elemanlarının veri tipini belirler. Bir başka deyişle elemanların veri tipi dizinin veri tipiyle aynı olmalıdır.
5. Dizinin uzunluğu tamsayı değişmez (integer literal) olup dizi kimliklendirmesi sırasında belirtilmelidir. Derleyici buna göre bellekte yer ayırır. Dolayısıyla dizi, bir kez kimliklendirildiğinde dizinin uzunluğu değiştirilemez.
6. Dizinin indisi (index) de her zaman sıfırdan başlar ve sıra belirttiğinden her zaman tamsayıdır (integer).

C dilinde beş tamsayı elemanı olan bir dizi aşağıdaki şekilde tanımlanır.

```
int tamsayiDizisi[5];
```

Beş gerçek sayı elemanı olan bir dizi ise aşağıdaki şekilde tanımlanır.

```
float reelsayiDizisi[5];
```

Dizinin elemanlarına tanımlama sırasında başlangıç değeri (initial value) değer verilebilir;

```
int tamsayiDizisi1[5]= {1, 2, 3, 4, 5}; //Elemanlar sırasıyla 1,2,3,4,5
int tamsayiDizisi2[5]= {0}; //Elemanların Hepsi 0
```

Dizinin elemanlarına hepsine başlangıç değeri verilmeyebilir;

```
float a[5] = {1.0, 2.0, [4] = 12.0}; // 1.Eleman 1.0, 2.Elaman 2.0 ve 5.eleman 12.0
```

Dizinin elemanlarına ayrı ayrı erişilip değerleri değiştirilebilir ya da yeni değer ataması yapılabilir;

```
int a[5]={0};
a[0]=1; //Birinci elemana 1 değeri atandı
a[1]++; //İkinci elemanın değeri artırıldı
a[2]--; //Üçüncü elemanın değeri azaltıldı
a[4]=12; //Beşinci elemana 12 değeri atandı.
```

Diziler belleği verimli kullanan ve tek bir değişken ile elemanlara erişim sağlayan bir çözüm sunar. Bir dizideki elemanlar, bellekte bitişik konumda yer aldığından herhangi bir öğeye kolaylıkla erişebiliriz. Dizi kullanmanın başlıca üstünlükleri şunlardır:

– İndisleri kullanarak dizi öğelere rastgele ve hızlı erişim. Her öğenin bir indisi (index) olduğundan doğrudan erişilebilir ve değiştirilebilir.
– Birden fazla öğeden oluşan tek bir dizi oluşturduğundan daha az kod satırı yazılır.
– Daha az kod satırı yazılarak sıralama yapılabilir.

Dizi kullanmanın zayıf yönleri ise;

– Kimliklendirme sırasında karar verilen değişmez (literal) sayıda elemanın üzerinde işlem yapılır.
– Dizi dinamik değildir. Araya elaman ekleme veya çıkarma yapılamaz.

Örnek olarak klavyeden girilen 5 adet tamsayıyı, giriş sırasının tersinden ekrana yazan C programını verebiliriz;

```
#include <iostream>
using namespace std;
int main() {
  int dizi[5]; /* 5 elemanı tamsayı olan bir dizi tanımlandı*/
  int indis; /* dizi elemanlarını gezecek ve indis olarak kullanılacak
                tamsayı bir değişken tanımlandı */

  for (indis =0; indis <5; indis ++){
    cout << indis << ". sayiyi girin:";
    cin >> dizi[ indis ]; /* indis ile belirtilen elemana klavyeden
                             olunan değer atanıyor */
  }

  cout << "Tersten Sayilar:" << endl;
  for (indis=4; indis >=0; indis --)
    cout << "dizi[" << indis << "]=" << dizi[ indis ] <<endl;
}
/* Program Çıktısı:
0. sayiyi girin:1
1. sayiyi girin:3
2. sayiyi girin:6
3. sayiyi girin:8
4. sayiyi girin:3
Tersten Sayilar:
dizi[4]=3
dizi[3]=8
dizi[2]=6
dizi[1]=3
dizi[0]=1

...Program finished with exit code 0
*/
```

Bir başka örnek; Klavyeden girilen 10 adet sınav notuna göre, ortalamanın üstünde olan notları ekrana yazan C programı;

```
#include <iostream>
using namespace std;
#define BOYUT 10
int main() {
    unsigned notlar[BOYUT]; /* Her terimi pozitif olan 10 elemanlı dizi */
    float ortalama, toplam=0;
    int indis; //indis için tamsayı değişken
    for (indis=0; indis<BOYUT; indis++){
        cout << indis << ". notu girin:";
        cin >> notlar[indis];
        toplam+=notlar[indis]; // Her eleman girildiğinde toplanıyor
   }
   ortalama=toplam/BOYUT;
   cout << "Ortalamanın (" << ortalama << ") Üzerindeki Notlar:"<< endl;
   for (indis=0; indis<BOYUT; indis++)
       if (notlar[indis]>ortalama)
           cout << "notlar [" << indis << "]=" << notlar[indis] << endl;
}
/* Program Çıktısı:
1. notu girin:85
2. notu girin:45
3. notu girin:60
4. notu girin:75
5. notu girin:40
6. notu girin:25
7. notu girin:35
8. notu girin:60
9. notu girin:50
Ortalamanın (57.5) Üzerindeki Notlar:
notlar [0]=100
notlar [1]=85
notlar [3]=60
notlar [4]=75
notlar [8]=60

...Program finished with exit code 0
*/
```

Bir başka örnek olarak 10 adet satış miktarlarının, ortalamaya olan uzaklıkları yani sapma (deviation) ve karesi alınmış sapmalar yani varyans (variance) ile standart sapmayı yazdıran C program verilebilir.

```
#include <iostream>
#include <iomanip>  // setw() ve setprecision için
#include <cmath>
using namespace std;
#define BOYUT 10
int main() {
    float satislar[BOYUT]={100.0,850.0,500.0,600.0,750.0,650.0,450.0,800.0,900.0,110.0};
    float ortalama, toplam=0, varyansToplam=0, standartSapma;
    int indis;
    for (indis=0; indis<BOYUT; indis++)
        toplam+= satislar[indis];
    ortalama=toplam/BOYUT;
    cout << "Ortalama:" << ortalama << endl;
    cout << setw(5);
    for (indis=0; indis<BOYUT; indis++) {
        float sapma= satislar[indis]-ortalama;
        varyansToplam+=sapma*sapma;
```

```
        cout << "Sapma: " << setw(5) << sapma
             << " Varyans: " <<  setw(8) << sapma*sapma << endl;
        /*
         setw(5): izleyen değişken konsola yazılırken 5 karakter
                  genişliğinde yazılsın.
         setw(8): izleyen değişken konsola yazılırken 8 karakter
                  genişliğinde yazılsın. */
    }
    standartSapma=sqrt(varyansToplam/BOYUT);
    cout << "Standart Sapma:"<< setprecision(8) << standartSapma;
    /* setprecision(8): izleyen reel sayı değişkeni konsola
                        yazılırken 9 karakter genişliğinde yazılsın. */
}
/* Program Çıktısı:
Ortalama:571
Sapma:  -471 Varyans:   221841
Sapma:   279 Varyans:    77841
Sapma:   -71 Varyans:     5041
Sapma:    29 Varyans:      841
Sapma:   179 Varyans:    32041
Sapma:    79 Varyans:     6241
Sapma:  -121 Varyans:    14641
Sapma:   229 Varyans:    52441
Sapma:   329 Varyans:   108241
Sapma:  -461 Varyans:   212521
Standart Sapma:270.49768

...Program finished with exit code 0
*/
```

Ortalamaya olan uzaklıklar yani sapma verilerin ortalamaya ne kadar yakın olduğunu gösteren dağılımdır. Sapmaların toplamı sıfır olabileceğinden karelerinin toplamı yani varyans hesaplanır. Varyansın eleman sayısına bağlı karekökü de standart sapmayı verir. Standart sapmanın büyük olması verilerin ortalamadan daha uzak yayıldıklarını; küçük bir standart sapma ise verilerin ortalama etrafında daha çok yakın gruplaştıklarını gösterir.

Bir başka örnek olarak 5 elemanlı diziye girilen elemanlardan tekil (unique) olanlarını bulan program verilebilir.

```
#include <iostream>
using namespace std;
#define BOYUT 5
int main() {
    int dizi[BOYUT];
    int indis;
    for (indis=0; indis<BOYUT; indis++) {
        cout << indis << ". Sayıyı Girin:";
        cin >> dizi[indis];
    }
    cout << "Dizi içinde tekil (unique) olan rakamlar:"<< endl;
    for (indis=0; indis<BOYUT; indis++) {
        int indis2, sayac = 0;  // Her elemanda sayacı sıfırla
        for (indis2 = 0; indis2 < BOYUT; indis2++)
            if (indis != indis2) //elemanın kendisini kontrol etmiyoruz
                if (dizi[indis] == dizi[indis2])
                    sayac++;
        if (sayac == 0)
            cout << "dizi["<< indis << "]:" << dizi[indis]
                 << " tekil olarak dizide bulundu."<< endl;
  }
}
```

```
/* Program Çıktısı:
0. Sayıyı Girin:23
1. Sayıyı Girin:12
2. Sayıyı Girin:23
3. Sayıyı Girin:14
4. Sayıyı Girin:14
Dizi içinde tekil (unique) olan rakamlar:
dizi[1]:12 tekil olarak dizide bulundu.

...Program finished with exit code 0
*/
```

Bir başka örnek olarak 5 elemanlı diziye girilen en büyük elemanına en küçük elemanını ekleyen program verilebilir;

```
#include <iostream>
#include <iomanip> // setw()
using namespace std;
#define BOYUT 5
int main() {
    int dizi[BOYUT]={10,12,3,4,6};
    int enBuyuk, enKucuk, enKucukIndex, enBuyukIndex, i ;
    cout << "ÖNCE:" << endl; //Dizinin İlk hali konsola yazdırılıyor.
    for (i = 0; i < BOYUT; i++)
        cout << setw(4) << dizi[i];
    cout << endl;
    enKucuk=dizi[0]; enBuyuk=dizi[0]; /* İlk elemanlar hem en küçük
                                         hem de en büyük olsun. */
    enKucukIndex=0,enBuyukIndex=0; /* En küçük ve en büyük elemanların
                                      indisi 0 olsun. */
    for (i=0; i<BOYUT; i++){ /* En küçük ve en büyük eleman ile
                                indislerini bulan kısım. */
        if (dizi[i]<enKucuk) {
            enKucuk=dizi[i];
            enKucukIndex=i;
        }
        if (dizi[i]>enBuyuk) {
            enBuyuk=dizi[i];
            enBuyukIndex=i;
       }
    }
    cout << "En Büyük Eleman: dizi[" << enBuyukIndex << "]:"
         << dizi[enBuyukIndex] << endl;
    cout << "En Küçük Eleman: dizi["<< enKucukIndex << "]: "
         << dizi[enKucukIndex] << endl;
    dizi[enBuyukIndex]+=dizi[enKucukIndex]; //En büyük elemana en küçüğünü ekleme
    cout << "SONRA:" << endl; //Dizinin son hali konsola yazdırılıyor.
    for (i = 0; i < BOYUT; i++)
        cout << setw(4) << dizi[i];
}
/* Program Çıktısı:
ÖNCE:
  10  12   3   4   6
En Büyük Eleman: dizi[1]:12
En Küçük Eleman: dizi[2]: 3
SONRA:
  10  15   3   4   6

...Program finished with exit code 0
*/
```

İki Boyutlu Diziler

Şu ana kadar gördüğümüz tek boyutlu dizilerdi. İki boyutlu diziler matematikten de bildiğimiz üzere matris (matrix) olarak adlandırılır. İki boyutlu dediğimizde en-boy ve satır-sütun gibi kavramlar akla gelmelidir. Matris işlemleri gibi bazı problemlerde; bir dizinin her bir elemanının da dizi olması istenir. Bu tür iki boyutlu dizilerde en içteki dizinin boyutu Kimliklendirmede sağda yer alır.

Aşağıda iki boyutlu matris tanımlamalarına örnek verilmiştir;

```
float matris[2][3];  /* Her bir elemanı 3 elemanlı bir dizi olan 2 boyutlu dizi:
                        2 satır üç sütunlu bir matris */
//Aşağıda 2x3 matrise ilk değer verme:
float matris2[2][3]= {
                       {1.0,2.0,3.0}, //Birinci Satır: 3 Elemanlı bir dizi
                       {2.0,4.0,6.0}  //İkinci Satır: 3 Elemanlı bir dizi
                     };
int karematris[2][2];
//2x2 matrise ilk değer verme:
int karematris2[2][2]= {
                         {1,2}, //Birinci Satır: 2 elemanlı bir dizi
                         {5,6}  //Birinci Satır: 2 elemanlı bir dizi
                        };
int karematris3[3][3];
//İlk Değerlieri DÜZ verme (flat initialization):
int karematris4[2][2]= { 1, 2, 3, 4 }; /* Toplamda 2x2=4 elemanı var.
                                          Bütün elemanlara peşpeşe ilk
                                          değer verilebilir */
int karematris5[3][3]= { 11, 22, 33, 21, 22, 23, 31, 32, 33};
/* Toplamda 3x3=9 elemanı var. Bütün elemanlara peşpeşe ilk değer verilebilir */
```

Örnek olarak 3x4'lük matris elemanlarını klavyeden girip, tablo halinde ekrana yazdıran programı verebiliriz;

```
#include <iostream>
#include <iomanip> // setw()
using namespace std;
#define SATIR 3
#define SUTUN 4
int main() {
    int matris[SATIR][SUTUN];
    int i,j;
    /*
    for (j=0; j<SUTUN; j++)  //Birinci Satır Okuyalım
         cin >> matris[0][j];
    for (j=0; j<SUTUN; j++)  //İkinci Satır Okuyalım
        cin >> matris[1][j];
    for (j=0; j<SUTUN; j++)  //Üçüncü Satır Okuyalım
        cin >> matris[2][j];
    //Bunun yerine iç içe iki for tanımlanır;
    */
    for (i=0; i<SATIR; i++) //İkinci boyut için i indisi
        for (j=0; j<SUTUN; j++)  //Birinci boyut için j indisi
            cin >> matris[i][j];
    cout <<"TABLO:"<< endl;
    for (i=0; i<SATIR; i++) {  //İkinci Boyut İçin
        for (j=0; j<SUTUN; j++)  //Birinci Boyut İçin
            cout << setw(4) << matris[i][j];
        cout << endl;
    }
}
/* Program Çıktısı:
```

```
1 3 4 5 8 7 6 5 4 4 5 6 8 9 0 2
TABLO:
   1   3   4   5
   8   7   6   5
   4   4   5   6

...Program finished with exit code 0
*/
```

Bir başka örnek olarak elemanları klavyeden girilen 3x2'lık matrisin satır ve sütun toplamlarını ekrana yazan programı verilebilir;

```
#include <iostream>
using namespace std;
#define SATIR 3
#define SUTUN 2
int main() {
    int matris[SATIR][SUTUN]; //Matris Tanımı
    int i,j;  //i sutun için, j satır için tanımlandı
    for (i=0; i<SATIR; i++) {  //İkinci Boyut (SATIR) İçin
        cout << i << ". Satırı Girin:";
        for (j=0; j<SUTUN; j++)  //Birinci Boyut (SUTUN) İçin
            cin >> matris[i][j];
    }
    int satirToplamlar[SATIR]={0}; //Bütün elemanları 0 olan dizi
    for (i=0; i<SATIR; i++) {  // İkinci Boyut İçin
        for (j=0; j<SUTUN; j++) // Birinci Boyut İçin
            satirToplamlar[i]+=matris[i][j];
        /* İçdeki for bitince tüm satırdaki elemanlar toplanmış oldu */
    } /* Dışdaki for ile de her bir satır için toplam tekrarlanıyor */
    cout << "Satır Toplamları:" << endl;
    for (i=0; i<SATIR; i++)
        cout << i << ". Satır Toplamı:" <<satirToplamlar[i] << endl;
    int sutunToplamlar[SUTUN]={0}; //Bütün elemanları 0 olan dizi
    for (j=0; j<SUTUN; j++) // Birinci Boyut İçin
        for (i=0; i<SATIR; i++) {  // İkinci Boyut İçin
            sutunToplamlar[j]+=matris[i][j];
        /* İçdeki for bitince tüm sutundaki elemanlar toplanmış oldu*/
    }/* Dışdaki for ile de her bir sutun için toplam tekrarlanıyor */
    cout << "Sütun Toplamları:" << endl;
    for (j=0; j<SUTUN; j++)
        cout << j << ". Sütun Toplamı:" << sutunToplamlar[j] << endl;
}
/* Program Çıktısı:
0. Satırı Girin:2 8
1. Satırı Girin:13 17
2. Satırı Girin:16 34
Satır Toplamları:
0. Satır Toplamı:10
1. Satır Toplamı:30
2. Satır Toplamı:50
Sütun Toplamları:
0. Sütun Toplamı:31
1. Sütun Toplamı:59

...Program finished with exit code 0
*/
```

Çok Boyutlu Diziler

Şu ana kadar gördüğümüz tek boyutlu diziler veya iki boyutlu diziler olan matrislerdir. Çok boyutlu dizilere örnek olarak En-Boy-Yükseklik içeren 3 boyutlu diziler verilebilir. Üç boyutlu dizilerde dizinin her bir elemanı bir matristir. Çok boyutlu dizilerde en içteki dizinin boyutu kimliklendirmede sağda yer alır. Aşağıdaki örnekte; çeşitli boyutlarda diziler tanımlanmıştır,

```
int ucbuyutludizi1[3][2][2];
// ucbuyutludizi1: elemanları 3 adet 2x2 matris olan üç boyutlu bir dizi
int ucbuyutludizi1[2][3][3];
// ucbuyutludizi2: elemanları 2 adet 3x3 matris olan üç boyutlu bir dizi
int ucbuyutludizi1[4][3][2];
// ucbuyutludizi3: elemanları 4 adet 3x2 matris olan üç boyutlu bir dizi
int dortbuyutludizi1[5][2][3][2];
/* dortbuyutludizi1: elemanları 5 adet olan ve her bir elemanı 2 adet 3x2 matris olan dört
boyutlu bir dizi */
```

Aşağıda üç boyutlu dizilerde bir ilk değer verme kod örneği verilmiştir;

```
#define DERINLIK 3
#define SATIR 3
#define SUTUN 3
int ucBoyutluDizi[DERINLIK][SATIR][SUTUN]={
    { {1,2,3},{4,5,6},{7,8,9} },
    { {11,12,13},{14,15,16},{17,18,19} },
    { {21,22,23},{24,25,26},{27,28,29} }
};
```

Çok boyutlu dizi örneği olarak şöyle bir örnek verilebilir; 3 farklı ders alan 10 öğrenci, her bir dersten 4 değişik not almaktadır. Bu notların ağırlıkları %10, %30, %20 ve %40'tır. Notlar rastgele 0 ile 100 arasında verilecektir. Her bir sınavın ortalaması ile her bir öğrencinin ağırlıklı not ortalamalarını bulan C programı yazınız.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
#define KACDERS 3
#define KACOGRENCI 10
#define KACDEGISIKNOT 4
int main() {
    int notlar[KACDERS][KACDEGISIKNOT][KACOGRENCI];
    int agirlik[KACDEGISIKNOT]={10,30,20,40};
        //Odevler:%10, Vize:%30, Hızlı Sınav:%20, Final:%40
    int i,j,k; //indis değişkenleri
    srand(time(NULL)); // rastgele için
    for (k=0; k< KACDERS; k++)
        for (i=0; i< KACOGRENCI; i++)
            for (j=0; j< KACDEGISIKNOT; j++)
                notlar[k][j][i]=rand()%101;
    //Her bir sınavın Ortalaması hesaplanacak:
    for (k=0; k< KACDERS; k++) {
        cout << k << ".Ders İçin:"<< endl;
        for (j=0; j<KACDEGISIKNOT; j++)  {
            float sinavToplam=0.0;
            for (i=0; i<KACOGRENCI; i++) {
                sinavToplam+=notlar[k][j][i];
            }
            cout << j << ". Sınav Ortalaması:" << sinavToplam/KACOGRENCI << endl;
        }
    }
    //Her bir Öğrencinin Ağırlıklı Notu hesaplanacak:
```

```
    for (k=0; k< KACDERS; k++) {
        cout << k << ".Ders İçin:" << endl;
        for (i=0; i<KACOGRENCI; i++) {
            float agirlikliNot=0.0;
            for (j=0; j<KACDEGISIKNOT; j++)  {
                agirlikliNot+=notlar[k][j][i]*agirlik[j]/100.0;
            }
           cout << i << ". Öğrenci Ortalaması:" << agirlikliNot << endl;
        }
    }
}
/* Program Çıktısı:
0.Ders İçin:
0. Sınav Ortalaması:58.4
1. Sınav Ortalaması:46.7
2. Sınav Ortalaması:54.6
3. Sınav Ortalaması:49.4
1.Ders İçin:
0. Sınav Ortalaması:41
1. Sınav Ortalaması:48.6
2. Sınav Ortalaması:37.1
3. Sınav Ortalaması:51.7
2.Ders İçin:
0. Sınav Ortalaması:54.7
1. Sınav Ortalaması:37.7
2. Sınav Ortalaması:43.3
3. Sınav Ortalaması:60.1
0.Ders İçin:
0. Öğrenci Ortalaması:62.2
1. Öğrenci Ortalaması:40.7
2. Öğrenci Ortalaması:60.3
3. Öğrenci Ortalaması:42.1
4. Öğrenci Ortalaması:51.9
5. Öğrenci Ortalaması:33.4
6. Öğrenci Ortalaması:58.7
7. Öğrenci Ortalaması:49.9
8. Öğrenci Ortalaması:46.1
9. Öğrenci Ortalaması:60
1.Ders İçin:
0. Öğrenci Ortalaması:49.6
1. Öğrenci Ortalaması:25.9
2. Öğrenci Ortalaması:30.5
3. Öğrenci Ortalaması:53
4. Öğrenci Ortalaması:55.4
5. Öğrenci Ortalaması:69.3
6. Öğrenci Ortalaması:48.3
7. Öğrenci Ortalaması:68.8
8. Öğrenci Ortalaması:38
9. Öğrenci Ortalaması:29
2.Ders İçin:
0. Öğrenci Ortalaması:28.3
1. Öğrenci Ortalaması:44
2. Öğrenci Ortalaması:42.9
3. Öğrenci Ortalaması:27
4. Öğrenci Ortalaması:52.9
5. Öğrenci Ortalaması:79
6. Öğrenci Ortalaması:51.5
7. Öğrenci Ortalaması:40
8. Öğrenci Ortalaması:78.1
9. Öğrenci Ortalaması:51.1
```

```
...Program finished with exit code 0
*/
```

Parametre Olarak Diziler

Parametre olarak gönderilen dizinin boyutu kadar köşeli parantez açılır ve kapatılır, ilk köşeli parantez içerisine eleman sayısını ifade eden değer yazılmaz, fakat diğerlerine eleman sayıları verilmek zorundadır. Bu durumda tek boyutlu diziler parametre olacak ise dizinin boyutu yazılmayabilir. Aşağıda dizileri parametre olarak alan fonksiyon bildirim örnekleri bulunulmaktadır;

```
void diziIsleyenFonksiyon1(int tekBoyutluDizi[],int boyut);
void diziIsleyenFonksiyon2(int pDizi[][2],int ikinciBoyut,int birinciBoyut);
void diziIsleyenFonksiyon3(int pDizi[][3][2],
                                int ucuncuBoyut,
                                int ikinciBoyut,
                                int birinciBoyut);
```

Aynı fonksiyonları aşağıdaki şekilde çağırabiliriz;

```
unsigned notlar[10]; //10 öğrenci notu barındırır.
unsigned dersNotlari[2][10]; //2 dersi alan 10 öğrenci için notlar;
unsigned bolumDersNotlari[4][2][10];
         //4 bölümde 2 dersi alan 10 öğrenci için notlar;
//...
diziIsleyenFonksiyon1(notlar,10);
diziIsleyenFonksiyon2(dersNotlari,2,10);
diziIsleyenFonksiyon3(bolumDersNotlari,4,2,10);
//...
```

Aşağıda bir diziyi okuyan, ekrana yazan ve eleman ortalamalarını hesaplayan fonksiyonlara sahip bir program verilmiştir;

```
#include <iostream>
#include <iomanip> // setw()
using namespace std;
void diziOku(int pDizi[],int pUzunluk);
void diziYaz(int pDizi[],int pUzunluk);
float diziOrtalama(int pDizi[],int pUzunluk);
int main() {
  int dizi[5];
  float ortalama;
  diziOku(dizi,5); /* fonsiyon geri dönüş değeri olmadığından
                      bir değişkene atanmıyor! */
  diziYaz(dizi,5); /* fonsiyon geri dönüş değeri olmadığından
                      bir değişkene atanmıyor! */
  ortalama=diziOrtalama(dizi,5); /* fonksiyon geri dönüş değeri
                                    bir değişkene atanıyor. Atandığı değişkenin
                                    tipi ile fonksiyonun geri dönüş tipi
                                    aynı olmalı! */
  cout << "Dizi Ortalaması:" <<ortalama ;
  return 0;
}
void diziOku(int pDizi[],int pUzunluk){
  int sayac;
  cout << pUzunluk << " Elemanlı Dizi Okunacaktır:";
  for (sayac=0; sayac < pUzunluk; sayac++)
     cin >> pDizi[sayac];
}
void diziYaz(int pDizi[],int pUzunluk){
  int sayac;
  cout << pUzunluk << " Elemanlı Dizi:" << endl;
  for (sayac=0; sayac < pUzunluk; sayac++)
```

```
     cout << setw(5) << pDizi[sayac] ;
  cout << endl;
}
float diziOrtalama(int pDizi[],int pUzunluk){
  int sayac;
  float toplam=0;
  for (sayac=0; sayac < pUzunluk; sayac++)
     toplam+=pDizi[sayac];
  return toplam/pUzunluk;
}
/* Program Çıktısı:
5 Elemanlı Dizi Okunacaktır:10 20 30 40 50
5 Elemanlı Dizi:
   10   20   30   40   50
Dizi Ortalaması:30

...Program finished with exit code 0
*/
```

Foreach Talimatı

Yalnızca bir dizideki öğeler arasında döngü oluşturmak için kullanılan `foreach` döngüsü (foreach loop) aralık tabanlı döngü (range based loop) olarak da bilinir. Bu döngü ileride göreceğimiz *Konteyner Şablonları* başlığındaki veri yapılarıyla da kullanılır. Aşağıdaki gibi yazılır;

```
for (veritipi dizi-elemanını-temsil-eden-değişken : dizi-kimliği)
       döngü-kodu;
```

Buna örnek olarak aşağıda bir dizi programı örneği verilebilir;

```
float dizi[10] = {10.0, 20.2, 30.3, 40.4, 50.5, 60.6, 70.7, 80.8, 90.9, 100.0};

for (float f : dizi) {
  cout << c << ", ";
}
```

GÖSTERİCİLER VE REFERANSLAR

Gösterici nedir? Niçin İhtiyaç Duyarız?

Şu ana kadar gördüğümüz `char`, `int`, `long`, `float`, `double` tipli değişkenler ve bunlara ait diziler, değer tipler (value type) olarak adlandırılır. Çünkü kimliklendirilen değişken tutulacak değeri içerir.

Değer tiplerin yanı sıra değişkenlerin adreslerini tutan değişkenlere de ihtiyaç duyarız. İşte adres tutan değişkenlere gösterici tipler (pointer type) adını veririz. Gösterici tipler, gösterdiği yerdeki verileri değiştirmek için de kullanılır. Bu nedenle gösterici tipler tanımlanırken gösterdiği yerdeki verinin tipine göre tanımlanırlar. Bütün gösterici tipler bellekte aynı miktarda yer kaplar. Çünkü hepsi adres tutar. Ancak işaret ettikleri yerdeki veriler, verinin tipine bağlı olarak bellekte farklı miktarda yer kaplar. Bütün bu tanımlamaların ışığında göstericileri, vatandaşların ikametlerini gösteren adres numaraları olarak düşünebiliriz.

Verilerin tipine göre gösterici tanımlandığından göstericiler türetilmiş tiplerdir (derived type). Gösterici tiplere ihtiyaç duyulmasının sebebi hız ve esnekliktir. Genel olarak birkaç madde ile sıralayacak olursak;

- Belleğin istenilen bölgesine erişim sağlamak için göstericiler kullanılır. Günümüz modern dillerinde referans tipler kullanılır, ancak referans tiplerde gösterilen adreste bir nesnenin bulunduğu garanti edilir göstericiler gibi her bellek bölgesine erişilemez.
- Bazen çalıştırma anında yeni bellek gölgelerine ihtiyaç duyarız bu durumda göstericiler gereklidir. Dinamik veri yapıları (dynamic data structure) göstericiler yardımıyla oluşturulup yönetilir.
- Fonksiyonlarda yerel değişkenler ve argümanlar yığın belleğe itilir ve fonksiyondan geri dönüldüğünde bu değişkenler eski haline yığın bellekten geri çekilerek çevrilir. Fonksiyonun yerel değişken ya da argümanlardan birini değiştirmesi istendiğinde fonksiyona argüman olarak değişkenin adresi verilir. Böylece fonksiyon içinde gösterici olarak işlem gören argümanlar fonksiyondan geri döndüğünde değerleri değişmiş olur.

Göstericilere olan ihtiyacı bir başka örnekle de açıklayabiliriz; Bir ormandaki ağaçları boylarına göre sıralamaya kalkarsak her birini yer değiştirme yöntemiyle sıralamak oldukça külfetli ve zaman alana bir işlemdir. Halbuki ağaçlara numara vererek ve bu numaraların karşısına uzunluklarını yazacağımız bir liste oluşturulduğunda sadece numaraları sıralamak oldukça kolay ve zahmetsiz bir işlemdir. İşte buradaki numaraları tutan değişkenler göstericilerdir.

Gösterici Tanımlama

Göstericiler, gösterdiği yerdeki verinin tipine göre tanımlanırlar.

Gösterici tanımlanırken veri tipi izleyen başvuru kaldırma işleci (dereference operator) olan yıldız (`*`) kullanılır. Başvuru kaldırma terimi, gösterici tarafından tutulan bellek adresinde saklanan değere erişmeyi de ifade eder.

```
veritipi* gostericikimligi;
```

Göstericilere değer atatama aşağıdaki şekilde yapılır;

```
gostericikimligi = &degiskenkimliği;
```

Bir değişkenin adresinin referans işleci (`&`) ile elde edildiğini öğrenmiştik. Aşağıda göstericilere ilişkin kod örnekleri verilmiştir;

```
char* ptrToChar;  /* tuttuğu adreste char tipinde veri olan
                     ptrToChar kimlikli göstericisi tanımlandı. */
int i=10;
int* ptrToInt=&i; /* tuttuğu adreste int tipinde veri olan
                     ptrToInt kimlikli göstericisi tanımlandı ve
                     ilk değer olarak i değişkeninin adresi atanıyor */
```

```
*ptrToInt =20; /* ptrToInt göstericisinin gösterdiği adresteki tamsayı
                  değeri 20 yaptık. ptrToInt göstericisi, i değişkeninin
                  adresini tuttuğundan i değişkeninin değeri de 20 olmuştur */
```

Bazı durumlarda göstericilerin veya gösterdiği değerlerin sabit olması gerekebilir. Üç durum ortaya çıkabilir;

- Gösterilen değerin sabit olması: Bu durumda gösterici tanımlaması aşağıdaki gibi yapılır;

```
int main() {
   int i=10;
   const int* ptrToi=&i; // değerin sabit olması durumunda gösterici tanımı
   *ptrToi=20; /* HATA! pi göstericisinin tuttuğu adresteki tamsayı
                  değeri 20 yapılamaz. */
}
```

- Göstericinin işaret ettiği adresin sabit olması: Kısaca göstericinin sabit olması durumunda tanımlama aşağıdaki gibi yapılır;

```
int main() {
   int i=10;
   int* const ptrToi=&i; /* göstericinin sabit olması durumunda gösterici tanımı.
                            Bu tür tanımlamada adres ilk değer (initial value)
                            olarak mutlaka verilmelidir. */
   int j=30;
   *ptrToi=20; // pi göstericisinin tuttuğu adresteki tamsayı değeri 20 olur.
   ptrToi =&j; //HATA! : pi göstericisinin tuttuğu adres değiştirilemez!
}
```

- Hem göstericinin hem de değerin sabit olması durumu:

```
int main() {
   int j=30;
   const int* const ptr=&j; /* Hem gösterici, hem de gösterilen veri sabit
                               Bu tür tanımlamada da adres ilk değer olarak
                               mutlaka verilmelidir. */
}
```

Atanmış kesin bir adresiniz yoksa, bir gösterici değişkenine `NULL` değeri atamak her zaman iyi bir uygulamadır. İlk değeri olmayan gösterici tanımlamak yerine, ilk değeri sıfır olan göstericiler tanımlanmak doru bir yöntemdir. Bu durumda ilk değer `nullptr` olarak atanır ve bu göstericiye, NULL gösterici (NULL pointer) adı verilir. Aşağıdaki gibi kullanılır;

```
veritipi* gostericikimligi = nullptr;
gostericikimligi = nullptr;
```

Örnek kod aşağıda verilebilir;

```
#include <iostream>
using namespace std;
int main(){
 int* ptr = nullptr;
 cout <<"ptr göstericisinin tuttuğu adres:" << ptr << endl;
 int agirlik=80;
 ptr=&agirlik;
 cout << "ptr göstericisinin tuttuğu adres:" << ptr << " ve değeri:" << *ptr << endl;
}
/* Program Çıktısı:
ptr göstericisinin tuttuğu adres: (nil)
ptr göstericisinin tuttuğu adres: 0x7ffdedc33dec ve değeri: 80

...Program finished with exit code 0
*/
```

Bir gösterici bir başka göstericinin adresini tutabilir. Buna çifte gösterici (double pointer) adı verilir. Aşağıda buna ilişkin bir örnek bulunmaktadır;

```
#include <iostream>
using namespace std;
int main(){
   int var = 10;
   int* intptr = &var;
   int** ptrptr = &intptr;  //doble pointer

   cout << "var:" << var << " &var:" << &var << endl;
   cout << "inttptr:" <<  intptr  << " *inttptr:" << *intptr << endl;

   cout << "var:" << var << " *intptr:" << *intptr << endl;
   cout << "ptrptr:" << ptrptr << " &ptrtptr:" << &ptrptr << endl;
   cout << "&intptr:" << &intptr << " *ptrptr :" << *ptrptr << endl;
   cout << "var:"<< var<< " *intptr:" << *intptr<<" **ptrptr:" << **ptrptr;
}
/* Program Çıktısı:
var:10 &var:0x7ffe0e839f74
inttptr:0x7ffe0e839f74 *inttptr:10
var:10 *intptr:10
ptrptr:0x7ffe0e839f78 &ptrtptr:0x7ffe0e839f80
&intptr:0x7ffe0e839f78 *ptrptr :0x7ffe0e839f74
var:10 *intptr:10 **ptrptr:10

...Program finished with exit code 0
*/
```

Gösterici Aritmetiği

Bilgisayarların tasarımından ötürü işlemciye bağlı belleklerde her bir ayrı adreslenir. Birkaç bayt(`char`) veri içeren bir bellek bölgesini işaret eden bir gösterici tanımlandığında göstericinin işaret ettiği adres, bir sonraki baytı göstermesi istendiğinde gösterici adresi 1 artar.

```
char dizi1[5]={'I','L','H','A','N'}; /* Bellekte Art arda 5 bayt yer ayrılmış dizi. */
char* pc=&dizi1[0]; /* pc göstericisi dizinin ilk elemanını gösteren adres (65FDE0)
varsayıldı. */
*pc='X';         //dizi {'X','L','H','A','N'} oldu.
pc++;            /* pc göstericisi bir sonraki baytı gösterir: (65FDE1) adres 1 artırıldı */
*pc='Y';         // dizi {'X','Y','H','A','N'} oldu.
pc+=2;           /* pc göstericisi 2 sonraki baytı gösterir: (65FDE2) adres 2 artırıldı. */
*pc='Z';         // dizi {'X','Y','H','A','Z'} oldu.
```

Benzer şekilde birkaç tamsayı (`int`) veri içeren bir bellek bölgesini işaret eden bir gösterici tanımlandığında göstericinin işaret ettiği adres, bir sonraki tamsayıyı göstermesi istendiğinde gösterici adresi 4 artar. Çünkü tamsayılar bellekte 4 bayt yer kaplar.

```
int dizi2[5]={2,0,3,10,2}; /* Bellekte art arda 5 tamsayı (5x4bayt) yer ayrılmış dizi. */
int* pi=&dizi2[0];  /*pi göstericisi dizinin ilk elemanını gösteren  adres (65FDE0)
varsayıldı. */
*pi=-1;          //dizi {-1,0,3,10,2} şekline döndü.
pi++;            /* pi göstericisi ikinci elemanını gösterir: (65FDE4) adres 4 artırıldı. */
*pi=-2;          //dizi {-1,-2,3,10,2} şekline döndü.
pi+=2;           /* pi göstericisi iki sonraki elemanını gösterir: (65FDEC) adres 8 artırıldı.
*/
*pi=-3;          //dizi {-1,-2,3,10,-3} şekline döndü.
```

Görüldüğü üzere göstericilerin üzerinde işleçler (operator) işlem yaparken göstericinin işaret ettiği verinin bellekte kapladığı yer dikkate alırlar. Göstericiler ile sadece aşağıdaki işleçler çalışır, diğer işleçlerde işlenen (operand) olarak kullanılmaz.

– Aritmetik ve atama işleçlerinden `++`, `--`, `+`, `-`, `+=` ve `-=`
– Karşılaştırma işleçlerinden `<`, `>` ve `==`

Gösterici Dizileri

Göstericiler de dizi olarak tanımlanabilir. Aşağıda buna ilişkin bir örnek verilmiştir;

```
#include <iostream>
using namespace std;
int main() {
  int i = 10, j=20;
  float f=1.0;
  int k=30, l=40;

  int* ptr[4]; /* int gösteren 4 gösteri içeren bir dizi tanımlandı */

  /* Göstericilere ilk değer veriliyor*/
  ptr[0] = &i; // Birinci eleman i değişkeninin adresini
  ptr[1] = &j; // İkini eleman j değişkeninin adresini
  ptr[2] = &l; // Üçüncü eleman l değişkeninin adresini
  ptr[3] = &k; // Dördüncü eleman k değişkeninin adresini

  // Değerlere Ulaşma
  int indis;
  for (indis = 0; indis < 4; indis++)
     cout << "ptr[" << indis <<"] ile gösterilen değer:" << *ptr[indis] << endl;
}
/* Program çalıştırıldığında;
ptr[0] ile gösterilen değer:10
ptr[1] ile gösterilen değer:20
ptr[2] ile gösterilen değer:40
ptr[3] ile gösterilen değer:30

...Program finished with exit code 0
*/
```

Parametre Olarak Göstericiler

Fonksiyonlar çağrılırken argüman olarak giren değerlerin değişmeyeceği *Fonksiyon Çağırma Süreci* başlığında anlatılmıştı. Fonksiyona parametre olarak giren değerler, fonksiyon bloğu içinde değişse bile fonksiyondan geri dönülürken aynı çağrı ortamını sağlamak için kaybolurlar.

```
#include <iostream>
using namespace std;
void degistir(int,int);
int main(){
   int a = 10, b = 20;
   degistir(a, b);
   cout << "Değiştir Sonrası a:" << a<< ", b:" << b << endl;
   //Çıktı: Değiştir Sonrası a:10, b:20
}
void degistir(int pX, int pY) {
   int z = pX;
   pX=pY;
   pY=z;
}
```

Fonksiyona giren argümanları gösterici olarak tanımlarsak çağrı ortamına geri dönüldüğünde yerel değişkenler de değişmiş olur. Çünkü fonksiyona değişken adresleri değer olarak girmiştir. Buna ilişkin örnek aşağıda verilmiştir;

```
#include <iostream>
```

```
using namespace std;
void degistir(int,int);
void degistir2(int*, int*);
int main(){
   int a = 10, b = 20;
   degistir(a, b);
   cout << "Değiştir Sonrası a:" << a << ", b:" << b << endl;
   //Çıktı: Değiştir Sonrası a:10, b:20
   degistir2(&a, &b);
   cout << "Değiştir2 Sonrası a:" << a << ", b:" << b << endl;
   //Çıktı: Değiştir2 Sonrası a:20, b:10
}
void degistir(int pX, int pY) {
   int z = pX;
   pX=pY;
   pY=z;
}
void degistir2(int* pX, int* pY){
   int z = *pX;
   *pX=*pY;
   *pY=z;
}
```

Göstericilerin Dizileri İşaret Etmesi

Çoğu durumda, bir dizi yardımıyla gerçekleştirdiğimiz işleri gösterici ile de gerçekleştirilebiliriz. Genellikle dizilerin kendisi yerine ilk elemanlarını işaret eden göstericiler kullanırız.

```
#include <iostream>
using namespace std;
int main () {
   int dizi[5] = {10, 20, 30, 40, 50};
   int* ptr = dizi;  // int* ptr=& dizi[0]; olarak da kodlanabilir.
   cout << "Dizi elemanlarına gösterici ile erişim:" << endl;
   for(int indis = 0; indis < 5; indis++)
      cout << "dizi[" << indis << "]:" << *(ptr+indis) << endl;
}
```

Fonksiyonlara dizileri gösteren göstericileri parametre olarak verebiliriz;

- Diziler parametre olarak kullanılırken adres operatörü kullanılmaz. Diziler, gösterici gibi davranırlar. Yani bir dizinin adı, dizinin ilk öğesinin adresi gibi davranır.

```
float notlar[10];
float notlarOrtalamasi(float*); //prototype
//...
float ort=notlarOrtalamasi(notlar); // yada notlarOrtalamasi(&notlar[0]);
//...
float matris[4][3];
float defetminant(float**,int,int); //prototype
//...
float det=defetminant(matris,4,3);
//...
```

- Gösterici gibi davrandıklarından dizi elemanları fonksiyon içinde yine dizi gibi kullanılabilir.
- Gösterici gibi davrandıklarından dizi elemanları fonksiyon içinde değiştirilebilir. Fonksiyondan dönüldüğünde elemanlar değişiklik yapılmış şekliyle işlemler devam eder.

Aşağıda öğrencilerin notlarına ilişkin işlemler yapılan bir program verilmiştir.

```
#include <iostream>
using namespace std;
#define OGRENCISAYISI 10
```

```
float notlarOrtalamasi(float*);
void notlariArtir(float*);
int main(){
   float notlar[OGRENCISAYISI]= { 55.0,60.0,70.0,35.0,30.0,50.0,65.0,90.0,95.0,100.0 };
   float toplam=notlarOrtalamasi(notlar);
   cout << "Notlar Ortalaması:" << toplam << endl;
   notlariArtir(notlar);  // Bu fonksiyonla dizi elemanları değiştiriliyor
   int indis;
   for (indis=0; indis<OGRENCISAYISI; indis++)
      cout << "Not[" << indis << "]=" << notlar[indis] << endl;
   return 0;
}
float notlarOrtalamasi(float* pNotlar){
   int indis;
   float top=0.0;
   for (indis=0; indis<OGRENCISAYISI; indis++)
      top+=pNotlar[indis]; // Gösterici dizi gibi kullanılıyor
   return top/OGRENCISAYISI;
}
void notlariArtir(float* pNotlar) {
   int indis;
   for (indis=0; indis<OGRENCISAYISI; indis++)
      if (pNotlar[indis]<=90) //Göstericinin gösterdiği değerler değiştiriliyor
          pNotlar[indis]+=10.0;
};
/* Program Çıktısı:
Notlar Ortalaması: 65.0
Not[0]=65.0
Not[1]=70.0
Not[2]=80.0
Not[3]=45.0
Not[4]=40.0
Not[5]=60.0
Not[6]=75.0
Not[7]=100.0
Not[8]=95.0
Not[9]=100.0

...Program finished with exit code 0
*/
```

Fonksiyonlardan Bir Diziyi Geri Döndürme

C++ dilinde bir fonksiyonun geri dönüş değeri olarak tüm bir dizi verilemez! Ancak, bir dizinin göstericisini fonksiyondan geri dönüş değeri olarak döndürebilirsiniz. Burada dikkat edilmesi gereken fonksiyondan çıkılınca dizinin bellekten kaldırılmamasıdır. Aşağıda bir tamsayı dizi göstericisini döndüren bir fonksiyon örnekleri verilmiştir;

```
int* tekBoyutluDiziDondur() {
    /*... */
}
int** ikiBoyutluDiziDondur() {
    /*... */
}
```

Yerel değişkenlerin yığın bellekte (stack segment) tutulduğu ve fonksiyondan geri dönünce fonksiyon yerelindeki değişkenlerin kaybolduğu *Çağrı Kuralı*

Çağrı kuralı (calling convention), bir fonksiyon çağrısıyla karşılaşıldığında argümanların fonksiyona hangi sıraya göre iletileceğini belirtir. İki olasılık vardır; Birincisi argümanlar soldan başlanarak sağa

doğru fonksiyona geçirilir. İkincisi ise C dilinde kullanılır ve argümanlar sağdan başlanarak sola doğru fonksiyona geçirilir.

```
#include <iostream>
using namespace std;
int topla(int, int, int);
int main () {
    int toplam;
    toplam=topla(10,20,30);
    cout << "toplam:" << toplam << endl;
}
int topla(int x, int y, int z) {
    cout << "x:"<< x << " y:" << y  << " z:" << z<< endl;
    return x+y+z;
}
/* Program çalıştırıldığında:
x:10 y:20 z:30
toplam:60
*/
```

Yukarıdaki örnek incelendiğinde argümanların hangi sırada fonksiyona geçirildiğinin bir önemi yoktur. Ancak aşağıdaki verilen örnekteki durumu inceleyelim;

```
#include <iostream>
using namespace std;
int topla(int, int, int);
int main () {
    int a = 1,toplam;
    toplam = topla(a, ++a, a++);
    cout << "toplam:" << toplam << endl;
}
int topla(int x, int y, int z) {
    cout << "x:"<< x << " y:" << y  << " z:" << z<< endl;
    return x+y+z;
}
/* Program çalıştırıldığında:
x:3 y:3 z:1
toplam:7
*/
```

Bu kodun çıktısı `1 2 3` olarak beklenir ancak çağrı kuralından ötürü çıktı `3 3 1` olur. Bunun nedeni C++ dilinin çağrı kuralının sağdan sola olmasıdır. Bunu şöyle açıklayabiliriz;

6. Fonksiyona sağdan ilk argüman olarak a değişkeni değeri 1 geçirilir.
7. Ardından a++ ifadesi ile a değişkeni 2'ye yükseltilir.
8. Sonra ++a ifadesi ile a değişkeni 3'e yükseltilir.
9. Fonksiyona ikinci argüman olarak 3 geçirilir.
10. Fonksiyona son olarak a'nın son değeri olan 3 geçirilir.

Depolama Sınıfları başlığında anlatılmıştı. Bu nedenle bir fonksiyon içinde tanımlanan bir dizinin göstericisi geri döndürülecek ise `static` olarak tanımlanır.

```
#include <iostream>
#include <iomanip> //setw()
#include <ctime>
using namespace std;
#define BOYUT 10
int* rastgeleOgrenciNotlari( ) {
    static int  notlar[BOYUT];
    for (int i = 0; i < BOYUT; ++i)
        notlar[i] = rand()%101;
    return notlar;
}
```

```
void notlariYaz(int* pNotlar,int pUzunluk) {
    cout << "Öğrenci Notları:" << endl;
    for (int i = 0; i < pUzunluk; ++i)
        cout << setw(4) << pNotlar[i];
    return;
}
int main () {
   int *notlar;
   srand(time(nullptr));
   notlar = rastgeleOgrenciNotlari();
   notlariYaz(notlar,BOYUT);
}
/* Program Çıktısı:
Öğrenci Notları:
  30  31  13  40  41  37  34  83  93  76

...Program finished with exit code 0
*/
```

Dinamik Hafıza

C++ dilinde bir fonksiyon bloğu içerisinde tanımlanan tüm değişkenler yığın bellek (stack segment) üzerinde kaplayacaktır. Program çalıştığında bir değişkene belleği dinamik olarak tahsis etmek için öbek bellek (heap segment) kullanılır. Dinamik bellek tahsisi için new işlecini (new operator) kullanırız. Bu işleç, dinamik dizi için öbek bellekte yer tahsis edilip tahsis edilen bellek bölgesinin ilk baytının adresini göstericiye atar. Tahsis edilen bellek bölgesini serbest bırakmak için ise `delete` işleci (delete operator) kullanılır.

Aşağıda bir `float` değişkene dinamik olarak yer tahsisi örneği verilmiştir;

```
#include <iostream>
using namespace std;
int main(){
    float* ptrFloat=nullptr;
    ptrFloat=new float;
    if (ptrFloat!=nullptr) {
        *ptrFloat=4.5;
        cout << *ptrFloat << endl;
    } else
        cout << "Bellek tahsisi yapılamadı!" << endl;
    delete ptrFloat;
}
```

C++ dilinde diziler de dinamik olarak göstericiler yardımıyla oluşturulur. Dinamik bellek tahsisi, çalışma zamanı sırasında bellek tahsis etmemize olanak tanır. Bir dizideki dinamik tahsis, özellikle bir dizinin boyutu derleme zamanında bilinmediğinde ve çalışma zamanı sırasında belirtilmesi gerektiğinde faydalıdır. Bir diziyi dinamik olarak tahsis etmek için;

- Tahsis edilen dizinin ilk elemanının adresini depolayacak bir gösterici tanımlanır.
- Sonra, belirli bir veri tipindeki bir diziyi barındıracak dizi için `new` işleci ile bellek alanını tahsis edilir.
- Bu tahsisi yaparken, kaç öğe içerebileceğini belirten dizinin boyutunu belirtilir. Bu belirtilen boyut, tahsis edilecek bellek miktarını belirler.

Dizi uzunluğu tamsayı olacak şekilde dinamik dizi aşağıdaki gibi tanımlanır;

```
veritipi* diziGöztericiKimliği= new veritipi[diziUzunluğu];
```

Aşağıda çalıştırma anında dinamik olarak oluşturtulan bir dizi örneği verilmiştir;

```
#include <iostream>
using namespace std;
int main() {
```

```
    // Çalıştırma Anında Oluştutulacak Dizi Boyutu Soruluyor
    cout << "Dizinin Boyutunu Girin: ";
    int size;
    cin >> size;

    int* arr = new int[size]; // Diziye dinamik bellek (heap) tahsis ediliyor.

    for (int i = 0; i < size; i++) // Bir döngü ile dizi elemanlarına değer atanıyor
        arr[i] = i * 2;

    cout << "Dinamik Dizi Elemanları: " << endl;
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;

    delete[] arr; // dizi bellekten kaldırılıyor. Tahsis edilen bellek serbest bırakılıyor.
}
/*Program çalıştırıldığında:
Dizinin Boyutunu Girin: 12
Dinamik Dizi Elemanları:
0 2 4 6 8 10 12 14 16 18 20 22

...Program finished with exit code 0
*/
```

Çok boyutlu dizilere de dinamik olarak bellek tahsisi yapılabilir;

```
double** pvalue  = nullptr;   // Göstericiye ilk değer olarak nullptr veriliyor
pvalue  = new double [3][4];  // 3x4 buyutlu martis için yer tahsisi yapılıyor.
//...
delete[] pvalue; // tahsis edilen bellek serbest bırakılıyor
```

Çöp Toplama

Çöp toplama (garbage collection), dinamik bellek yönetimi söz konusu olduğunda önemli bir kavramdır. C++ dilinde geliştiriciler belleği manuel olarak yönetmelidir, bu hem güçlendirici hem de göz korkutucu olabilir. Otomatik çöp toplama özelliğine sahip dillerin aksine, C++ programcıların `new` ile belleği tahsis etmesini ve `delete` kullanarak ve tahsis edilmiş belleği serbest bırakmasını gerektirir.

C++'da bellek yönetimi yığın bellek (stack memory) ve öbek belleği (heap memory) belleği kavramları etrafında döner. Yığın belleği, yerel değişkenlerin depolandığı yerdir ve boyutu sınırlıdır. Bu belleğin boyutu derleyici tarafından belirlenir ve otomatik olarak yönetilir. Ancak öbek (heap) belleği, dinamik bellek tahsisinin gerçekleştiği yerdir. Burada, bellek çalışma zamanında tahsis edilir ve bu da daha fazla esneklik sağlar ancak aynı zamanda dikkatli bir yönetim gerektirir.

`new` İşleci kullandığınızda, öbek bellek üzerinde bir miktar bellek tahsis edilir. Bu bellek, `delete` işleci kullanarak açıkça serbest bırakana kadar tahsis edilmiş olarak kalır. Tahsis edilen belleği serbest bırakmamak, artık ihtiyaç duyulmayan belleğin tahsis edilmiş olarak kalmasına ve kaynakların verimsiz kullanımına yol açan bellek sızıntılarına (memory leak) yol açabilir. Buna karşılık, hala kullanımda olan belleği silerseniz, tanımlanmamış davranışa ve program çökmelerine yol açabilir.

C++ dilinde etkili çöp toplama için yığın (stack) ve öbek (heap) belleği arasındaki dengeyi anlamak esastır. Belleği düzgün bir şekilde yönetmek uygulama performansını artırabilir, bellek sızıntılarını önleyebilir ve genel kararlılığı iyileştirebilir.

C++ dilinde bellek yönetimi çoğunlukla manuel olarak programcı tarafından yapılır. Bu, programcıların belleği ayırma ve ayırmayı kaldırma sorumluluğunu üstlenmesi gerektiği anlamına gelir. `new` ve `delete` işleçlerini doğru kullanmak, uygulamanızın sorunsuz çalışmasını sağlamak için hayati önem taşır.

```
#include <iostream>
int main() {
```

```
    int* ptr = new int;
    *ptr = 61;

    std::cout << "Value: " << *ptr << std::endl;
    delete ptr;
}
```

Yukarıdaki örnekte `ptr` göstericisine öbek bellekte `new` ile yer ayrılmış, bellekteki değer `61` olarak değiştirilmiş ve `delete` ile bu bellek bölgesi serbest bırakılmıştır. Burada `delete` unutulmuş olsaydı, bellek ayrılmış olarak kalır ve bu da bellek sızıntısına yol açardı. Bu manuel yaklaşım, programcının dikkat ve disiplinini gerektirir çünkü bu ayrıntıların gözden kaçırılması önemli sonuçlara yol açabilir.

Manuel bellek yönetimi esneklik sağlarken, aynı zamanda birkaç tuzağı da beraberinde getirir. Yaygın bir sorun, tahsis edilen belleğin `delete` kullanılmasının unutulması ve dolayısıyla asla serbest bırakılmaması durumunda oluşan bellek sızıntılarıdır. Bir diğer tuzak da sarkan göstericilerdir (dangling pointers). Bu göstericiler, tahsisi kaldırılmış belleğe hala başvurduğunda meydana gelir. Bu tür belleğe erişim, tanımlanmamış davranışa, çökmelere veya veri bozulmasına yol açabilir;

```
#include <iostream>
int main() {
    int* ptr = new int(61);
    delete ptr;
    std::cout << *ptr << std::endl;
    return 0;
}
/* Program Çıktısı:
Segmentation fault
*/
```

Bu örnekte, `delete` işleci ile bellek serbest bırakıldıktan sonra, `ptr` sarkan gösterici haline gelir. Serbest bırakılmış belleğe erişmeye çalışmak bir segmentasyon hatasıyla (segmentation fault) sonuçlanır. Bu tür sorunları önlemek için, silme işleminden sonra işaretçileri geçersiz kılmak çok önemlidir.

C++'da manuel bellek yönetimiyle ilişkili yaygın tuzaklardan kaçınmak için izleyebileceğiniz birkaç iyi uygulama vardır:

- her zaman `new` ve `delete` işleçlerini eşleştirin: Belleği `new` kullanarak ayırdığınızda, `delete` ile onu serbest bıraktığınızdan emin olun. Bu, bellek sızıntılarını önlemeye yardımcı olur.
- Akıllı göstericiler kullanın: C++ 11 ile hayatımıza belleği otomatik olarak yöneten `std::unique_ptr` ve `std::shared_ptr` gibi akıllı göstericiler girdi. Bu göstericiler ihtiyaç duyulmadığında belleği otomatik olarak silerek bellek sızıntılarını ve sarkan göstericileri (dangling pointer) işaretçileri önlemeye yardımcı olurlar. *Akıllı Göstericiler* başlığını inceleyiniz.
- Göstericilere her zaman `nullptr` ile ilk değer ver: Bu uygulama sarkan göstericileri önlemeye yardımcı olur.

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<int> ptr = std::make_unique<int>(61);
    std::cout << "Value: " << *ptr << std::endl;
}
```

Referanslar

Bir değişken referans olarak tanımlandığında, var olan bir değişken için alternatif bir isim haline gelir. Bir değişken, bildirime '&' eklenerek referans olarak bildirilebilir. Yani, bir referans değişkenini başka bir değişkene referans görevi görebilen bir değişken türü olarak tanımlayabiliriz.

```
veri-tipi &referans-kimliği = varolan-bir-değişken;
```

Buradaki referans işleci bir değişkenin veya herhangi bir belleğin adresini belirtmek için kullanılır. Aşağıda buna ilişkin örnek bir program verilmiştir;

```
#include <iostream>
using namespace std;
int main(){
    int x = 10;
    int& ref = x; // ref ile var olan x değişkenine referans tanımlanmıştır.
    ref = 20; // x değişkeninin yeni değeri 20 olmuştur.
    cout << "x = " << x << endl; // x = 20
    x = 30; // x değişkeninin yeni değeri 30 olmuştur
    cout << "ref = " << ref << endl; // ref = 30
}
```

Fonksiyonlara parametre geçirme konusunda yaygın olarak kullanılan yöntemler değerle geçme (pass by value) ve referansla geçmedir (pass by reference).

- Değerle geçme, bir fonksiyonun parametrelerinin değerlerinin bir kopyası oluşturularak fonksiyon icra edilmesidir. Bu durumda fonksiyondan geri dönüldüğünde girdi olarak kullanılan değişkenler değişmez.
- Referansla geçme, değişkenlerin kendisi değil de referanslarının parametre olarak fonksiyona geçirilmesidir. Böylece parametre olarak kullanılan girdi değişkenlerinin fonksiyon içerisinde değiştirilmesine izin verilir. Çünkü değişkenin değeri değil adresi parametre olarak fonksiyona geçirilmiştir.

Göstericilerle bunu yapabiliriz ancak bu durumda kod okunabilirliğini azalır ve programcının hata yapma olasılığı artar.

```
#include <iostream>
using namespace std;
void degistir(int,int);
void degistir2(int&, int&);
int main(){
   int a = 10, b = 20;
   degistir(a, b);
   cout << "Değiştir Sonrası a:" << a << ", b:" << b << endl;
   //Çıktı: Değiştir Sonrası a:10, b:20
   degistir2(a, b);
   cout << "Değiştir2 Sonrası a:" << a << ", b:" << b << endl;
   //Çıktı: Değiştir2 Sonrası a:20, b:10
}
void degistir(int pX, int pY) {
   int z = pX;
   pX=pY;
   pY=z;
}
void degistir2(int& pX, int& pY){
   int z = pX;
   pX=pY;
   pY=z;
}
```

Referanslar ile göstericileri karşılaştırırsak;

- Göstericiler herhangi bir zamanda NULL gösterici (NULL pointer) olabildiği gibi başka bir nesneyi işaret edilebilir. Bir referans tanımlandığı anda bir değişken ile ilk değer verilmelidir. Göstericiler tanımlandıktan sonra herhangi bir zamanda ilk değer verilebilir ya da gösterdiği değişken değiştirilebilir.
- Hiçbir değişkeni işaret etmeyen NULL referanslarınız olamaz! Her zaman bir referansın meşru bir değişkene bağlı olduğunu varsayabilmelisiniz.

– Bir referans imal edilen bir nesneyi işaret ediyorsa, başka bir nesneyi göstermek üzere referansı değiştirilemez!

Göstericiler yerine referanslar kullanmak kodumuzun okunmasını ve bakımını daha kolay hale getirilebilir. Bir C++ fonksiyonu, bir gösterici döndürdüğü gibi bir referansı da döndürebilir.

Bir fonksiyon bir referans döndürdüğünde, dönüş değerine örtük bir gösterici döndürür. Böylece atama ifadesinin sol tarafında bir fonksiyon kullanılabilir hale gelir.

```
#include <iostream>
#include <ctime>
using namespace std;

float dizi[5] = {10.0, 20.0, 30.0, 40.0, 50.0};
float& indistekiDeger( int indis ) {
   return dizi[indis];
}
int main () {

   cout << "Değiştirmeden Önce Dizi" << endl;
   for ( int i = 0; i < 5; i++ ) {
      cout << "dizi[" << i << "] = ";
      cout << dizi[i] << endl;
   }
   indistekiDeger(1) = 100.0;
   indistekiDeger(3) = 200.0;
   cout << "Değiştirmeden Sonra Dizi" << endl;
   for ( int i = 0; i < 5; i++ ) {
      cout << "dizi[" << i << "] = ";
      cout << dizi[i] << endl;
   }
}
/* Program Çıktısı:
Değiştirmeden Önce Dizi
dizi[0] = 10
dizi[1] = 20
dizi[2] = 30
dizi[3] = 40
dizi[4] = 50
Değiştirmeden Sonra Dizi
dizi[0] = 10
dizi[1] = 100
dizi[2] = 30
dizi[3] = 200
dizi[4] = 50

...Program finished with exit code 0
*/
```

Bir referans döndürürken, referans alınan nesnenin kapsam (scope) dışına çıkmamasına dikkat edilmesi gerekir;

```
int& fonksiyon() {
   int q;
   // return q; /* Derleme hatası olur.
                   Çünkü fonksiyon dışında bu değişken bellekte yoktur.*/
   static int x;
   return x; /* x değişkeni tüm program süresince hayatta olduğundan
                bu kod güvenlidir. */
}
```

NESNE YÖNELİMLİ PROGRAMLAMA

Nesne Yönelimli Programlama Paradigması

Yapısal programlamada talimatlar (statement) art arda koda yazılarak programlama yapılır ve programların neler yaptığı bu talimatlar izlenerek anlaşılabilir. Talimatların zincirin halkaları gibi birbirinin peşi sıra yazılarak yapılan programlamaya paradigması adı verilir. Bu paradigmaya sahip diller, yazılımı yapılacak sürece ilişkin nelerin yapılacağını değil, işin nasıl yapılacağını belirtirler

Emreden programlama (imperative programming) bir örneği olan yapısal programlamada talimatlar (statement) art arda koda yazılarak programlama yapılır. 1980'li yıllara yazılım ihtiyaçları kadar ihtiyaçları yapısal programlama ile çözülmüştür. Kodlar büyüdükçe sorunlar artmış ve *Nesne Yönelimli Programlama İhtiyacı* başlığı altında anlatılan problemler ortaya çıkmıştır. Bu yıllarda Simula ve Smalltalk yaklaşımı yazılımcıların imdadına yetişmiştir.

Bu dillerde nesneler (object) programın temel yapıtaşlarıdır. Nesneler; durum (state) ve davranışlara (behavior) sahiptir. Programlama, nesnelerin birbirlerine ileti göndermesiyle (message-passing) yapılır.

1979 senesinde bir Danimarkalı bilgisayar bilim adamı olan *Bjarne Stroustrup*, sonradan C++ olarak bilinecek olan "C with Classes" üzerinde çalışmaya başladı ve 1985 yılında ilk sürümünü yayınladı. C++ Programlama Dili;

- C Dilinden türetilmiştir.
- Düşük düzey programa yapılabilir
- İcra hızı yüksektir.
- C dilinden daha fazla kütüphaneye sahiptir.
- Nesne yönelimli programlamayı destekler.
- Diğer dillere göre daha etkin hafıza yönetimi sağlar
- Standart Template Library -STL ile jenerik (generic) programlamayı destekler.
- Birçok kavramı bir anda özümsemek gerektiğinden öğrenme eğrisi diktir.

Emreden paradigma (imprative programming) aksine nesnelerin birbirlerine ileti göndermesi bakış açısıyla yapılan programlamaya nesne yönelimli programlama (object oriented programming) paradigması adı verilir.

Sınıf ve Nesne Nasıl Tanımlanır?

Nasıl ki dünyamızda evler bir mimari plan (blueprint) üzerinden inşa ediliyor, nesneler (object) de bir plan üzerinden inşa edilir. Nesnelere ilişkin verilerin tutulduğu durumların (state) ve nesnelerin göstereceği davranışlarının (behavior) tanımlandığı bu plana sınıf (class) adı verilir. Durumlar (state), nesnelere ilişkin verilerin tutulduğu alanlardır (field).

Bir mimari plandan birçok ev yapılabileceği gibi, bir sınıftan birçok nesne imal edilebilir. İmal edilen her nesne (object) sınıfın bir örneğidir (instance). Sınıflar, aşağıdaki şekilde tanımlanır;

```
class sınıf-kimliği {
    veri-tipi alan-değişkeni-kimliği;
    veri-tipi davranış-kimliği();
} [örnek-değişkeni-1][, örnek-değişkeni-2];
```

Sınıflar, temelde yapı (struct) ve birlik (union) gibi tanımlanır ancak yapı gibi üyeleri sadece veriler olmaz, fonksiyonlar da sınıfların üyeleri olabilir. Sınıf üyesi olan fonksiyonlar, örnek değişkenlerin davranışlarını (behavior) gösterdiği yöntemlerdir (method).

Özet olarak sınıflar, kodumuzda imal edilecek nesnelere ilişkin ortak davranış ve özellikleri tasnif eden bir şablon bileşenidir. Aşağıda ortak davranış ve özelliklerin tanımlandığı bir `Ogrenci` sınıfı örneği verilmiştir.

```
#include <iostream>
using namespace std;

class Ogrenci {
public: /* bu sınıftan imal edilecek nesnelerin
    umuma açık (public) davranış ve özellikleri bu kısımda tanımlanır.
    Bir sonraki başlıkta anlatılmıştır.
    */
    unsigned yas; /* "yas"  kimlikli alan(field),
    ogrencinin yaşına ait durumlara(state) ait verileri tutar.
    */
    char cinsiyet; /* "cinsiyet"  kimlikli alan(field),
    ogrencinin cinsiyetine ait durumlara(state) ait verileri tutar.
    */
    void yasiniSoyle() /* yasiniSoyle() yöntemi(method),
    öğrencinin yaşını söyleme davranışını(behavior) tanımlıyor */
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle() /* cinsiyetSoyle() yöntemi(method),
    öğrencinin cinsiyetini söyleme davranışını(behavior) tanımlıyor */
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
} ilhan;  // Ogrenci sınıfından "ilhan" nesnesi imal edildi
int main() {
  ilhan.yas=50; // "ilhan" nesnesinin "yas" özelliği değiştirildi.
  ilhan.yasiniSoyle(); /* "ilhan" nesnesine yasiniSoyle()
  iletisi gönderildi (message passing). */
  ilhan.cinsiyet='E';
  ilhan.cinsiyetSoyle();  /* "ilhan" nesnesine cinsiyetSoyle()
  iletisi gönderildi (message passing).*/

  Ogrenci gullu; // Ogrenci sınıfından "gullu" nesnesi imal edildi
  gullu.yas=30; // "gullu" nesnesinin "yas" özelliği değiştirildi.
  gullu.cinsiyet='K';
  gullu.yasiniSoyle(); /* "gullu" nesnesine yasiniSoyle() iletisi
  gönderildi (message passing). */
  gullu.cinsiyetSoyle();  /* "gullu" nesnesine cinsiyetSoyle()
  iletisi gönderildi (message passing). */
}
/* Program çalıştırıldığında:
Yaşım:50
Cinsiyetim: ERKEK
Yaşım:30
Cinsiyetim: KADIN

...Program finished with exit code 0
*/
```

Program çalıştırıldığında evrensel olarak `ilhan` nesnesi oluşturulacaktır. Ana fonksiyon icra edilmeye başlandığında da `gullu` nesnesi gibi nesne imal edilebilir. Burada alanların yani imal edilen nesnelere ilişkin durumların davranışları etkilediği gözden kaçırılmamalıdır. Bu durumlara ilişkin veriler alanlarda (field) saklanmaktadır.

Sınıf Üyelerine Erişim ve Erişim Değiştiricileri

Sınıftan imal edilen nesnelerin üyelerine başka nesne ve programların nasıl erişeceği erişim değiştiricileri (access modifier) ile belirlenir. Görünürlük (visibility) sıfatları olarak da adlandırılan bu değiştirticiler kısaca sınıf üyelerinin sınıf içinden ve dışından erişimi belirleyen sıfatlardır. Bunlar;

- `public`: İmal edilecek nesnenin diğer nesneler tarafından ulaşılabilen yani umuma açık davranış ve özellikleri sınıf içinde bu belirleyici altında toplanır.
- `private`: İmal edilen nesnelerin mahrem/şahsi/kişisel davranış ve özellikleri sınıf içinde bu belirleyici altında toplanır. Ön tanımlı (default) erişim değiştiricidir. Bir sınıfta hiçbir erişim değiştirici yoksa tanımlanan durum ve davranışlar mahrem (private) olarak kabul edilir.
- `protected`: İmal edilen nesneler ile "bu sınıftan miras alan sınıflardan imal edilecek" nesnelerin ulaşabileceği korumalı (protected) davranış ve özellikleri bu belirleyici altında toplanır. Bu belirleyici ileride Kalıtım başlığında incelenecektir.

Aşağıda erişim belirleyicilerin uygulandığı basit bir `Kisi` sınıfı örnek verilmiştir. Bu sınıftan imal edilen nesnelerin `yas` özelliği diğer tüm nesneler tarafından erişilip değiştirilebilir, ancak `sir` özelliğine ise hiçbir nesne tarafından erişilemez. `maas` özelliğine ise `Kisi` sınıfından türemiş sınıflardan imal edilen nesneler tarafından bu özelliğe erişilebilir. İleride göreceğiz.

```
#include <iostream>
using namespace std;

class Kisi {
public: /* bu erişim değiştirici sonrası tanımlanacak üyeler;
    sınıftan imal edilecek nesnelerin umuma açık (public)
    davranış ve özellikleridir.
    */
    unsigned yas; /* "yas"  kimlikli alan(field),
    ogrencinin yaşına ait durumlara(state) ilişkin verileri tutar.
    imal edilecek nesnelerda umuma açık bir özelliktir.
    */
private:/* bu erişim değiştirici sonrası tanımlanacak üyeler;
    sınıftan imal edilecek nesnelerin mahrem (private)
    davranış ve özellikleridir.
    */

    int sir; /* "sir"  kimlikli alan(field),
    ogrencinin sırrına ait durumlara(state) ilişkin verileri tutar.
    imal edilecek nesnelerde mahrem bir özelliktir.
    diğer nesneler/programlar tarafından bu özelliğe erişilemez.
    */
protected: /* bu erişim değiştirici sonrası tanımlanacak üyeler;
    sınıftan imal edilecek nesneler ile bu sınıftan türetilmiş
    sınıflardan imal edilece nesnelerin erişebileceği korumalı (protected)
    davranış ve özellikleridir.
    */
    float maas;/* "maas"  kimlikli alan(field),
    ogrencinin maaşına ait durumlara(state) ilişkin verileri tutar.
    imal edilecek nesnelerde mahrem bir özelliktir.
    Ogrenci sınıfından türemiş sınıflar (ileride göreceğiz) dışında
    diğer nesneler/programlar tarafından bu özelliğe erişilemez.
    */
};
int main() {
    Kisi ilhan;  //Kisi sınıfından "ilhan" nesnesi imal edildi
    ilhan.yas=50; // "ilhan" nesnesinin "yas" özelliği değiştirildi.
    //ilhan.sir=-1; //HATA: Ana program ilhan nesnesinin sir özelliğine ulaşamaz.
    //ilhan.maas=10000.0; //HATA: Ana program ilhan nesnesinin maas özelliğine ulaşamaz.
}
```

Nesne Yapıcısı

Bir mimari plandan birçok ev yapılabileceği gibi, bir sınıftan birçok nesne imal edilebilir. İmal edilen her nesne (object) sınıfın bir örneğidir (instance). İmal edilecek nesnelerin varsayılan (default) olarak belirlenen durumlarla (state) imal edilmesi gerekiyorsa nesne yapıcısı (constructor) kullanılır.

Yapıcıların görevi bir sınıftan nesneleri, varsayılan durumlara sahip olarak imal etmektir. Nesnelere ait veriler, alanlarda (field) tutulurlar ve nesnenin durumumu (state) bu veriler belirler. Çünkü durumlar nesnenin davranışını etkiler.

Nesne yapıcıları sınıf kimliğiyle aynı olup bir değer geri döndürmeyen fonksiyonlarmış gibi tanımlanır. Amacı kısaca imal edilecek nesnelerin durumlarına ilk değer vermektir.

```
#include <iostream>
using namespace std;
class Ogrenci {
public: /* bu erişim değiştirici sonrası tanımlanacak üyeler;
    sınıftan imal edilecek nesnelerin umuma açık (public)
    davranış ve özellikleridir.
    */
    Ogrenci() { // Ogrenci yapıcısı
        // Yapıcı içerisinde imal edilecek nesnelerine ilk değer verilir.
        yas=6;
        cinsiyet='E';
        // Yapıcı içerisinde return talimatı bulunmaz!
    }
    unsigned yas; /* "yas"  kimlikli alan(field),
    ogrencinin yaşına ait durumlara(state) ait verileri tutar.
    */
    char cinsiyet; /* "cinsiyet"  kimlikli alan(field),
    ogrencinin cinsiyetine ait durumlara(state) ait verileri tutar.
    */
    void yasiniSoyle() /* yasiniSoyle() yöntemi(method),
    öğrencinin yaşını söyleme davranışını(behavior) tanımlıyor */
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle() /* cinsiyetSoyle() yöntemi(method),
    öğrencinin cinsiyetini söyleme davranışını(behavior) tanımlıyor */
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
};
int main() {
    Ogrenci ilhan; /* Ogrenci sınıfından "ilhan" nesnesi; Ogrenci() yapıcısı
    kullanılarak yası 6 ve cinsiyeti E olarak imal edildi. */

    ilhan.yasiniSoyle(); /* "ilhan" nesnesine yasiniSoyle()
    iletisi gönderildi (message passing). */
    ilhan.cinsiyetSoyle();  /* "ilhan" nesnesine cinsiyetSoyle()
    iletisi gönderildi (message passing).*/

    ilhan.yas=50; // "ilhan" nesnesinin "yas" özelliği değiştirildi.
    ilhan.yasiniSoyle(); /* "ilhan" nesnesine yasiniSoyle() iletisi
    gönderildi (message passing). */
}
```

Örnekte `Ogrenci` sınıfından imal edilecek her nesnenin yaşı 6, ve cinsiyeti 'E' olarak üretilecektir. Hiçbir parametresi olamayan bu yapıcıya ön tanımlı yapıcı (default constructor) adı verilir.

Dinamik Nesne İmalatı

Nesneleri çalıştırma anında (run time) da imal edebiliriz. Nesneleri çalışma zamanında yapıcılar kullanarak imal ederiz ve buna dinamik başlatma (dynamic initialization) adı verilir. Yapıcısı çağrılırken veri tutan alanlara (field) ilk değer değerler için mantık içeriyorsa,

```
SınıfKimliği* imal-edilen-nesne-göstericisi = new SınıfKimliği(yapıcı-argümanları);
```

Dinamik olarak `new` işleci (new operator) ile imal edilecek nesneye öbek bellekte (heap segment) yer tahsis edilir. Ardından nesnenin yapıcısı ile veri tutan alanlarına (field) ilk değer değerler için mantık çalıştırılır ve nesne göstericisi geri döner.

Nesne göstericileri üzerinden sınıf üyelerine dolaylı işleç (indirection operator) yani (`->`) ile erişilir. Bu nokta işleci ile aynı önceliklidir.

Dinamik olarak imal edilmiş nesne kullanıldıktan sonra `delete` işleci (delete operator) ile bellekten kaldırılabilir. Çeşitli yapıcılara sahip dinamik olarak nesne imal edilen bir kod örneği aşağıda verilmiştir;

```
#include <iostream>
using namespace std;
class Karmasik {
public:
    float gercek, sanal;
    Karmasik() // varsayılan yapıcı (default constructor)
    {
        gercek = 0.0;
        sanal = 0.0;
        cout<< "Varsayılan (default) yapıcı çağrıldı" << endl;
    }
    Karmasik(float pGercek, float pSanal): gercek(pGercek),sanal(pSanal) {
        // parametreli yapıcı
        cout<< "Parametreli yapıcı çağrıldı" << endl;
    }
    Karmasik(float pGercek): Karmasik(pGercek,0.0) {
        //delegating constructor
        cout<< "Öncesinde parametreli yapıcı çağrılacak -> "
            << "Tek parametreli yapıcı çağrıldı" << endl;
    }
    void yaz() { //yaz yöntemi
        cout<< gercek << "+" << sanal << +"i" << endl;
    }
};
int main(void) {
    Karmasik* pc1=new Karmasik(); /* hafıza tahsisi: allocate memory
     Bu durumda faıza ayrıldıktan sonra alanlara ilk değerler varsayılan yapıcı
     (default constructor) ile veriliyor
     */
    pc1->yaz();
    Karmasik* pc2=new Karmasik(2.0,3.0); /* hafıza tahsisi: allocate memory
     Bu durumda hafıza ayrıldıktan sonra alanlara ilk değerler parametreli
     yapıcı (default constructor) ile veriliyor */
    pc2->yaz();
    Karmasik* pc3=new Karmasik(4.0); /* hafıza tahsisi: allocate memory
     Bu durumda faıza ayrıldıktan sonra alanlara ilk değerler devredilen yapıcı
     (delegating constructor) ile veriliyor
    */
    pc3->yaz();
    delete pc3; // hafızayı serbest bırakma:deallocate memory
    delete pc2; // hafızayı serbest bırakma:deallocate memory
```

```
    delete pc1; // hafızayı serbest bırakma:deallocate memory
}
/* Program Çıkısı:
Varsayılan (default) yapıcı çağrıldı
0+0i
Parametreli yapıcı çağrıldı
2+3i
Parametreli yapıcı çağrıldı
Öncesinde parametreli yapıcı çağrılacak -> Tek parametreli yapıcı çağrıldı
4+0i

...Program finished with exit code 0
*/
```

Soyut, Somut ve Bilgi Gizleme

Soyut (abstract) kavramı kelime itibariyle detay içermeyen, özet anlamına gelir. Somut (concrete) ise en ince detayına kadar bilinen, elle tutulur anlamına gelir. Bir şey hakkında soyutlama yapıldığında, o şeyin detayıyla ilgili olan tüm bilgiler gizlenir (information hiding).

Bir sınıftan imal edilen nesneler tüm durum (state) veya davranışlarını (behavior) diğer nesnelere şeffaf olarak gösterip diğer nesnelerin değiştirmesine izin verilmemesi gerekir. Bu durumda nesnenin durum ve davranışları tutarsız hale gelir. Bu nedenle bir sınıftan imal edilen nesnelerin durum ve davranışlarının diğer nesnelere veya dış dünyaya kontrollü bir şekilde açılması gerekir.

Soyutlama (abstraction), kullanıcıya gerekli bilgileri gösterme ve kullanıcıya göstermek istemediği veya belirli bir kullanıcıyla ilgisi olmayan ayrıntıları gizleme işlemidir. Bir sınıfta soyutlama iki türlü yapılır;

- Veri soyutlaması (data abstraction): nesnenin durumları üzerinden yapılan soyutlama.
- Kontrol soyutlaması (kontrol abstraction): nesnenin davranışları üzerinde yapılan soyutlama.

Sarmalama

Yukarıda verilen `Ogrenci` örneğinden devam edecek olursak imal edilecek nesnenin `yas` durumuna bir başka nesne veya program `-1` gibi bir değer atayabilir. Bu durumda `yasiniSoyle()` davranışında istenmeyen bir çıktı verilmesine sebep olur. Bunu gidermek için veri soyutlaması yapılması gerekir. Bu durumda yas alanına bir değer koymayı ve yas alanından bir değer almayı kontrol altına almalıyız. Benzer durum `cinsiyet` alanı için de geçerlidir.

Veri soyutlamasını yapmak için `yas` ve `cinsiyet` alanlarını mahrem (private) olarak belirleyip bu alanlara değer atama ve değer okuma işlevlerini yerine getirecek `get` ve `set` yöntemlerini umuma açık olarak tanımlamalıyız;

```
#include <iostream>
using namespace std;

class Ogrenci {
private: /* bu erişim değiştirici sonrası tanımlanacak üyeler;
    sınıftan imal edilecek nesnelerin mahrem (private)
    davranış ve özellikleridir.
    */
    unsigned yas; /* "yas"  kimlikli alan(field),
    ogrencinin yaşına ait durumlara(state) ait verileri tutar.
    */
    char cinsiyet; /* "cinsiyet"  kimlikli alan(field),
     ogrencinin cinsiyetine ait durumlara(state) ait verileri tutar.
    */
public: /* bu erişim değiştirici sonrası tanımlanacak üyeler;
     sınıftan imal edilecek nesnelerin umuma açık (public)
```

```
     davranış ve özellikleridir.
    */
    Ogrenci() { // Ogrenci yapıcısı
        // Yapıcı içarisinde imal edilecek nesnelerine ilk değer verilir.
        yas=6;
        cinsiyet='E';
        // Yapıcı içerisinde return talimatı bulunmaz!
    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <3)
            cout << "Öğrenci 3 Yaşından Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle() /* yasiniSoyle() yöntemi(method),
    öğrencinin yaşını söyleme davranışını(behavior) tanımlıyor */
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle() /* cinsiyetSoyle() yöntemi(method),
    öğrencinin cinsiyetini söyleme davranışını(behavior) tanımlıyor */
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
};
int main() {
    Ogrenci ilhan; /* Ogrenci sınıfından "ilhan" nesnesi;
    Ogrenci() yapıcısı kullanılarak yası 6 ve cinsiyeti E olarak imal edildi. */
    ilhan.yasiniSoyle(); /* "ilhan" nesnesine yasiniSoyle()
    iletisi gönderildi (message passing). */
    ilhan.cinsiyetSoyle();  /* "ilhan" nesnesine cinsiyetSoyle()
    iletisi gönderildi (message passing).*/

    ilhan.setYas(0); // "ilhan" nesnesinin "yas" özelliği değiştirildi.
    ilhan.yasiniSoyle(); /* "ilhan" nesnesine yasiniSoyle() iletisi
    gönderildi (message passing). */
    ilhan.setCinsiyet('H');
    ilhan.cinsiyetSoyle();
}
```

İşte bir durumun (state) veri soyutlaması yapılmış haline özellik (property) adı veririz. Çünkü artık duruma dışarıdan doğrudan müdahale edilemez. Nesnenin durum üzerinde kontrol hakları vardır.

Benzer şekilde davranışlar (behavior) da nesne dışına kontrollü olarak açılabilir. Bu durumda da kontrol soyutlaması (control abstraction) yapılır. Aşağıda faktöriyel hesaplayan bir hesap makinesinin ekranı yenileme mahrem davranışı buna örnek verilebilir;

```
#include <iostream>
using namespace std;
class HesapMakinesi {
private:
    int hesaplanan;
    void ekraniYenile() {
        cout << "Hesaplanan:" << hesaplanan << endl;
    }
public:
    HesapMakinesi() { //Yapıcı
        hesaplanan=0;
        ekraniYenile();
    }
    void setHesapMakinesi(int pSayi) {
        hesaplanan=pSayi;
        ekraniYenile();
    }
    int getHesapMakinesi() {
        return hesaplanan;
    }
    void makineyiSifirla() {
        hesaplanan=0;
        ekraniYenile();
    }
    void FaktoriyelHesapla() {
        unsigned temp=1;
        for (int sayac = hesaplanan; sayac > 0; sayac--)
            temp =  temp*sayac;
        hesaplanan=temp;
        ekraniYenile();
    }
};
int main ()
{
  HesapMakinesi hesaplayici; // hesaplayici nesnesi imal edildi
  hesaplayici.setHesapMakinesi(4);
  hesaplayici.FaktoriyelHesapla();
  hesaplayici.setHesapMakinesi(6);
  hesaplayici.FaktoriyelHesapla();
  hesaplayici.makineyiSifirla();
}
```

Bir sınıf üzerinde veri soyutlaması (data abstraction) veriler üzerine kılıf çekilir, hem de kontrol soyutlaması (control abstraction) ile davranışların üzerine bir kılıf çekilir. Bir sınıf üzerinde her iki işlemin birlikte yapılasına sarmalama (encapsulation) adı verilir. Soyutlamaların gerekli olup olmadığına programcı karar verir.

Sarmalama işlemi örneklerde görüldüğü üzere erişim değiştiricilerle (access modifier) yapılır. Böylece imal edilen nesnelerin verileri ve davranışları güvenli bir şekilde dış dünyaya açılmış olur. Böylece, sınıf ve nesne gibi bileşenlerin içeriğini bilmeden soyut (abstract) olarak kullanmak mümkün olmaktadır. Programcı tasarladığı sınıfın soyut olarak katlanılacağını düşünerek sınıfları tasarlamalıdır.

Özetle sarmalama (encapsulation) tekniğini uygulamak, kullanımı kolay sınıf (class), bileşen (component) , kütüphane (library) ve çerçeve yapılar (framework) elde etmemizi sağlar;

- n adet veri (data) ve n adet davranış (behavior) sarmalama işlemine tabi tutulursa sınıf,
- n adet sınıf sarmalama işlemine tabi tutulursa bileşen,
- n adet bileşen sarmalama işlemine tabi tutulursa kütüphane,
- n adet kütüphane sarmalama işlemine tabi tutulursa çerçeve yapılar oluşur.

Böylece bize projemizde proje süresini ve maliyetini kısaltma üstünlüğünü verir.

Kalıtım

Öğrenci, öğretmen ve müdürün olduğu bir sistemi örnek alarak ele alalım. Bunlardan her birinden nesne imal etmek için her birine ayrı ayrı sınıf tanımlamak gerekir.

```
class Ogrenci {
private:
    unsigned yas;
    char cinsiyet;
    unsigned ogrenciNo;
public:
    Ogrenci() {
        yas=18;
        cinsiyet='E';
    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <18)
            cout << "Yaş 18 den Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle()
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle()
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
    void dersCalis() {
        cout << "Ders Çalışıyorum..." << endl;
    }
```

```
};
```

Öğretmen için ayrı bir sınıf;

```
class Ogretmen {
private:
    unsigned yas;
    char cinsiyet;
    unsigned verdigiDersKodu;
public:
    Ogretmen() {
        yas=18;
        cinsiyet='E';
    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <18)
            cout << "Yaş 18 den Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle()
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle()
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
    void dersVer() {
        cout << "Ders Veriyorum..." << endl;
    }
};
```

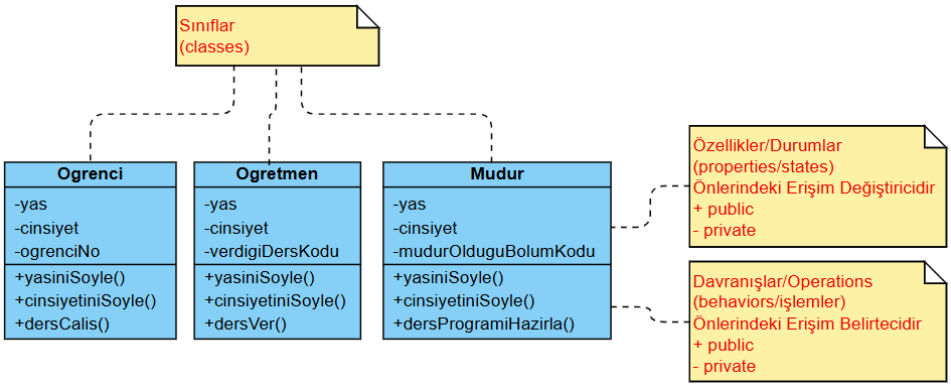
Müdür için ayrı bir sınıf;

```
class Mudur {
private:
    unsigned yas;
    char cinsiyet;
    unsigned muduruOlduguBolumKodu;
```

```
public:
    Mudur() {
        yas=18;
        cinsiyet='E';
    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <18)
            cout << "Yaş 18 den Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle()
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle()
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
    void dersProgramiHazırla() {
        cout << "Ders Programı Hazırlıyorum..." << endl;
    }
};
```

Yazılım analizi yapılırken takım içerisinde iletişimi kolaylaştırmak için aşağıdaki sınıf diyagramları (class diagram) kullanılır. Bunlar Unified Modelling Language-UML dilinin bir parçasıdır.

Bu sınıfların her birinde `yas` ve `cinsiyet` gibi ortak özellikler (property) veya `yasiniSoyle()` ve `cinsiyetSoyle()` gibi ortak davranışlar (behavior) bulunur. Her sınıf için bunlar tekrar tekrar tanımlanırlar. Hâlbuki bu gibi ortak özellikler ve davranışlar tek bir sınıf altında toplanabilir. Bütün bu ortak özellik ve davranışlar tek bir sınıfta toplandığı zaman daha yönetilebilir bir kod elde edilir.



Şekil 22. Öğrenci, Öğretmen ve Müdür UML Sınıf Diyagramları

Ortak özellik ve davranışların bir sınıfta toplanarak bu sınıftan miras alınmasına kalıtım (inheritance) adı verilir. Yukarıdaki örnekte ortak özellik ve davranışlar `Kisi` sınıfına toplanarak aşağıdaki şekilde kod yeniden düzenlenebilir;

Aşağıda verilen ve Kişi olarak tanımlanan genel sınıf (general class), taban sınıf (base class) veya ebeveyn sınıf (parent class) olarak adlandırılır. Daha sonra tanımlanacak Öğrenci, Öğretmen ve Müdür sınıflarının ortak özellik ve davranışlarını barındırır.

```
class Kisi {
private:
    unsigned yas;
    char cinsiyet;
public:
    Kisi() {
        yas=18;
        cinsiyet='E';
    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <18)
            cout << "Yaş 18 den Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle()
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle()
    {
        if (cinsiyet=='E' || cinsiyet=='e')
```

```
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
};
```

Aşağıda taban sınıftan türemiş (derived) Öğrenci, Öğretmen ve Müdür sınıfları verilmiştir. Bu sınıflar genel sınıf olan Kişi sınıfının yanında kendine özel özellik ve davranışlara da sahiptir. Özel sınıf (private class), türemiş sınıf (derived class) veya çocuk sınıf (child class) olarak adlandırılan bu sınıflar, mirasçı olup, kendilerinden imal edilmiş nesneler; hem taban sınıfın özellik ve davranışlarını, hem de kendi özellik ve davranışlarını sergilerler.

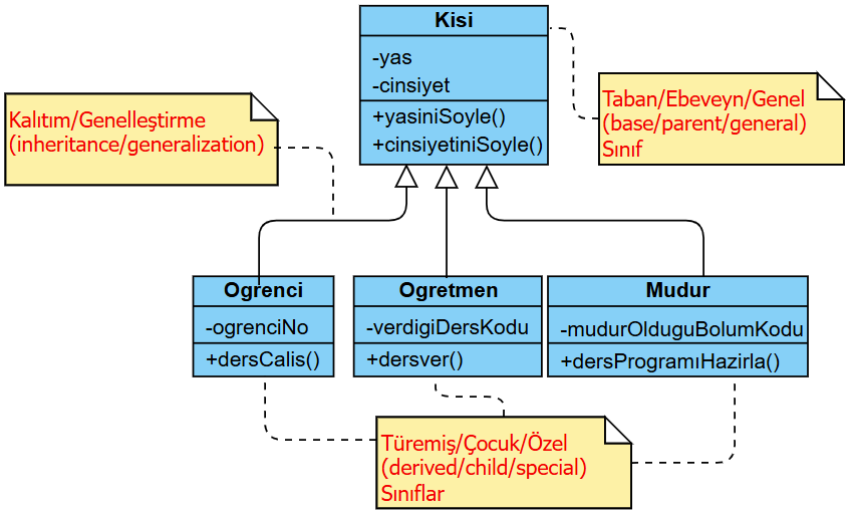
```
class Ogrenci: public Kisi {
private:
    unsigned ogrenciNo;
public:
    Ogrenci() {
    }
    void dersCalis() {
        cout << "Ders Çalışıyorum..." << endl;
    }
};
class Ogretmen:public Kisi {
private:
    unsigned verdigiDersKodu;
public:
    Ogretmen() {
    }
    void dersVer() {
        cout << "Ders Veriyorum..." << endl;
    }
};
class Mudur:public Kisi {
private:
    unsigned muduruOlduguBolumKodu;
public:
    Mudur() {
    }
    void dersProgramiHazırla() {
        cout << "Ders Programı Hazırlıyorum..." << endl;
    }
};
```

Programlamanın yapıldığı ana fonksiyonda bu üç sınıftan da nesne imal edilmiş ve çeşitli davranışlar göstermesi için kendilerine ileti gönderilmiştir (message-passing).

```
int main() {
    Ogrenci ilhan;
    Ogretmen mehmet;
    Mudur abdullah;

    ilhan.yasiniSoyle();
    ilhan.cinsiyetSoyle();
    ilhan.dersCalis();

    mehmet.yasiniSoyle();
    mehmet.cinsiyetSoyle();
    mehmet.dersVer();

    abdullah.yasiniSoyle();
```

```
    abdullah.cinsiyetSoyle();
    abdullah.dersProgramiHazırla();
}
```

Yeni haliyle sınıf diyagramları aşağıda verilmiştir;



Şekil 23. Kişi Taban Sınıfı Eklenmiş Kalıtım UML Sınıf Diyagramı

Yukarıdaki örnekler incelendiğinde türemiş sınıfların taban sınıftan miras alırken üç farklı şekilde alabilirler;

- Umuma açık (public) kalıtım: Taban sınıfın umuma açık üyelerini türetilmiş sınıfta umuma açık hale getirir ve taban sınıfın korumalı (protected) üyeleri de türetilmiş sınıfta korumalı olarak kalır.
- Mahrem (private) kalıtım: Taban sınıfın umuma açık ve korumalı üyelerini türetilmiş sınıfta mahrem hale getirir.
- Korumalı (protected) kalıtım: Taban sınıfın umuma açık ve korumalı üyelerini türetilmiş sınıfta korumalı hale getirir.

Bunu aşağıdaki kod örneğiyle anlayabiliriz;

```
class Base { //taban sınıf
public:
    /* burada tanımlanan üyeler umuma açıktır.
       Yani bu üyelere sınıf dışı ve içinden erişilebilir.
    */
    int x;
protected:
    /* burada tanımlı üyelere türeyen sınıftan türemiş sınıflardan erişilir.
       Bir başka deyişle miras alan sınıflar tarafından burada tanımlanan
       üyelere erişilir. */
    int y;
private:
    /* burada tanımlı üyelere yalnızca bu sınıftan imal edilen nesneler
       erişir. Bir başka deyişle yalnızca bu sınıf erişebilir.*/
    int z;
};
class PublicDerived: public Base { //public inheritance
  // Bu sınıfta x alanı(field), public olmuştur.
  // Bu sınıfta y alanı(field), protected olmuştur.
  // Bu sınıfta z alanı(field) ERİŞİLEMEZ olmuştur. -> Public-Derived
};
class PrivateDerived: private Base { //private inheritance
  // Bu sınıfta x alanı(field), private olmuştur.
  // Bu sınıfta y alanı(field), private olmuştur.
  // Bu sınıfta z alanı(field) ERİŞİLEMEZ olmuştur. -> Private-Derived
};
```

```
class ProtectedDerived: protected Base { //protected inheritance
  // Bu sınıfta x alanı(field), protected olmuştur.
  // Bu sınıfta y alanı(field), protected olmuştur.
  // Bu sınıfta z alanı(field) ERİŞİLEMEZ olmuştur. -> Protected-Derived
};
```

Değişim yönetimini kolayca yapma üstünlüğünü veren kalıtımın (inheritance) iki önemli özelliği vardır; Paylaşılan bir ortak kod (shared code) vardır ve paylaşılan kodda yapılan değişiklik anında alt sınıflara yansır (snap change).

Örneğimizden gidecek olursak, Kişi sınıfında yapılacak bir özellik veya davranış eklemesi anında bu sınıftan türemiş sınıflarda yapılmış olur.

Korumalı (protected) olarak taban sınıftan alınan durum ve davranışlara ait bir örnek aşağıda verilmiştir;

```
#include <iostream>
using namespace std;
class Kisi {
protected: //Burada tanımlana alan ve davranışlara türemiş sınıflardan da erişilebilir.
    unsigned yas;
    char cinsiyet;
public:
    Kisi() {
        yas=18;
        cinsiyet='E';
    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <18)
            cout << "Yaş 18 den Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle()
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle()
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
```

```
    }
};
class Ogrenci: public Kisi {
private:
    unsigned ogrenciNo;
public:
    Ogrenci() {
        yas=35;  // Artık yas alanına bu türemiş sınıftan da erişilebiliyor.
        cinsiyet='K'; // Artık cinsiyet alanına bu türemiş sınıftan da erişilebiliyor.
    }
    void dersCalis() {
        cout << "Ders Çalışıyorum..." << endl;
    }
};
int main() {
    Ogrenci ayse;

    ayse.yasiniSoyle();
    ayse.cinsiyetSoyle();
    ayse.dersCalis();
}
```

Kalıtımda üyelere erişim, özet olarak aşağıdaki tabloda verilmiştir;

Üyelere Erişim	public	protected	private
Aynı Sınıf İçinde	var	var	var
Türemiş Sınıfta	var	var	yok
Sınıf Dışından	var	yok	yok

Tablo 19. Kalıtımda Sınıf Üyelerine erişim.

Türetilmiş bir sınıf, aşağıdaki istisnalar dışında tüm taban sınıf yöntemlerini miras alır;

1. Taban sınıfın yapıcıları, yıkıcıları (destructor) ve kopya yapıcıları.
2. Taban sınıfın aşırı yüklenmiş işleçleri.
3. Taban sınıfın arkadaş (friend) fonksiyonları.

Nesne Yönelimli Programlamada sınıflar, birden fazla taban sınıftan miras alamayacak şekilde tasarlanır. Mantık olarak bir ev, birden fazla mimari plandan hazırlanmayacağından, çoklu kalıtım (multiple inheritance) sınıf tasarımında kullanılmaz.

C++ dilinde bir türemiş sınıfın birden fazla ebeveyn sınıftan miras alması engellenmemiştir. İleride göreceğimiz ara yüzler (interface) de C++ dilinde sınıf olarak kodlandığından sanki çoklu kalıtım varmış gibi algılanmaktadır.

Saf nesne yönelimli diller olan Java ve .Net dillerinde ara yüzler ayrı bir anahtar kelimeyle tanımlandığından doğal olarak birden çok sınıftan miras alma engellenmiştir.

Aşırı Yükleme

C dilinden farklı olarak C++ dilinde aşırı yükleme (overloading); aynı davranışın farklı yöntemlerle (method) yeniden tanımlanmasıdır.

Fonksiyonlarda (function) aşırı yükleme, farklı parametrelerle fonksiyonun yeniden tanımlanmasıdır. Aşırı yüklemede fonksiyonun kimliği ve geri dönüş tipi değişmez. Farklı argümanlar ile fonksiyon gövdesi ile yeniden tanımlanır.

Aşağıda bir fonksiyonun aşırı yüklemesine bir örnek verilmiştir.

```
#include <iostream>
using namespace std;
void topla(int a, int b) {  //İlk tanımlanan topla fonksiyonu
  cout << "sum = " << (a + b);
}
```

```
void topla(double a, double b) {  // Aşırı yüklenmiş topla fonksiyonu
    cout << endl << "sum = " << (a + b);
}
int main() {
    topla(1, 2);
    topla(2.5, 5.5);
}
```

Fonksiyonlarda varsayılan argümanlar kullanıldığında aşırı yükleme tanımlanmasına çok dikkat edilmelidir;

```
void topla(int a = 10, int b = 20); // geçerli
void topla(int a = 22, int b = 2); // HATA: fonksiyon imzası (signature) aynı!
/* Yani parametreler veri tipleriyle aynı sırada tanımlı. Sadece varsayılan argümanlar
değişik.
*/
void f(int a); // HATA: varsayılan argümanlı fonksiyon bunu karşılıyor.
void f(int a, b); // HATA: varsayılan argümanlı fonksiyon bunu karşılıyor.
```

Sınıflardaki yöntemlere (method) de fonksiyonlar gibi aşırı yükleme yapılır. Yani farklı parametrelerle yöntem yeniden tanımlanır. Aşırı yüklemede yöntemin kimliği ve geri dönüş tipi değişmez. Farklı argümanlar ile yöntemin gövdesi ile yeniden tanımlanır

```
#include <iostream>
using namespace std;
class Karmasik {
public:
    float gercek, sanal;
    Karmasik() { // varsayılan yapıcı (default constructor)
        gercek = 0.0;
        sanal = 0.0;
    }
    void yaz() { //yaz yöntemi
        cout<< gercek << "+" << sanal << +"i" << endl;
    }
    void yaz(char pKarakter) { // aşırı yüklenmiş yaz yöntemi
        //aşırı yüklenmiş yaz yöntemi
        cout << pKarakter << gercek << "+" << sanal << +"i" <<pKarakter << endl;
    }
    void yaz(char pIlkKarakter, char pSonKarakter) { //aşırı yüklenmiş bir başka yaz yöntemi
        cout << pIlkKarakter << gercek << "+" << sanal
             << "i" << pSonKarakter << endl;
    }
};
int main(void) {
    Karmasik c1;
    c1.gercek=1;
    c1.sanal=2;
    c1.yaz();             // 1+2i
    c1.yaz('@');          // @1+2i@
    c1.yaz('[', ']');     // [1+2i]
}
```

Hali hazırda tanımlı olan işleçlerle (operator) bizim tanımladığımız sınıflardan imal edilen nesneler üzerinde işlem yapmak için aşırı yükleme yapılabilir. Ya da istenmeyen işleçler =delete belirleyicisi (=delete specifier) ile sınıfla işlem yapması kaldırılabilir. Bu işleç aynı zamanda kalıtımdan davranılan davranışları ve yapıcıları kaldırmak için de kullanılabilir. Aşağıda toplama işlecine aşırı yükleme örneği verilmiştir;

```
#include <iostream>
using namespace std;
class Karmasik {
```

```
public:
    float gercek, sanal;
    Karmasik() { // varsayılan yapıcı (default constructor)
        gercek = 0.0;
        sanal = 0.0;
    }
    Karmasik(float pGercek, float pSanal): gercek(pGercek),sanal(pSanal) {
    }
    Karmasik(float pGercek): gercek(pGercek) {
        sanal = 0.0;
    }
    void yaz() { //yaz yöntemi
        cout<< gercek << "+" << sanal << +"i" << endl;
    }
    Karmasik operator+(const Karmasik& pKarmasik) {
        /* + işleci aşırıyükleniyor (overloading): iki karmaşık sayı toplanıyor */
        Karmasik toplam;
        toplam.gercek = gercek + pKarmasik.gercek;
        toplam.sanal = sanal + pKarmasik.sanal;
        return toplam;
    }
    Karmasik operator+(const float pGercek) {
        /* + işleci aşırı yükleniyor (overloading): karmaşık sayı ile
           gerçek sayı toplanıyor */
        Karmasik toplam;
        toplam.gercek = gercek + pGercek;
        toplam.sanal = sanal;
        return toplam;
    }
    Karmasik(const Karmasik& pKarmasik) { /*bir başka aşırı yüklenmiş,
                                            kopya yapıcı (copy constructor):
                                            atama (=) için tanımlanmalıdır. */
        gercek= pKarmasik.gercek;
        sanal=pKarmasik.sanal;
    }
    void operator-(const Karmasik &) = delete;/* Örnek olarak Çıkarma işlemi
                                                 engellenmiş. */
};
int main(void) {
    Karmasik c1;
    c1.yaz();
    Karmasik c2(1.0,1.0);
    c2.yaz();
    Karmasik c3(1.0);
    c3.yaz();
    Karmasik c4=c2+c3;
    c4.yaz();
    c1=c2+c3;
    c1.yaz();
}
```

Yapıcıların Aşırı Yüklenmesi

Yapıcıların görevi kısaca imal edilecek nesnelerin durumlarına (state) ilk değer vermektir. Durumlara ilişkin veriler de alanlarda (field) tutulurlar. Nesneleri de dışarıdan verilen parametrelerle farklı durumlarda imal etmek için aşırı yüklenmiş yapıcıları kullanabiliriz. Yani farklı parametrelerle yapıcıları yeniden tanımlayabiliriz.

Yapıcıların kimliği sınıf kimliğiyle aynıdır. Aşırı yüklemede tanımlanacak yapıcının kimliği de sınıf kimliği ile aynı olur. Yapıcılar, yöntemlerin aksine geri değer döndürmezler. Aşağıda aşırı yüklenmiş yapıcı, parametreli (parameterized constructor),

```
#include <iostream>
using namespace std;
class Karmasik {
public:
    float gercek, sanal;
    Karmasik() { /* varsayılan (default constructor):
        Gerçek ve sanal kısım 0 olarak nesneler imal edilir. */
        gercek = 0.0;
        sanal = 0.0;
    }
    Karmasik(float pGercek, float pSanal) { /* aşırı yüklenmiş yapıcı:
        Gerçek ve sanal kısım parametrelerden gelen değerler olacak şekilde
        olarak nesneler imal edilir. */
        gercek = pGercek;
        sanal = pSanal;
    }
    Karmasik(float pGercek) {/* aşırı yüklenmiş bir başka yapıcı:
        Gerçek parametrelerden gelen değer olacak şekilde sanal kısım 0
        olarak nesneler imal edilir. */
        sanal=0.0;
    }
    Karmasik(const Karmasik& pKarmasik) { /*bir başka aşırı yüklenmiş,
        kopya yapıcı (copy constructor):
        Parametre olarak bir başka karmaşık sayı veriliyor ve gerçek ve sanal
        Durumları, parametreden gelen karmaşık sayı ile aynı olacak şekilde
        nesneler imal edilir. */
        gercek= pKarmasik.gercek;
        sanal=pKarmasik.sanal;
    }
    void yaz() { //yaz yöntemi
        cout<< gercek << "+" << sanal << +"i" << endl;
    }
};
int main(void) {
    Karmasik c1; // varsayılan yapıcı ile imal edilen c1 nesnesi
    c1.yaz(); // 0+0i
    Karmasik c2(2.0,1.0); // aşırı yüklenmiş yapıcı ile imal edilen c2 nesnesi
    c2.yaz(); // 2+1i
    Karmasik c3=c2; // kopya yapıcı ile imal edilen c3 nesnesi
    c3.yaz(); // 2+1i
    Karmasik c4(12.0);
    c4.yaz(); //12+0i
}
```

Yukarıdaki örnekte;

- Aynı sınıftan bir nesneyi bir başka imal edilmiş nesnenin durumlarına sahip olarak imal edebiliriz. Bir nesneyi bir yapıcıya argüman olarak geçirmek ve yeni nesneye argümanın durumlarını vermek gerekir. `Karmasik(const Karmasik& pKarmasik)` şeklinde kodlanmış bu yapıcıya kopya yapıcı (copy constructor) adı verilir.
- Yukarıdaki örnekte `Karmasik(float pGercek, float pSanal)` ve `Karmasik(float pGercek)` yapıcıları parametreli yapıcı (parameterized constructor) olarak aşağıdaki şekilde tanımlanabilir.

```
    Karmasik(float pGercek, float pSanal): gercek(pGercek), sanal(pSanal) {
      /* parametreli yapıcı, her bir alana parametrede atama yapılır.*/
    }
    Karmasik(float pGercek): gercek(pGercek), sanal(0.0) {
```

```
    }
```

– Üstte örneği verilen ikinci yapıcı, yapıcı devretme (delegating constructor) ile aşağıdaki şekilde de tanımlanabilir;

```
    Karmasik(float pGercek): Karmasik (pGercek, 0.0) {
       /* Bir yapıcı görevini bir başka yapıcıya devrederek
          Alanların (field) ilk değerleri belirleniyor. */
    }
```

Yukarıda belirtilen ön tanımlı yapıcı (default constructor), =default belirleyicisi (=default specifier) ile değiştirilebilir. Bu belirleyici aşırı yüklenmiş fonksiyonlarda da kullanılabilir[16]. Aşağıda bu belirleyiciye ilişkin örnek verilmiştir;

```
#include <iostream>
using namespace std;
class A {
public:
    A(int x) // Kullanıcı Tanımlı parametreli yapıcı
    {
        cout << "Parametreli Yapıcı" << endl;
    }
    A() = default; // derleyiciye A() yapıcısının ön tanımlı olduğu bildiriliyor
};
int main()
{
    A a; // A() yapıcısı ile nesne imal edilir.
    A x(1); // A(int x) yapıcısı ile nesne imal edilir.
    return 0;
}
```

Kalıtım (inheritance) durumunda ise taban sınıfın yapıcısı çağrılır. Bu duruma aşağıdaki örnek verilebilir;

```
#include <iostream>
using namespace std;

class Kisi {
private:
    unsigned yas;
    char cinsiyet;
public:
    Kisi() {
        yas=18;
        cinsiyet='E';
    }
    Kisi(unsigned pYas, char pCinsiyet): yas(pYas), cinsiyet(pCinsiyet) {

    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <18)
            cout << "Yaş 18 den Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
```

[16] https://www.geeksforgeeks.org/explicitly-defaulted-deleted-functions-c-11/

```
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle()
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle()
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
};
class Ogrenci: public Kisi {
private:
    unsigned ogrenciNo;
public:
    Ogrenci() {
        //yas 18 ve cinsiyet 'E' olarak taban sınıfta tanımlanır.
    }
    Ogrenci(unsigned pYas, char pCinsiyet): Kisi(pYas,pCinsiyet) /* Taban
      sınıftaki yas ve cinsiyet private olduğundan ulaşılamaz.
      Ama taban sınıfın yapıcısına parametreler gönderilerek ilk değer
      verilebilir. */
    {
        ogrenciNo=1;
    }
    void dersCalis() {
        cout << "Ders Çalışıyorum..." << endl;
    }
};

int main() {
    Ogrenci ilhan;
    ilhan.yasiniSoyle(); // Yaşım:18
    ilhan.cinsiyetSoyle(); // Cinsiyetim ERKEK
    ilhan.dersCalis();

    Ogrenci ayse(25,'K');
    ayse.yasiniSoyle(); // Yaşım:25
    ayse.cinsiyetSoyle(); // Cinsiyetim KADIN
    ayse.dersCalis();
}
```

Ön tanımlı argümanlar (default argument) da yapıcılarda kullanılabilir. Karmaşık sayılar örneği aşağıda verilmiştir;

```
#include <iostream>
```

```
using namespace std;
class Karmasik {
    float gercek, sanal;  //bu üyeler varsayılan olarak private
public:
    Karmasik(float pGercek=0.0, float pSanal=0.0): gercek(pGercek),sanal(pSanal) {
    }
    void yaz() { //yaz yöntemi
        cout<< gercek << "+" << sanal << +"i" << endl;
    }
    Karmasik operator+(const Karmasik& pKarmasik) {
        Karmasik toplam;
        toplam.gercek = gercek + pKarmasik.gercek;
        toplam.sanal = sanal + pKarmasik.sanal;
        return toplam;
    }
    Karmasik operator+(const float pGercek) {
        Karmasik toplam;
        toplam.gercek = gercek + pGercek;
        toplam.sanal = sanal;
        return toplam;
    }
    Karmasik(const Karmasik& pKarmasik) { // copy constructor
        gercek= pKarmasik.gercek;
        sanal=pKarmasik.sanal;
    }
};
int main(void) {
    Karmasik c1; // Karmasik(0.0,0.0) yapıcısı ile imal edildi.
    c1.yaz();
    Karmasik c2(1.0,1.0); // Karmasik(1.0,1.0) yapıcısı ile imal edildi.
    c2.yaz();
    Karmasik c3(1.0); // Karmasik(1.0,0.0) yapıcısı ile imal edildi.
    c3.yaz();
    Karmasik c4=c2+c3; // c4 kopya yapıcısı ile imal edildi.
    c4.yaz();
    c4=c2+c3; // c4 kopya yapıcısı ile imal edildi.
    c4.yaz();
}
```

Yapıcıların aşırı yüklenmesinin çeşitli üstünlükleri vardır;

- Nesne imal etmede esneklik: Aynı sınıftan farklı durumlara sahip nesneler imal edilebilir.
- Gelişmiş kod bakımı ile daha temiz ve okunabilir kod oluşur.
- Nesne imal etme mantığı diğer nesnelerden/programlardan gizlenir.
- Kopya yapıcılar ile nesne klonlama basitleşir.

Taşıma Yapıcısı

Bir kopya yapıcı (copy constructor) yazmak için, yani bir nesneyi kopyalayan ve yeni bir nesne oluşturan bir fonksiyon oluşturmak için, normalde aşağıda gösterilen sözdizimini seçerdik;

```
class A {
public:
    int a;
    int b;
    A(const A &other) {
        this->a = other.a;
        this->b = other.b;
    }
};
```

A için A tipinde başka bir nesneye referans alan bir yapıcımız olurdu ve nesneyi yöntemin içinde elle kopyalardık. Alternatif olarak, tüm üyeleri otomatik olarak kopyalayan `A(const A &) = default;` yazabilirdik.

Ancak bir taşıma yapıcısı (move constructor) oluşturmak için lvalue referansı yerine rvalue referansı alacağız. Yani referans alınan nesne durumları yeni nesneye taşınıyor. Örneğin;

```
class Cuzdan {
public:
    int para;
    Cuzdan () = default; //ön tanımlı yapıcı (default constructor)
    Cuzdan (Cuzdan &&diger) {
        this->para = diger.para;
        diger.para = 0;
    }
};
```

Burada durumları taşınan referans nesnenin durumunun sıfırlandığına dikkat edilmesi gerekir. Taşıma semantiği orijinal örnekten (instance) 'durum çalmaya' izin verecek şekilde tasarlanmıştır. Orijinal örneğin bu çalmadan sonra nasıl görünmesi gerektiğini üzerinde durulmalıdır. Bu durumda, değeri sıfıra değiştirmeseydik tüm nesnelerdeki para miktarını iki katına çıkarmış olurduk;

```
Cuzdan a;
a.para = 10;
Cuzdan b (std::move(a)); // B(B&& diger); yapıcısı çağrılır.
std::cout << a.para << std::endl; //0
std::cout << b.para << std::endl; //10
```

Böylece eski bir nesneden yeni bir nesne imal etmiş olduk.

This Göstericisi Kelimesi

C++'da her nesne, `this` adı verilen önemli bir gösterici aracılığıyla kendi adresine erişebilir. `this` göstericisi (this pointer) tüm üye yöntemler için örtük bir parametredir. Aşağıda karmaşık sayıları örneği verilmiştir;

```
#include <iostream>
using namespace std;

class Karmasik {
    float gercek, sanal;
public:
    Karmasik(float pGercek=0.0, float pSanal=0.0) {
        this->gercek=pGercek;
        this->sanal=pSanal;
    }
    void yaz() { //yaz yöntemi
        cout<< this->gercek << "+" << this-> sanal << +"i" << endl;
    }
    Karmasik operator+(const Karmasik& pKarmasik) {
        Karmasik toplam;
        toplam.gercek = this->gercek + pKarmasik.gercek;
        toplam.sanal = this->sanal + pKarmasik.sanal;
        return toplam;
    }
    Karmasik operator+(const float pGercek) {
        Karmasik toplam;
        toplam.gercek = this->gercek + pGercek;
        toplam.sanal = this->sanal;
        return toplam;
    }
    Karmasik(const Karmasik& pKarmasik) {
```

```
        this->gercek= pKarmasik.gercek;
        this->sanal=pKarmasik.sanal;
    }
};
int main(void) {
    Karmasik c1(1.0,2.0);
    c1.yaz();
    Karmasik c2(3.0,5.0);
    c2.yaz();
    Karmasik c3=c1+c2;
    c3.yaz();
    Karmasik c4;
    c4=c2+c3;
    c4.yaz();
}
```

Friend Anahtar Kelimesi

Bir sınıfın arkadaş (friend) yöntemi, o sınıfın bloğu dışında tanımlanır ancak sınıfın tüm mahrem (private) ve korumalı (protected) üyelerine erişim hakkına sahiptir. Arkadaş yöntemlerin bildirimleri/prototipleri sınıf içerisinde yer almasına rağmen, arkadaşlar üye fonksiyon değildir.

```
#include <iostream>
using namespace std;

class Karmasik {
    float gercek, sanal;  //private members
public:
    Karmasik(float pGercek=0.0, float pSanal=0.0) {
        this->gercek=pGercek;
        this->sanal=pSanal;
    }
    friend void yaz(Karmasik pKarmasik);
};
void yaz(Karmasik pKarmasik) {
    cout << pKarmasik.gercek << "+" <<pKarmasik.sanal <<"i" <<endl;
}
int main(void) {
    Karmasik c1(1.0,2.0);
    yaz(c1); // 1+2i
    Karmasik c2(3.0,5.0);
    yaz(c2); // 3+5i
}
```

Bu durum bir sınıfa akış (stream) nesnelerinden veri çıkarma işleci (stream extraction operator) olan `>>` ile akışlara veri ekleme işleci (stream insertion operator) olan `<<` için çok kullanılırlar. Karmaşık sayı sınıfı için aşağıdaki gibi bir örnek verilebilir;

```
#include <iostream>
using namespace std;

class Karmasik {
private:
float gercek, sanal;  //private members
public:
    Karmasik(float pGercek=0.0, float pSanal=0.0) {
        this->gercek=pGercek;
        this->sanal=pSanal;
    }
    friend ostream& operator<<(ostream& output, const Karmasik& pKarmasik) {
        /*
```

```
        ostream sınıfına, pKarmasik nesnesindeki mahrem (private) ve
        korumalı (protected) üyelere erişme yetkisi, friend anahtar
        kelimesiyle verilmiştir.
        */
        output << pKarmasik.gercek << "+" << pKarmasik.sanal << +"i";
        return output;
    }

};
int main(void) {
    Karmasik c1(1.0,2.0);
    cout << c1 << endl;  // 1+2i
    Karmasik c2(3.0,5.0);
    cout << c2 << endl;  // 3+5i
}
```

Statik Sınıf Üyeleri

C++ Dilinde statik üye yöntemi (static member method), imal edilecek herhangi bir nesneye değil, sınıfın kendisine ait olan özel bir fonksiyon türüdür. Bu yöntemleri tanımlamak için `statik` anahtar kelimesi kullanılır. Sınıfın bir örneği olacak nesne imal edilmesine gerek kalmadan, sadece sınıf kimliğini kullanarak doğrudan çağrılabilirler. Bu tür yöntemlere yardımcı yöntem (utility method) adı verilir. Bu yöntemler program çalışmaya başladığında var olur ve program bitinceye kadar erişilebilirler. Ayrıca yalnızca sınıfın statik üyeleri veya diğer statik fonksiyonlarıyla çalışabilirler.

```
#include <iostream>
using namespace std;

class YardimciSinif {
public:
    static void programciAdi() {
        cout << "Programcı Adı: Ilhan OZKAN" <<endl;
        cout << "Tarih:" << __DATE__  << endl;
    }
};

int main(void) {
    YardimciSinif::programciAdi(); /* static üye yöntemi çağırma;
    SınıfKimligi::yöntemadi() şeklinde kapsam çözümleme işleci :: ile
    çağrılır. */
}
```

Statik yöntemlere benzer şekilde statik alanlar da tanımlanabilir. Statik olan bu veriler veri bellekte (data segment) saklanırlar ve bu veri üyeleri de statik yöntemler gibi kullanılır. Statik üyeler genellikle bir sınıfın tüm örnekleri arasında paylaşılması gereken paylaşılan kaynakları veya sayaçları yönetmek için kullanılır.

İleride anlatılacak olan tasarım desenlerinden (design pattern) tekil nesne deseni (singleton pattern) statik üyelere örnek olarak verilebilir. Bu desende sınıftan yalnızca bir nesne/örnek imal edilen ve bu tekil nesneye evrensel erişim sağlayan bir tasarım desenidir. Burada statik üyeler, sınıfın tek bir paylaşılan örneğinin bakımına izin verdikleri için bu deseni uygulamak için mükemmeldir.

Const Sınıf Üyeleri

Sınıfların `const` yöntemleri (const method), veri üyelerinin yani durumları (state) tutan alanları (field) değiştirme izni verilmeyen fonksiyonlardır. Bir üye fonksiyonunu `const` yapmak için, const anahtar kelimesi fonksiyon prototipine ve ayrıca fonksiyon tanımındaki başlığına eklenir.

Üye yöntemler gibi bir sınıfın nesneleri de `const` olarak bildirilebilir. `const` olarak bildirilen bir nesne değiştirilemez ve bu nedenle, bu fonksiyonlar nesneyi değiştirmemeyi garantilediğinden, yalnızca const üye yöntemlerini çağırabilir.

Bir `const` nesnesi, nesne bildirimine `const` anahtar sözcüğünü önek olarak ekleyerek tanımlanabilir. `const` nesnelerinin veri üyesini değiştirmeye yönelik herhangi bir girişim, derleme zamanı hatasıyla (compile time error) sonuçlanır.

```
#include <iostream>
using namespace std;

class Sinif {
private:
    int alan;
public:
    Sinif() {
        alan=-1;
    }
    int getAlan() const {
        return alan;
    }
    void setAlan(int pAlan) { // Bu yöntem const tanımlanılırsa hata alınır!
        alan=pAlan;
    }
};

int main() {
    Sinif* nesne=new Sinif();
    cout<<"nesne.alan:" << nesne->getAlan() << endl;
    nesne->setAlan(12);
    cout<<"nesne.alan:" << nesne->getAlan() << endl;
    delete nesne;
    const Sinif* constNesne=new Sinif();
    constNesne->setAlan(9); // Hata cont nesne değiştirilemez!
}
```

Mutable Depolama Sınıfı

Bazen, `const` ile sabit olarak tanımlanmış nesne veya yapının (struct) bir veya daha fazla veri üyesini değiştirme gereksinimi doğabilir. İşte bu durumda bu üyeler `mutable` anahtar kelimesiyle tanımlanır.

```
#include <iostream>
using namespace std;
class Kisi {
public:
    int tcno;
    mutable int okulno;

    Test(){
        tcno = 43233445505;
        okulno = -1;
    }
};
int main(){
   const Kisi ogrenci1; // ogrenci1 nesnesi const olarak tanımlandı

   ogrenci1.okulno = 2000; // sabit olan nesnenin okulno değiştiriliyor.
   cout << ogrenci1.okulno;

   ogrenci1.tcno = -1; //HATA: sabit nesnenin alanı değiştirilemez.
}
```

Nesne Yıkıcı

Yıkıcı (destructor), bir nesne yok edileceği zaman otomatik olarak çağrılan yapıcı gibi bir yöntemdir. Bir sınıf içinde yalnızca bir yıkıcı tanımlanabilir. Yani, bir yıkıcı, bir nesne yok edilmeden önce çağrılacak son yöntemdir. Aşağıdaki gibi tanımlanır.

```
~sınıf-kimliği() {
 // ...
}
```

Aşağıda imal edilen karmaşık sayıların nasıl yok edildiğine ilişkin örnek verilmiştir;

```
#include <iostream>
using namespace std;

static int nesneSayaci=0;

class Karmasik {
    float gercek, sanal;
public:
    Karmasik(float pGercek=0.0, float pSanal=0.0): gercek(pGercek),sanal(pSanal) {
        nesneSayaci++;
        cout << nesneSayaci << " nesne imal edildi" << endl;
    }
    void yaz() { //yaz yöntemi
        cout<< gercek << "+" << sanal << +"i" << endl;
    }
    Karmasik operator+(const Karmasik& pKarmasik) {
        Karmasik toplam;
        toplam.gercek = gercek + pKarmasik.gercek;
        toplam.sanal = sanal + pKarmasik.sanal;
        return toplam;
    }
    Karmasik operator+(const float pGercek) {
        Karmasik toplam;
        toplam.gercek = gercek + pGercek;
        toplam.sanal = sanal;
        return toplam;
    }
    Karmasik(const Karmasik& pKarmasik) {
        gercek= pKarmasik.gercek;
        sanal=pKarmasik.sanal;
    }
    ~Karmasik() {
        nesneSayaci--;
        cout << nesneSayaci << " nesne yok edildi" << endl;
    }
};
int main(void) {
    Karmasik c1(1.0,2.0);
    c1.yaz();
    Karmasik c2(3.0,5.0);
    c2.yaz();
    Karmasik c3=c1+c2;
    c3.yaz();
    Karmasik c4;
    c4=c2+c3;
    c4.yaz();
}
/*Program Çıktısı:
1 nesne imal edildi
```

```
1+2i
2 nesne imal edildi
3+5i
3 nesne imal edildi
4+7i
4 nesne imal edildi
5 nesne imal edildi
4 nesne yok edildi
7+12i
3 nesne yok edildi
2 nesne yok edildi
1 nesne yok edildi
0 nesne yok edildi

...Program finished with exit code 0
*/
```

Geçersiz Kılma

Yöntemi geçersiz kılma (overriding), nesne yönelimli programlamanın, taban sınıfta önceden tanımlanmış bir yöntemi türemiş bir sınıfta yeniden tanımlamasına olanak tanıyan bir kavramıdır. Yani taban sınıfın davranışının türemiş sınıfta değiştirilmesidir. C++ dili iki tür fonksiyon geçersiz kılmayı destekler:

- Derleme Zamanı Geçersiz Kılma
- Çalışma Zamanı İşlevi Geçersiz Kılma

Derleme zamanında geçersiz kılma aşağıdaki şekilde yapılır;

```
class Taban-Sınıf-Kimliği {
erişim-belirleyici:
     // geçersiz kılınacak yöntem:
     geri-dönüş-veri-tipi yöntem-kimliği() {
     }
};

class Türemiş-Sınıf-Kimliği: erişim-belirleyici Taban-Sınıf-Kimliği {
erişim-belirleyici:
     // geçersiz kılan yöntem:
     geri-dönüş-veri-tipi yöntem-kimliği() {
     }
};
```

Her iki sınıf ta da yöntemin kimliği aynıdır. Taban sınıftan imal edilen nesne taban sınıfın yöntemini, türemiş sınıftan imal edilen nesne ise türemiş sınıfın yöntemini icra eder.

```
#include <iostream>
using namespace std;

class Baba {
public:
    void yap()
    {
        cout << "Baba şu şekilde yapar ..." << endl;
    }
};
class Cocuk : public Baba {
public:
    void yap()
    {
        cout << "Çocuk şu şekilde yapar..." << endl;
    }
```

```
};
int main() {
    Baba baba;
    baba.yap(); // Baba şu şekilde yapar ...
    Cocuk cocuk;
    cocuk.yap(); // Çocuk şu şekilde yapar...
}
```

Çalışma zamanında geçersiz kılma aşağıdaki şekilde yapılır;

```
class Taban-Sınıf-Kimliği {
erişim-belirleyici:
     // geçersiz kılınacak yöntem:
     virtual geri-dönüş-veri-tipi yöntem-kimliği() {
     }
};

class Türemiş-Sınıf-Kimliği: erişim-belirleyici Taban-Sınıf-Kimliği {
erişim-belirleyici:
     // geçersiz kılan yöntem:
     geri-dönüş-veri-tipi yöntem-kimliği() override {
     }
};
```

Her iki sınıf ta da yöntemin kimliği aynıdır. Geçersiz kılınacak yöntem, taban sınıfın yöntemini türemiş sınıfta değiştirilebileceğini belirtmek için `virtual` anahtar kelimesiyle tanımlanır. Türemiş sınıfta ise geçersiz kılacağı yöntemi `override` anahtar kelimesiyle yeniden tanımlar.

```
#include <iostream>
using namespace std;

class Baba {
public:
    virtual void yap()
    {
        cout << "Baba şu şekilde yapar ..." << endl;
    }
};
class Cocuk : public Baba {
public:
    void yap() override
    {
        cout << "Çocuk şu şekilde yapar..." << endl;
    }
};
int main() {
    Cocuk cocuk;
    cocuk.yap(); // Çocuk şu şekilde yapar...
    Baba* babaPtr=&cocuk;
    babaPtr->yap(); // Çocuk şu şekilde yapar...
/*
    Baba& babaPtr=cocuk;
    babaPtr.yap(); // Çocuk şu şekilde yapar...
*/
}
```

Roller

Ara yüzler (interface) nesnelerin sahip oldukları rollerdir (role). C++ dilinde roller, sınıf (class) olarak kodlanır. Ancak saf nesne yönelimli dillerde ara yüzler (interface) `interface` saklı kelimesiyle tanımlanırlar.

Örnek olarak bir öğretmen; Evde baba rolünde olup babanın sahip olduğu çocuklara bakma, yemek yapma gibi davranışları gösterir. İşte öğretmen rolündedir ve öğrenmenin sahip olduğu; öğretme, sınav/değerlendirme yapma, karne hazırlama, ders notu hazırlama gibi davranışları vardır. Bu öğretmen müzisyen rolünde ise gitar çalma, akort yapma gibi davranışları da vardır. İşte bunların hepsi ayrı bir roldür ve her bir rolde o role ilişkin davranışları sergiler.

C++ Dilinde, saf nesne yönelimli dillerin (Java, C# gibi) aksine çoklu kalıtımı (multiple inheritance) destekler gibi görünür, çünkü `interface` gibi bir anahtar kelimeye sahip olmadığından ara yüz olarak tanımlanmış birçok sınıftan miras alabilir.

Ara yüzler, soyut (abstract) sınıflarda tanımlanır ve veri soyutlaması (data abstraction) yapılmaz. Bir sınıfın soyut olabilmesi için en az bir saf sanal yönteme (pure virtual method) sahip olmalıdır. Sanal sınıflardan nesne imal edilemez! Sanal sınıflardaki sanal yöntemler (virtual method) türeyen sınıfta yeniden tanımlanabilir. Türeyen sınıfta mutlaka geçersiz kılınır (override).

```
#include <iostream>
using namespace std;

class IBaba { //Baba Rolündeki davranışlar:
public:
   virtual void cocugaBak() = 0; // saf sanal yöntem-pure virtual method
   virtual void hanimaYardimEt() = 0;
   /* Bu sınıfta en az bir saf sanal yöntem olduğundan
      bu sınıftan nesne imal edilemez! */
};
class IMuzisyen { //Müzisyen Rolündeki Davranışlar:
public:
   virtual void gitarCal() = 0; // saf sanal yöntem-pure virtual method
   virtual void piyanoCal() = 0;
   /* Bu sınıfta en az bir saf sanal yöntem olduğundan
      bu sınıftan nesne imal edilemez! */
};
class IOgretmen { //Öğretmen Davranışları:
public:
   virtual void ogret() = 0; // saf sanal yöntem-pure virtual method
   virtual void dersNotuHazirla() = 0;
   /* Bu sınıfta en az bir saf sanal yöntem olduğundan
      bu sınıftan nesne imal edilemez! */
};
class Kisi:  public IBaba, public IMuzisyen, public IOgretmen {
    /*
      (interface realization/implementation):
      Gerçekleştirilen IBaba, IMuzisyen ve IOgretmen arayüzlerinde
      saf sanal yöntemler olduğundan bu Kişi sınıfında hepsi
      geçersiz kılınmalıdır (override).
    */
public:
    string adi;
    string soyadi;
    void adiniSoyle() {
        cout << adi << " " << soyadi << endl;
    }
    // IBaba aracılığıyla miras alınan davranışlar:
    void cocugaBak() override {
        cout << "Çocuğa Bakıyorum..." << endl;
    }
    void hanimaYardimEt() override {
        cout << "Hanıma Yardım Ediyorum..." << endl;
    }
    // IMuzisyen aracılığıyla miras alınan davranışlar:
    void gitarCal() override {
```

```
        cout << "Gitar Çalıyorum..." << endl;
    }
    void piyanoCal() override {
        cout << "Piyano Çalıyorum..." << endl;
    }
    // IOgretmen aracılığıyla miras alınan davranışlar:
    void ogret() override {
        cout << "Öğretiyorum..." << endl;
    }
    void dersNotuHazirla() override {
        cout << "Ders notu hazırlıyorum..." << endl;
    }
};
int main() {
    Kisi ilhan;
    ilhan.adi = "İlhan";
    ilhan.soyadi = "ÖZKAN";

    //imal edilen nesne tüm arayüzdeki davranışları gösterebilir:
    ilhan.adiniSoyle(); // İlhan ÖZKAN
    ilhan.ogret(); // Öğretiyorum...
    ilhan.gitarCal(); // Gitar Çalıyorum...
    ilhan.hanimaYardimEt(); // Hanıma Yardım Ediyorum...

    //imal edilen nesne sadece bir roldeki davranışları da gösterebilir:
    IBaba& baba=ilhan;
    baba.cocugaBak(); // Çocuğa Bakıyorum...
    baba.hanimaYardimEt(); // Hanıma Yardım Ediyorum...

    IMuzisyen& muzisyen = ilhan;
    muzisyen.gitarCal(); // Gitar Çalıyorum...
    muzisyen.piyanoCal(); // Piyano Çalıyorum...

    IOgretmen& ogretmen = ilhan;
    ogretmen.dersNotuHazirla(); // Ders notu hazırlıyorum...
    ogretmen.ogret(); // Öğretiyorum...
}
```

Bir sınıfın bir ara yüzdeki davranışları kendine uygulamasına ara yüz gerçekleştirme (interface realization/interface implementation) adı verilir.

Çok Biçimlilik

Çok biçimlilik (polymorphism), birden çok nesnenin olduğu bir ortamda, verilen aynı iletiye (message) karşı, nesnelerin farklı davranış (behavior göstermeleridir (same message-different behavior).

Çok biçimliliğin uygulanabilmesi (implementation) için üç şart vardır;

- Kalıtım (inheritance): Ortak davranışın tanımlandığı taban sınıf ve bu davranışın değiştiği türeyen sınıflar.
- Nesnelere aynı iletiyi verebilmek için ortak davranışı tanımlayan Sanal Yöntem (virtual method).
- Geçersiz Kılan Yöntem (override method): Devralınan davranışı kendine uyarlayan ve devralınan yöntemi geçersiz kılan yeni bir yöntem.

```
#include <iostream>
using namespace std;

class SoyutKisi { //Soyut Kisi Sınıfı
public:
    virtual void raporYaz() = 0;
    /* saf sanal metot:Bu metot, Kişi sınıfını SOYUT yapar. Ayrıca;
```

```
     raporYaz(): Türeyen sınıflardan imal edilecek nesnelere
     aynı mesajı vermek için tanımlanan ortak davranıştır.
    */

    // adi özelliği için veri soyutlaması (data abstraction) yapılıyor.
    string getAdi() {
        return adi;
    }
    void setAdi(string pAdi) {
        if (pAdi!="") this->adi=pAdi;
    }
    SoyutKisi(string pAdi): adi(pAdi) {
    }
private:
    string adi;
};

class Ogrenci: public SoyutKisi {  //Ortak davranış SoyutKisi sınıfından devralınıyor.
public:
    void raporYaz() override { //devralınan yöntemler geçersiz kılınıyor (override)
        cout << getAdi() <<":Öğrenci Ödevilerine İlişkin Rapor Yazıyor..." << endl;
    }
    Ogrenci(string pOgrenciAdi):SoyutKisi(pOgrenciAdi){
    }
};

class Ogretmen: public SoyutKisi {  //Ortak davranış SoyutKisi sınıfından devralınıyor.
public:
    void raporYaz() override { //devralınan yöntemler geçersiz kılınıyor (override)
        cout << getAdi() <<":Ogretmen Verdiği Derslere İlişkin Rapor Yazıyor..."
             << endl;
    }
    Ogretmen(string pOgretmenAdi):SoyutKisi(pOgretmenAdi){
    }
};

class Mudur: public SoyutKisi {  //Ortak davranış SoyutKisi sınıfından devralınıyor.
public:
    void raporYaz() override { //devralınan yöntemler geçersiz kılınıyor (override)
        cout << getAdi() <<":Müdür Akademik Takvime İlişkin Rapor Yazıyor..."
             << endl;
    }
    Mudur(string pMudurAdi):SoyutKisi(pMudurAdi){
    }
};
int main() {
    SoyutKisi* kisi[3];
    kisi[0]=new Ogrenci("İlhan ÖZKAN");
    kisi[1]=new Ogretmen("Hasan YILMAZ");
    kisi[2]=new Mudur("Recep AY");

    for (int i=0; i<3; i++)
        kisi[i]->raporYaz();
        /*
         Her kişiye aynı "raporYaz()" mesajı veriliyor->
         Karşılığında farklı davranışlar sergileniyor.
        */

    for (int i=2; i>=0; i--)
        delete kisi[i];
}
```

```
/* Program Çıktısı:
İlhan ÖZKAN:Öğrenci Ödevilerine İlişkin Rapor Yazıyor...
Hasan YILMAZ:Ogretmen Verdiği Derslere İlişkin Rapor Yazıyor...
Recep AY:Müdür Akademik Takvime İlişkin Rapor Yazıyor...

...Program finished with exit code 0
*/
```

Çok biçimlilik bize çalıştırma anında (run time) üstünlük sağlar. Bu nedenle nesneler de çalıştırma anında imal edilmiştir. Program çalıştırıldığında her `kisi` nesnesine aynı ileti verilmiş ama bu nesneler farklı davranış göstermiştir.

Yazılımcı için çok biçimlilik, farklı tipte nesnelerin aynı isimli yöntemleri (method) çağrıldığında her bir nesnenin kendine özgü davranışı gerçekleştirme yeteneği olup çalıştırma anında üstünlük sağlar. Yani yazılımcı, nesnenin tipine bakmaksızın, nesnelerden aynı davranışı göstermesini ister. Burada gösterilecek davranışın, taban sınıfın davranışı mı olduğu veya türeyen sınıfın davranışı mı olduğuna çalıştırma anında (run time) karar verilir. İşte bu karar verme işlemine geç bağlama (late binding) veya dinamik bağlama (dynamic binding) adı verilir.

Esnekliğin istenmediği bazı durumlarda ise karar verme işlemi derleme anında yapılır. Buna ise statik bağlama (static binding) adı verilir. İkisi arasındaki seçim programcı tarafından yapılır.

Yazılım yapılacak sistemin başlangıç koşullarına bağlı tasarım kararlarının tümü bilinemez. Ayrıca sistemin genişlemesi için gerekli tüm kodlamanın bitirilmesi beklenemez. İşte bunu gerçekleştiren dinamik bağlama, yazılım mimarisinin daha esnek (mevcut bileşenin sistemde yeniden yapılandırmasının kolayca yapılması) ve genişleyebilir (yeni bileşenlerin kolayca sisteme eklenmesi) olmasını sağlar.

Sınıf Çeşitleri

Soyut sınıfların (abstract class) bir örneği (instance) olamaz. Yani bu sınıftan nesne imal edilemez. Bu sınıfın en az bir soyut yöntemi (abstract method) olur. Soyut sınıflar hangi davranışın gösterileceğini belirler ama bu davranışların nasıl gösterileceğini anlatmaz. Bu nedenle soyut yöntem için gövde tanımlanmaz.

```
#include <iostream>
using namespace std;
class SoyutKisi { //Soyut Kisi Sınıfı
public:
    virtual void raporYaz() = 0;  // Bu saf sanal yöntem sınıfı soyut yapar.
    string getAdi() {
        return adi;
    }
    void setAdi(string pAdi) {
        if (pAdi!="") this->adi=pAdi;
    }
    SoyutKisi(string pAdi): adi(pAdi) {
    }
private:
    string adi;
};
int main() {
    // SoyutKisi soyutkisi; //HATA: Soyut sınıftan nene imal edilemez!
}
```

Somut sınıflardan (concrete class) nesne imal edilebilir yani örneği (instance) olabilir. Bu sınıflarda davranış ve durumlar eksiksiz tanımlıdır. Bir sınıf tanımlarken soyut bir sınıfa genel davranış ve durumları toplamak ve ondan miras alarak somut sınıfları tanımlamak her zaman iyi bir seçimdir.

```
#include <iostream>
using namespace std;
```

```
class SoyutKisi { //Soyut Kisi Sınıfı
public:
    virtual void raporYaz() = 0; // Saf sanal yöntem. Bu sınıfı soyut yapar.
    string getAdi() {
        return adi;
    }
    void setAdi(string pAdi) {
        if (pAdi!="") this->adi=pAdi;
    }
    SoyutKisi(string pAdi): adi(pAdi) {
    }
private:
    string adi;
};

class SomutKisi: public SoyutKisi {
    //Ortak davranış SoyutKisi sınıfından devralınıyor.
public:
    void raporYaz() override {
        cout << getAdi() <<": Rapor Yazıyor..." << endl;
    }
    // Bu sınıfta sanal yöntem olmadığından nesne imal edilebilir.
    SomutKisi (string pAdi):SoyutKisi(pAdi){
    }
};
int main() {
    SomutKisi somutKisi("Ilhan");
    somutKisi.raporYaz(); // "Ilhan: Rapor Yazıyor..."
}
```

Taban sınıf (base class), miras alınan sınıf veya genel özellik ve davranışların toplandığı yani genelleştirmenin yapıldığı genel sınıf (general class) veya ebeveyn sınıftır (parent).

Türemiş sınıf (derived class), miras alan sınıf veya davranış ve durumlar hakkında uzmanlaşmış yani uzman sınıf (special class) veya çocuk sınıftır (child class). Türemiş ya da miras almış sınıf.

Kök sınıf (root class), babası olmayan ya da yetim sınıf olup aynı zamanda baba sınıftır.

Yaprak sınıf (leaf class), kendisinden henüz miras alan bir sınıf olmayan bir sınıftır. Kısaca türememiş sınıftır.

Kısır sınıf (sealed class) kendisinden türeyen bir sınıf tanımlayamayan bir sınıftır. Kısaca türeyemeyen bir sınıftır. C++ diline 2011 yılında bunu yapabilmek için `final` anahtar kelimesi (final keyword) eklenmiştir.

```
#include <iostream>
using namespace std;

class Dikdortgen final { //Somut Dikdirtgen Sınıfı
public:
    virtual float alanHesapla() {
        return en*boy;
    };
    Dikdortgen(float pEn,float pBoy): en(pEn),boy(pBoy) {
    }
private:
    float en, boy;
};

class Kare: public Dikdortgen {  //HATA: Dikdörtgen sınıfı final ile tanımlanmış
public:
    Kare(float pEn) : Dikdortgen(pEn, pEn) {
```

```
    }
};
int main() {
    Dikdortgen dikdortgen(4.0,5.0);
    cout << "Dikdörgenin alanı:" << dikdortgen.alanHesapla() << endl;
}
```

Final anahtar kelimesi bir sınıfta sadece yöntemler içinde kullanılabilir. Bu durumda türeyen sınıfta bu yöntem geçersiz kılınamaz (override);

```
#include <iostream>
using namespace std;

class Dikdortgen { //Somut Dikdirtgen Sınıfı
public:
    virtual float alanHesapla() {
        return en*boy;
    };
    virtual float kisaKenar() final {
        return (en>boy)?boy:en;
    }
    Dikdortgen(float pEn,float pBoy): en(pEn),boy(pBoy) {
    }
private:
    float en, boy;
};

class Kare: public Dikdortgen {
public:
    float kisaKenar() overide {
        //HATA: Dikdörtgen sınıfında kisaKenar yöntemi final ile tanımlanmış
        //...
        return 0.0;
    }
    Kare(float pEn) : Dikdortgen(pEn, pEn) {
    }
};
int main() {
    Dikdortgen dikdortgen(4.0,5.0);
    cout << "Dikdörgenin kısa kenarı:" << dikdortgen.kisaKenar() << endl;
}
```

Tekil sınıftan (singleton class) yalnızca bir nesne imal edilebilir. Yani yalnızca bir örneği vardır. Bunun olabilmesi için mahrem (private) yapıcısı olması gerekir. *Tekil Nesne* başlığında incelenecektir.

Yardımcı sınıf (utility class), üyeleri statik olan sınıflardır. Ayrıca evrensel yapılandırma ayarlarını (configuration settings) veya değişmezleri (literal) depolamak için kullanılabilir. Bir kaynak havuzunu (önbellek, veri tabanı bağlantı havuzu, vb.) yönetmek ve örnekler arasında paylaşılan bir günlük sistemi uygulamak için yararlıdır. Bunların dışında, izleme yöntemi (trace method) çağrıları için de kullanılabilir.

Unified Modeling Language

Unified Modeling Language (UML) nesneye dayalı yazılım geliştirme ile birlikte gelişen bir modelleme aracıdır. Bu nedenle Nesne Yönelimli Programlama başlığında açıklanan nesne yönelimli birçok kavram, izleyen sayfalarda daha da pekişecektir. Adından da anlaşılacağı üzere nesneye dayalı problem çözmenin kalbi, model oluşturmaktır.

Model, gerçek dünyadaki karmaşık problemin esaslarını soyut olarak önümüze koyar. Buradan hareketle UML'i, basit bir dil, gösterim ya da sözdizimi (syntax) gibi algılamalıyız. UML'in, yazılımı nasıl geliştireceğimizi kesin olarak belirtmediği göz önüne alınmalıdır.

UML, bir grafik modelleme dili olup yazılım sistemlerini oluşturan ana elemanları (Ki bunlar İngilizce "artifact" olarak adlandırılıp ve el yapımı anlamına gelmektedir.) çeşitli şekillerden oluşacak şekilde tanımlamak için oluşturulmuş bir kurallar bütünüdür. Örnek verecek olursak, bir mimari projeye ilişkin maket ne ise yazılım için UML de aynı şeyi ifade eder. Yani UML, söz konusu yazılımın neyi yapacağını anlatan çeşitli şekillerin anlamlı bir şekilde bir araya getirilmesini sağlayan bir standarttır.

Yazılım mühendisliğiyle (software engineering) birlikte 1989 ve 1994 yılları arasında elliden fazla modelleme dili kullanılmaya başlanmıştır. Bu nedenle bu dönem yöntem savaşlarının yaşandığı bir dönem olarak adlandırılır. Bu yöntemlerin çoğu kendi sözdizimini oluşturmuş olmakla birlikte kullandıkları dil elemanları açısından birbirleriyle benzerlik göstermektedirler.

Doksanlı yılların ortalarında üç yöntem, daha güçlü ve yaygın hale gelmiştir. Bu yöntemlerin her biri, diğerlerinin içerdiği elemanları da içeriyordu ancak her birinin diğerine karşı çeşitli güçlü özellikleri bulunuyordu. Bu yöntemler:

– *Booch* yönteminin tasarım (design) ve kodlama (implementation) yönleri oldukça iyi idi. *Grady Booch* Ada diliyle oldukça çalışmıştı ve nesne yönelimli tekniklerin Ada diline uygulanması konusunda yaptığı çalışmalarla bu konuda önemli bir oyuncu olmuştur. Booch metodu güçlü olmasına rağmen model, birçok bulut şeklinden oluşmasından dolayı gösterim şekli oldukça sevimsizdi.
– Object Modeling Technique (OMT) yöntemi, *Jim Rumbaugh* tarafından geliştirilmiş olup, analiz ve veri merkezli bilgi sistemlerinin modellenmesinde oldukça iyi bir yöntemdir.
– Object Oriented Software Engineering (OOSE) modeli use-case olarak bilinen bir model olup *Ivar Jacobson* tarafından geliştirilmiştir. Use-case modeli, yazılımı yapılacak sistemin davranışlarını anlamaya yönelik gelişmiş güçlü bir tekniktir.

1994 yılında *Jim Rumbaugh* ve General Electric firmasından ayrılan *Grady Booch* birlikte Rational şirketini kurarak bir araya geldiler. Bu birlikteliğin amacı, kendi yöntemlerini bir araya getirecek yeni ve tek bir yöntem olan "Unified Method"u oluşturmak idi.

1995 yılında *Ivar Jacobson* da Rational şirketine katıldı. *Ivar Jacobson*'ın use-case'ler konusundaki fikirleriyle birlikte Rational şirketi, bugün Unified Modeling Language-UML olarak adlandırılan, yeni bir "Unified Method" geliştirdi. Üç kişiden oluşan bu grup "Three Amigos" olarak bilinir.

Yöntem savaşlarının ardından güçlü yazılım üreticileri bir araya gelerek OMG adında bir şirketler birliği kurdular[17]. Bu şirketler birliği 1997 yılında UML'e sahip çıkarak herkes tarafından kullanılan bir dil olmasını ve bir standart haline gelmesini sağlamıştır.

UML sadece, gerçek dünyadaki süreçlerin modelini ortaya koyan bir gösterim (notation) sistemidir. Süreçlerin standartlaştırılmasına yönelik bir dil değildir. UML ile ortaya çıkan modeller, süreçleri soyut (abstract) olarak tanımlayacaktır, ancak süreçlerin doğruluğu hakkında bir bilgi vermeyecektir. Yani, A şirketi için çalışan bir süreç, B şirketine uygulandığında felaketle sonuçlanabilir.

Model, yukarıda da anlatıldığı üzere çözülecek probleme ilişkin bir soyutlamadır (abstraction). Etki alanı (domain) ise problemin yaşandığı gerçek dünyayı ifade eder.

Model, birbirlerine ileti (message) gönderen nesnelerden (object) oluşur. Bu modelde nesneler canlı (alive) olarak düşünülmelidir. Nesneler bir şeyler bilir (know) ve bildikleriyle bir şeyler yaparlar (do).

Modelde nesneler, nesne tanımında verilen özellik (property) ve alanları (field) gösteren niteliklere (attribute) sahiptirler. Nesnelerin gösterdikleri davranış (behavior) ya da geliştirdiği yöntemler (method) ise işlem (operation) olarak adlandırılır. Nesnenin özelliklerine ait o anki değerler ise nesnenin durumunu (state) belirler.

UML, sekiz tür diyagrama sahiptir ve bu kitapta sınıf diyagramlarına yer verilecektir[18].

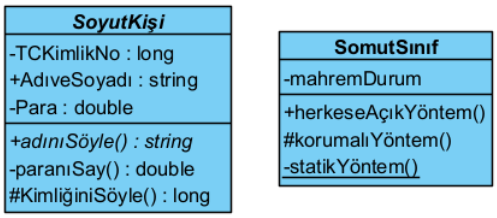
[17] OMG (Object Management Group), www.omg.org

[18] www.togethersoft.com/services/practical_guides/umlonlinecourse/index.html

Sınıf Diyagramları

Sınıf diyagramları (class diagram), yazılımı geliştirilecek olan sisteme ait sınıfları (class) ve bu sınıflar arasındaki ilişkiyi (relationship) gösterir. Sınıf diyagramları durağandır (static). Yani sadece sınıfların birbiriyle etkileşimini (what interacts) gösterir, bu etkileşimler sonucunda ne olduğunu (what happens) göstermez.

Sınıf diyagramlarında sınıflar, bir dikdörtgen ile temsil edilir. Bu dikdörtgen üç parçaya ayrılır. En üstteki birincisine sınıf ismi yazılır. Ortadaki ikincisinde nitelikler (attribute), en alttaki ve sonuncusunda ise ve işlemler (operation) yer alır.



Şekil 24. Örnek Bir Sınıf Diyagramı

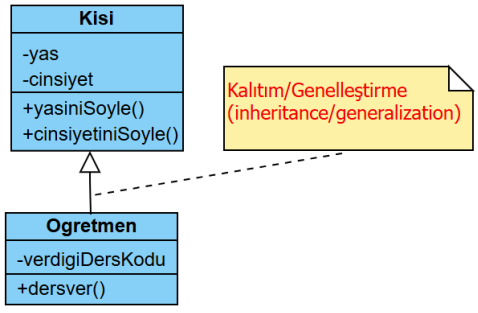
Sınıf isminin italik yazılması, sınıfın soyut (abstract class) sınıf olduğunu gösterir. Benzer şekilde işlemlerin yer aldığı kısımda, yöntemlerin italik yazılması halinde, ilgili yöntemin soyut yöntem (abstract method) olduğunu gösterir.

Sınıflarda erişim belirleyiciler (access modifier), özel karakterlerle birbirinden ayrılır. Burada "-" karakteri mahrem (private), "+" karakteri umuma açık (public), "#" karakteri ise korumalı (protected) olduğunu anlatmaktadır. Statik üyeler (static member), altı çizili olarak gösterilir.

Sınıflar Arası İlişkiler

Nesneleri imal ettiğimiz sınıflar arasında ya ilişki yoktur ya da aşağıda sıralanan altı çeşit ilişki vardır;

Genelleştirme (generalization), taban sınıfla (base class) ile türemiş sınıf (derived class) arasındaki kalıtım (inheritance) ilişkisidir. Örneği *Kalıtım* başlığında verilmiştir. Aşağıda Kişi sınıfı ile bu sınıftan miras alan Öğretmen sınıfının UML diyagramı verilmiştir. Ara yüzler (interface) veya roller de benzer şekilde gösterilir.



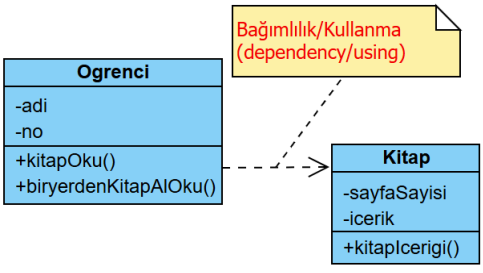
Şekil 25. Genelleştirme İlişkisi UML Diyagramı

Bağımlılık (dependency) ya da bir başka deyişle kullanma (using), bir sınıf ile diğer arasındaki geçici ilişkidir. Burada kullanan sınıf istemci (client) diğer sınıf ise sağlayıcı (supplier) olarak adlandırılır. İki türlü bağımlılık vardır; Birincisinde istemci sınıf, sağlayıcıyı sınıfı parametre olarak kullanır (dependency as a parameter). İkincisinde ise istemci sınıf, sağlayıcı sınıftan örnekleme yapar (dependency as an instantiation).

```
#include <iostream>
using namespace std;

class Kitap {
public:
    string kitapIcerigi() {
```

```
        return icerik;
    }
    Kitap(int pSayfaSayisi, string pIcerik): sayfaSayisi(pSayfaSayisi), icerik(pIcerik) {
    }
private:
    string icerik;
    int sayfaSayisi;
};
class Ogrenci {
public:
    void kitapOku() {
        Kitap* kitap=new Kitap(150,"C++ ile Nesne Yönelimli Programlama Notları");
        /* pKitap nesnesine örnekleme olarak bağımlılık:
           dependency as an instantiation */
        cout << "kitap imal edilip kullanılıyor..." << endl;
        string notlar=kitap->kitapIcerigi();
        //...
        delete kitap;
    }
    void biryerdenKitapAlOku(Kitap* pKitap) {
        /* pKitap nesnesine parametre olarak bağımlılık:
           dependency as a parameter */
        cout << "kitap dışardan alınıp kullanılıyor..." << endl;
        string okunacak=pKitap->kitapIcerigi();
        //...
    };

    Ogrenci(string pAdi,int pNo): adi(pAdi),no(pNo) {
    }
private:
    string adi;
    int no;
};
int main() {
    Ogrenci* ogrenci= new Ogrenci("Ali",12000);
    ogrenci->kitapOku();
    Kitap* kitap=new Kitap(350,"Kurumsal yazılım Geliştirme");
    ogrenci->biryerdenKitapAlOku(kitap);
    delete kitap;
    delete ogrenci;
}
/* Program Çıktısı:
kitap imal edilip kullanılıyor...
kitap dışardan alınıp kullanılıyor...

...Program finished with exit code 0
*/
```

Örnekteki Öğrenci ve Kitap sınıfı arasındaki bağımlılık ilişkisi aşağıdaki UML diyagramında verilmiştir.



Şekil 26. Bağımlılık İlişkisi UML Diyagramı

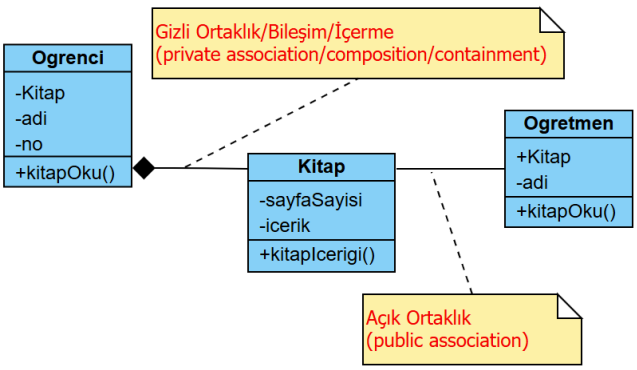
İş Birliği (association), iki farklı sınıf arasındaki sürekli olan bütün-parça (whole-part) ilişkisidir. Bu ilişkide, bir sınıf diğer sınıftan nesne örnekler (imal eder), kullanır ve öldürür. İlişkinin sürekli olabilmesi için kullanılan sınıftan bir değişken alan (field) olarak tanımlanır. İki türü vardır;

- Açık ortaklık (public association): Kullanılan sınıfa ilişkin değişken `public` erişimi ile tanımlanır.
- Gizli ortaklık (private association), içerme (containment) ya da bileşim (composition): Kullanılan sınıfa ilişkin değişken `private` erişimi ile tanımlanır ve dışarıdan erişime izin verilmez.

```
#include <iostream>
using namespace std;

class Kitap {
public:
    string kitapIcerigi() {
        return icerik;
    }
    Kitap(int pSayfaSayisi, string pIcerik): sayfaSayisi(pSayfaSayisi), icerik(pIcerik) {
    }
private:
    string icerik;
    int sayfaSayisi;
};
class Ogrenci { // kitapsız öğrenci olmaz! (private association to kitap)
private:
    string adi;
    int no;
    Kitap* ptrKitap;
public:
    Ogrenci(string pAdi,int pNo): adi(pAdi),no(pNo) {
        ptrKitap=new Kitap(150,"C ile Yapısal Programlama");
        //kitap nesnesi ile ogrenci nesnesi birlikte imal ediliyor:
    }
    void kitapOku() {
        cout << "Öğrenci ile birlikte imal edilen kitap kullanılıyor..." << endl;
        cout << "Okunan Kitap:" << ptrKitap->kitapIcerigi() << endl;
        //...
    }
};
class Ogretmen { // Kitapsız öğretmen olmaz! Ama kitabını değiştirebilir.
//public association to kitap
private:
    string adi;
    Kitap* ptrKitap;
public:
    Ogretmen(string pAdi): adi(pAdi) {
        //kitap nesnesinin referansı ile ogretmen nesnesi birlikte imal ediliyor:
        ptrKitap=new Kitap(200,"C++ ile Nesne Yönelimli Programlama");
    }
    // Kitap* üyesi get ve set yöntemleriye public hale getiriliyor.
    void setKitap(Kitap* pKitap) {
        ptrKitap=pKitap;
    }
    Kitap* getKitap() {
       return ptrKitap;
    }
    void kitapOku() {
        cout << "Öğretmen ile birlikte imal edilen kitap kullanılıyor..." << endl;
        cout << "Okunan Kitap:" << ptrKitap->kitapIcerigi() << endl;
        //...
    }
};
```

```
int main() {
    Ogrenci* ogrenci=new Ogrenci("Ali",12000);
    ogrenci->kitapOku();

    Ogretmen* ogretmen=new Ogretmen("Ilhan");
    ogretmen->kitapOku();

    Kitap* ogretmeninOkuduguKitap=ogretmen->getKitap();
    cout << "Orgetmenin Okuduğu Kitaba Main İçinden Ulaşılabiliyor;" << endl
         << "Okunan Kitap:" << ogretmeninOkuduguKitap->kitapIcerigi() << endl;

    Kitap* baskaKitap=new Kitap(250,"Kurumsal Yazılım Geliştirme");
    ogretmen->setKitap(baskaKitap);
    ogretmen->kitapOku();
    delete ogrenci, ogretmen, baskaKitap;
}
/*Program Çıktısı:
Öğrenci ile birlikte imal edilen kitap kullanılıyor...
Okunan Kitap:C ile Yapısal Programlama
Öğretmen ile birlikte imal edilen kitap kullanılıyor...
Okunan Kitap:C++ ile Nesne Yönelimli Programlama
Orgetmenin Okuduğu Kitaba Main İçinden Ulaşılabiliyor;
Okunan Kitap:C++ ile Nesne Yönelimli Programlama
Öğretmen ile birlikte imal edilen kitap kullanılıyor...
Okunan Kitap:Kurumsal Yazılım Geliştirme

...Program finished with exit code 0
*/
```

Yukarıdaki örnekte Öğrenci ile Kitap arasındaki gizli ortaklık ile Öğretmen ve Kitap arasındaki açık işbirliği aşağıdaki UML diyagramında gösterilmiştir;



Şekil 27. Açık ve Gizli İş Birliği UML Diyagramı

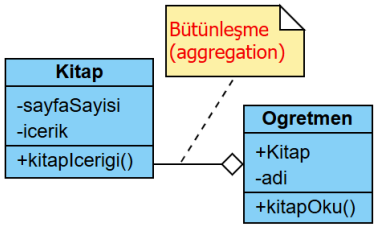
UML diyagramlarında, çift yönlü ortaklıklar iki oka sahip olabilir veya hiç ok olmayabilir ve tek yönlü ortaklıkta veya kendi kendine ortaklık bir oka sahiptir.

Bütünleşme (aggregation), ilişkisi de iki farklı sınıf arasındaki sürekli olan bütün-parça (whole-part) ilişkisidir. Açık ortaklığa (public association) benzer, farklı olarak bu ilişkide kullanılan sınıfa ait başka yerde imal edilmiş nesneyi ya da hazır olan nesneyi kullanır. Tek fark kullanılacak sınıf nesnesinin kullananın iradesi dışında yaratılmış olmasıdır.

```
#include <iostream>
using namespace std;

class Kitap {
public:
    string kitapIcerigi() {
        return icerik;
    }
```

```
    Kitap(int pSayfaSayisi, string pIcerik): sayfaSayisi(pSayfaSayisi), icerik(pIcerik) {
    }
private:
    string icerik;
    int sayfaSayisi;
};
class Ogretmen {
private:
    string adi;
    Kitap* ptrKitap;
public:
    Ogretmen(string pAdi,Kitap* ppKitap): adi(pAdi),ptrKitap(ppKitap) {
        //öğretme nesnesi kitap nesnesi olamadan imal edilemez.
        //Bir nesne imal edilmeden bi kitap nesnesi olmalı
        //aggregation to kitap
    }
    //Kitap* durumu get ve set metodlarıyla public hale getiriliyor.
    void setKitap(Kitap* pKitap) {
        ptrKitap=pKitap;
    }
    Kitap* getKitap() {
        return ptrKitap;
    }
    void kitapOku() {
        cout << "Öğretmen ile birlikte imal edilen kitap kullanılıyor..." << endl;
        cout << "Okunan Kitap:" << ptrKitap->kitapIcerigi() << endl;
        //...
    }
};
int main() {
    Kitap kitap= Kitap(150,"C++ ile Nesne Yönelimli Programlama");
    Ogretmen ilhan("Ilhan",&kitap);
    ilhan.kitapOku();
}
/* Program Çıktısı:
Öğretmen ile birlikte imal edilen kitap kullanılıyor...
Okunan Kitap:C++ ile Nesne Yönelimli Programlama

...Program finished with exit code 0
*/
```

Yukarıdaki örnekte verilen Öğretmen ve Kitap arasındaki bütünleşme ilişkisi aşağıdaki UML diyagramında gösterilmiştir.

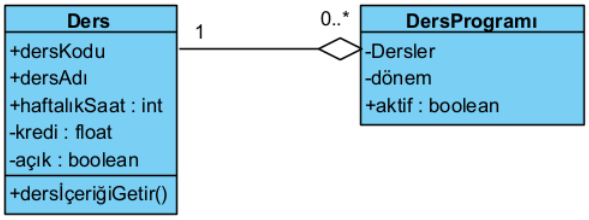


Şekil 28. Bütünleşme UML Diyagramı

Sınıflar arası ilişkilerde birleşme çizgilerinde rakamlar da bulunabilir; Çokluk (multiplicity), ilişkide bir sınıfın örnek (instance) sayısını gösterir. İlişkiyi gösteren çizginin sonunda bir numara olarak gösterilir. Bu numara, mümkün olabilecek örnek sayısıdır. Çokluk, açık ortaklık (public association), gizli ortaklık (private association) ve bütünleşmeye (aggregation) uygulanır;

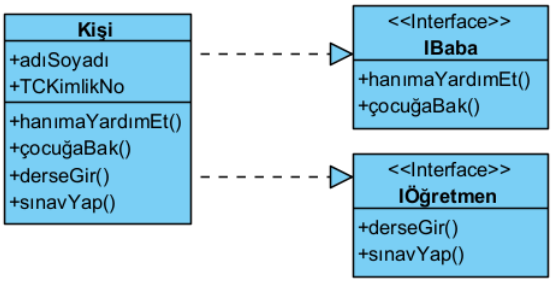
- `0..1` Sıfır ya da 1 örnek.
- `0..` veya `*` Örnek sayısı sınırsızdır.
- `1` Yalnızca 1 örnek

– `1..` En az bir örnek



Şekil 29. Sınıf Diyagramlarında Çokluk Örneği

Ara yüz gerçekleştirme (interface realization) işlemi, ucunda üçgen olan kesikli çizgi ile gösterilir. Üçgenin sivri köşesi gerçekleştirilen ara yüzü gösterir. Çemberler, ara yüzleri (interface) göstermenin bir başka yoludur. Bu durumda daha kısa ve etkili anlatıma sahip diyagramlara sahip oluruz. Bu tip basitleştirilmiş diyagramlar lolipop diyagramı olarak da adlandırılır.



Şekil 30. Sınıf Diyagramlarında Ara yüz gerçekleştirme

Sınıf Tasarlama İlkeleri

Nesne yönelimli programlama (object oriented programming); Kodumuzda değişim yönetimi (change management), gösterici kullanımı yerine referansları kullanarak güvenli kod (safe code) aynı kodları tekrar yazmak (duplicate code) yerine kodların yeniden kullanımı (reusing) sağlayan aşağıdaki özellikleri kullanırız;

– Kod küçüldüğü için kalıtım (inheritance),
– Veri ve kontrol soyutlaması (abstraction) yaparak daha güvenli bileşen tasarladığımız için sarmalama (encapsulation),
– Yazılım mimarisinin daha esnek ve genişleyebilir olması için çok biçimlilik (polymorphism).

Ayrıca yazılım geliştirmede;

– Zayıf bağlaşımın (loosely coupling) sağlanması için yazılımın bir noktasında olabilecek değişikliğin, diğer kısımlarda değişiklik gerektirmeyecek şekilde tasarlanması gerekir. Yani yapılan değişiklik, çorap söküğü gibi yazılımın diğer kısımlarını değiştirmemelidir.
– Aynı çağrışım kümesine ilişkin bütün bileşenler bir arada geliştirilmeli diğer kümelerle karıştırılmamalıdır (seperation of concern). Yani bileşenler arasında yüksek uyum (high cohesion) olmalıdır.

İşte yeniden kullanım, zayıf bağlaşım ve yüksek uyum sağlamak için programcı, nesne yönelimli programlama özellikleriyle birlikte temel ilkelere (principle) uyar.

Programcı kodu tasarlarken, yazarken ve yeniden düzenlerken bu ilkeleri aklında tutarsa; Kod çok daha temiz, genişletilebilir ve test edilebilir olur.

Bunlar ilkelerin İngilizce baş harflerinin alınarak isimlendirilen SOLID ilkeleridir;

Tek Sorumluluk İlkesi (Single Responsibility Principle-SRP):

Sınıflar, tasarlanırken birden fazla işten sorumlu tutulmamalıdır. Mümkün olduğunca bir işle ilgili olarak tasarlanmalıdır. Bir sınıfın tek bir şey yapması gerektiğini ve dolayısıyla değişmesi için tek bir nedeninin olması gerektiğini belirtir.

Açık-Kapalı İlkesi (Open Closed Principle-OCP):

Sınıflar, değişikliklere kapalı (closed for modification) eklentilere açık (open for extension) şekilde tasarlanmalıdır. Bu şekilde değişikliklere açık genişleyebilir modüllere sahip oluruz. Değişikliklere kapalı olmak, mevcut bir sınıfın kodunu değiştirmemek anlamına gelirken, eklentilere açık olma yeni işlevler eklemek anlamına gelir.

Liskov İkame İlkesi (Liskov Substitution Principle-LSP):

Alt sınıflar türediği sınıfların yerine konulabilmelidir. Bu prensipten ilk olarak Barbar Liskov tarafından bahsedilmiştir. Alt sınıflar, türediği sınıf davranışlarıyla kullanılabilmelidir. Burada amaç, alt sınıfların daha çok değişebileceği göz önüne alınarak değişimlerden an az şekilde etkilenmektir.

Türemiş sınıfından bir nesneyi, Taban sınıfından bir nesne bekleyen herhangi bir yönteme parametre olarak geçirdiğimizde yöntemin herhangi bir tuhaf çıktı vermemesi gerektiği anlamına gelir. Bu beklenen davranıştır, çünkü kalıtım kullandığımızda alt sınıfın, üst sınıfın sahip olduğu her şeyi miras aldığını varsayarız. Alt sınıf davranışı genişletir ancak asla daraltmaz. Dolayısıyla bir sınıf bu ilkeye uymadığında, tespit edilmesi zor bazı kötü hatalara yol açar.

Ara Yüzleri Ayırma İlkesi (Interface Segregation Principle-ISP):

Genel amaçlı ara yüzler yerine istemciye bağlı ara yüzler tasarlanmalıdır. Yani roller, birbirinden ayrı olacak şekilde tanımlanmalıdır. Birden fazla rol tek bir rol altında olacak şekilde düşünülmemelidir. Bu nedenle buna rollerin ayrılması prensibi (Role Segregation Principle-RSP) olarak da adlandırılır.

Bağımlılıkları Ters Çevirme İlkesi (Dependency Inversion Principle-DMP):

Sınıflarımızın somut sınıflara ve fonksiyonlara değil, ara yüzlere veya soyut sınıflara bağlı olması gerektiğini belirtir. Nesne yönelimli programlamanın en önemli ilkesidir. Aslında bu ilke ile açık-kapalı ilkesi doğrudan birbiriyle ilişkilidir.

YAPILAR VE BİRLİKLER

Yapı Tanımlaması

Yapısal programlamaya adını veren yapı (struct), türetilmiş (derived) veya kullanıcı tanımlı (user defined) bir veri tipidir. Farklı tiplerdeki elemanları bir arada gruplandıran özel bir veri tipi tanımlamak için `struct` anahtar kelimesini kullanırız.

Yapı bir veya birden fazla ilkel veri tipinin (`char`, `int`, `long`, `float`, `double`, ... ve bu tiplere ait diziler) bir araya gelmesiyle oluşturulan yeni veri tipleridir. Diziler, aynı tipte elemanlardan oluşmasına rağmen, yapılar farklı dipte elemanların bir araya gelmesiyle oluşabilir;

```
struct yapı-kimliği {
    veri-tipi yapı-elemanı-kimliği1;
    veri-tipi yapı-elemanı-kimliği1;
    ...
} [değişken-kimliği1][,değişken-kimliği2];
```

Örnek olarak Öğrenci; adı, soyadı, numarası, yaşı, cinsiyeti gibi farklı öğeler ile bir `ogrenci` yapısı tanımlanabilir;

```
struct ogrenci {
        unsigned yas;
        char cinsiyet;
        float kilo;
        unsigned boy;
    } ogrenci1;
```

Tanımlanan yapı kullanılarak değişkenler aşağıdaki gibi tanımlanabilir;

```
struct yapı-kimliği değişken-kimliği;
```

Yukarıda tanımlanan yapı kimliği kullanılarak birçok değişken kimliklendirilebilir;

```
struct ogrenci ogrenci2,ogrenci3;
```

Yapılar sınıflara benzer aralarındaki fark erişim belirleyicisinin halka açık (`public`) olmasıdır;

```
struct Vector {
    int x;
    int y;
    int z;
};
// Aşağıdakine eşdeğerdir:
class Vector {
public:
    int x;
    int y;
    int z;
};
```

Yapı Elemanlarına Erişim

Tanımlanan yapı değişkenleri üzerinden her bir alamana nokta işleci (dot operator) erişilir. Bu operatör de parantez öncelik işleci ile aynı önceliktedir. Atama (`=`) işleci bir yapıyı (struct) doğrudan kopyalamak için kullanılabilir. Ayrıca, bir yapının üyesinin değerini başka birine atamak için atama işlecini de kullanabiliriz.

```
#include <iostream>
using namespace std;
int main() {
    struct ogrenci {
```

```
        unsigned yas;
        char cinsiyet;
        float kilo;
        unsigned boy;
    } ogrenci1;

    ogrenci1.yas=19;
    ogrenci1.cinsiyet='E';
    ogrenci1.kilo=75.5;
    ogrenci1.boy=180;
    cout << "Öğrenci1'in:" << endl << "yaşı:" << ogrenci1.yas<< ", cinsiyeti: "
         << ogrenci1.cinsiyet << ", kilosu:"<< ogrenci1.kilo
         << ", boyu:" << ogrenci1.boy << endl;

    struct ogrenci ogrenci2={25,'K',55.0,'K'}; //yapı değişkenine ilk değer verme.
    cout << "Öğrenci 2'nin:" << endl << "yaşı:" << ogrenci2.yas<< ", cinsiyeti: "
         << ogrenci2.cinsiyet << ", kilosu:"<< ogrenci2.kilo << ", boyu:"
         << ogrenci2.boy << endl;

    struct ogrenci ogrenci3=ogrenci2;
    cout << "Öğrenci 3'ün:" << endl << "yaşı:" << ogrenci3.yas<< ", cinsiyeti: "
         << ogrenci3.cinsiyet << ", kilosu:"<< ogrenci3.kilo << ", boyu:"
         << ogrenci3.boy << endl;
}
```

Yapı Göstericileri

Yapı göstericileri, tıpkı diğer değişkenlere gösterici tanımladığımız gibi tanımlayabiliriz. Gösterici üzerinden yapı değişkenlerine dolaylı işleç (indirection operator) yani (`->`) ile erişilir. Bu nokta işleci ile aynı önceliklidir.

Yapılar ve göstericileri; veri tabanları, dosya yönetim uygulamaları ve ağaç ve bağlı listeler gibi karmaşık veri yapılarını işlemek gibi farklı uygulamalarda kullanılır.

```
#include <iostream>
using namespace std;
int main() {
    struct ogrenci {
        unsigned yas;
        char cinsiyet;
        float kilo;
        unsigned boy;
    } ogrenci1;

    ogrenci1.yas=19;
    ogrenci1.cinsiyet='E';
    ogrenci1.kilo=75.5;
    ogrenci1.boy=180;

    cout << "Öğrenci1'in:" << endl << "yaşı:" << ogrenci1.yas
         << ", cinsiyeti: " << ogrenci1.cinsiyet
         << ", kilosu:" << ogrenci1.kilo
         << ", boyu:" << ogrenci1.boy << endl;

    struct ogrenci* ogrenciGosterici;
    ogrenciGosterici=&ogrenci1;
    cout << "Öğrenci1'in:" << endl << "yaşı:" << ogrenciGosterici->yas
         << ", cinsiyeti: " << ogrenciGosterici->cinsiyet
         << ", kilosu:" << ogrenciGosterici->kilo
         << ", boyu:" << ogrenciGosterici->boy << endl;
}
```

Takma İsimler

Halihazırda mevcut veri tiplerinin adını yeniden tanımlamak için `typedef` anahtar kelimesi kullanılır. Takma isimler (typedef), yapı (struct) ve daha sonra göreceğimiz birlik (union) gibi veri tiplerinin çokça kullanıldığı kodlarda kodun boyunu oldukça kısaltır ve aşağıdaki şekilde tanımlanır;

```
typedef mevcutisim takmaisim;
```

Aşağıda takma isim verilmiş yeni veri tipleriyle yazılmış bir kod örneği verilmiştir;

```
#include <iostream>
using namespace std;

typedef char karakter;
typedef unsigned int pozitiftamsayi;
typedef float gerceksayi;

struct ogrenci {
    pozitiftamsayi yas;
    karakter cinsiyet;
    gerceksayi kilo;
    pozitiftamsayi boy;
};
typedef struct ogrenci Ogrenci;
int main() {
    Ogrenci ogrenci1;
    ogrenci1.yas=19;
    ogrenci1.cinsiyet='E';
    ogrenci1.kilo=75.5;
    ogrenci1.boy=180;
    cout << "Öğrenci1'in:" << endl << "yaşı:" << ogrenci1.yas
         << ", cinsiyeti: " << ogrenci1.cinsiyet
         << ", kilosu:" << ogrenci1.kilo
         << ", boyu:" << ogrenci1.boy << endl;
}
```

Fonksiyonlara da takma isim verilebilir;

```
typedef int Fonk(int); /* Fonk bir fonksiyona verilen bir takma isimdir;
Fonk, int tipinde parametre alan ve int geri döndüren her fonksiyon olabilir */

int faktoriyel(int n) { //faktoriyel fonksiyonu da Fonk tipindedir.
    return (n==1)? 1: n*faktoriyel(n-1);
}
int onKat(int i) {
    return i*10;
}
int main() {
    Fonk *fn; // fn bir Fonk tipinde olan fonksiyonlara olan göstericidir.
    fn = &faktoriyel; // faktoriyel fonksiyonunu göster
    fn(3); // 3! Hesaplanır
    fn = &onKat; // onKat fonksiyonunu göster
    fn(4); // 4*10 hesaplanır
}
```

Yapı Dizileri

Bir yapı (struct) değişkeni, ilkel tiplerden (`char`, `int`, `float`, ...) tanımlanan bir diziye benzer şekilde bir yapı dizisi tanımlayabiliriz. Ayrıca yapı değişkenini bir fonksiyona parametre olarak gönderebilir ve bir fonksiyondan bir yapı döndürebilirsiniz.

```
#include <iostream>
```

```
using namespace std;
enum cinsiyet {BELIRTILMEMIS,KADIN,ERKEK};
struct ogrenci {
    unsigned yas;
    int cinsiyet;
    float kilo;
    unsigned boy;
};
typedef struct ogrenci Ogrenci;
int main() {
    Ogrenci ogrenciler[30];
    cout << "Yaşı Giriniz:";
    cin >> ogrenciler[0].yas;
    cout << "Cinsiyet Giriniz (0-1-2):";
    cin >> ogrenciler[0].cinsiyet;
    cout << "Kilo Giriniz:";
    cin >> ogrenciler[0].kilo;
    cout << "Boy Giriniz:";
    cin >>ogrenciler[0].boy;

    cout << "Öğrenci:" << endl << "yaşı:" << ogrenciler[0].yas
        << ", cinsiyeti: " << ogrenciler[0].cinsiyet
        << ", kilosu:" << ogrenciler[0].kilo
        << ", boyu:" << ogrenciler[0].boy << endl;
}
```

Parametre Olarak Yapılar

Yapılar değer tipler olduğundan, yapılara ait göstericileri parametre olarak kullanmak, olabilecek değişiklikleri aktarmak için üstünlük sağlar.

```
#include <iostream>
using namespace std;
enum cinsiyet {BELIRTILMEMIS,KADIN,ERKEK};
struct ogrenci {
    unsigned yas;
    int cinsiyet;
    float kilo;
    unsigned boy;
};
typedef struct ogrenci Ogrenci;
void ogreciYaz(Ogrenci);
void ogrenciOku(Ogrenci*);
int main() {
  Ogrenci ogrenciler[30];
  ogrenciOku(&ogrenciler[0]);
  ogreciYaz(ogrenciler[0]);
}
void ogreciYaz(Ogrenci pOgrenci) {
    cout << "Öğrenci:" << endl << "yaşı:" << pOgrenci.yas
         << ", cinsiyeti: " << pOgrenci.cinsiyet
         << ", kilosu:" << pOgrenci.kilo
         << ", boyu:" << pOgrenci.boy << endl;
}
void ogrenciOku(Ogrenci* pOgrenci) {
    cout << "Yaşı Giriniz:";
    cin >> pOgrenci->yas;
    cout << "Cinsiyet Giriniz (0-1-2):";
    cin >> pOgrenci->cinsiyet;
    cout << "Kilo Giriniz:";
    cin >> pOgrenci->kilo;
```

```
    cout << "Boy Giriniz:";
    cin >> pOgrenci->boy;
}
```

Anonim Yapılar

Anonim yapı, kimliği veya `typedef` ile tanımlanmayan bir yapı tanımıdır. Genellikle başka bir yapının içine yerleştirilir. 2011 C sürümü ile kullanılmaya başlanan bu özelliğin aşağıda sıralanan üstünlükleri vardır;

- Esneklik (flexibility): Anonim yapılar, verilerin nasıl temsil edildiği ve erişildiği konusunda esneklik sağlayarak daha dinamik ve çok yönlü veri yapılarına olanak tanır.
- Kolaylık (convenience): Bu özellik, farklı veri tiplerini tutabilen bir değişkenin kompakt bir şekilde temsil edilmesine olanak tanır.
- Başlatma Kolaylığı (easy of initialization): Yapı değişkenine ilişkin ek kimliklendirme yapılmadan ilk değer verilmeleri ve kullanılmaları daha kolay olabilir.

```
#include <iostream>
using namespace std;
enum cinsiyet {BELIRTILMEMIS,KADIN,ERKEK};
struct ogrenci {
    unsigned yas;
    int cinsiyet;
    float kilo;
    unsigned boy;
    struct {
        int gun;
        int ay;
        int yil;
    };
};
typedef struct ogrenci Ogrenci;
void ogreciYaz(Ogrenci);
int main() {
  Ogrenci ogrenci={ 45,KADIN, 65.0, 180, {01, 02, 2000} };
  ogreciYaz(ogrenci);
}
void ogreciYaz(Ogrenci pOgrenci) {
    cout << "Öğrenci:" << endl << "yaşı:" << pOgrenci.yas
         << ", cinsiyeti: " << pOgrenci.cinsiyet
         << ", kilosu:" << pOgrenci.kilo
         << ", boyu:" << pOgrenci.boy
         << ", tarihi:" << pOgrenci.gun << "-" << pOgrenci.ay << "-" << pOgrenci.yil
         << endl;
}
```

Öz Referanslı Yapılar

Kendi kendine yani öz referanslı yapı (self-referetial struct), öğelerinden bir veya daha fazlası kendi türündeki göstericilerden oluşan bir yapıdır. Kendi kendine referanslı kullanıcı tanımlı yapılar, bağlantılı listeler ve ağaçlar gibi karmaşık ve dinamik veri yapıları (dynamic data structure) için yaygın olarak kullanılırlar. Dinamik veri yapıları yapısal programlama sürecinde çokça geliştirilen ve artık standart hale gelen koleksiyonlardır. C++ dilinde koleksiyonlar, artık hazır bir kütüphanedir ve *Konteyner Şablonları* başlığında incelenecektir.

```
#include <iostream>
using namespace std;
enum cinsiyet {BELIRTILMEMIS,KADIN,ERKEK};
struct ogrenci {
    unsigned yas;
```

```
    int cinsiyet;
    float kilo;
    unsigned boy;
    struct ogrenci* sonrakiOgrenci;
};
typedef struct ogrenci Ogrenci;
void ogrecileriYaz(Ogrenci*);
int main() {
  Ogrenci ogrenci1={ 45,KADIN, 65.0, 165, nullptr };
  Ogrenci ogrenci2={ 55,ERKEK, 85.5, 170, nullptr };
  Ogrenci ogrenci3={ 25,ERKEK, 65.5, 155, nullptr };
  ogrenci1.sonrakiOgrenci=&ogrenci2;
  ogrenci2.sonrakiOgrenci=&ogrenci3;
  ogrecileriYaz(&ogrenci1);
}
void ogrecileriYaz(Ogrenci* pOgrenci) {
    while(pOgrenci!=nullptr) {
        cout << "Öğrenci:" << counter++ << endl << "yaşı:" << pOgrenci->yas
             << ", cinsiyeti: " << pOgrenci->cinsiyet
             << ", kilosu:" << pOgrenci->kilo
             << ", boyu:" << pOgrenci->boy << endl;
        pOgrenci=pOgrenci->sonrakiOgrenci;
    }
}
/* Program Çıktısı:
Öğrenci:0
yaşı:45, cinsiyeti: 1, kilosu:65, boyu:165
Öğrenci:1
yaşı:55, cinsiyeti: 2, kilosu:85.5, boyu:170
Öğrenci:2
yaşı:25, cinsiyeti: 2, kilosu:65.5, boyu:155

...Program finished with exit code 0
*/
```

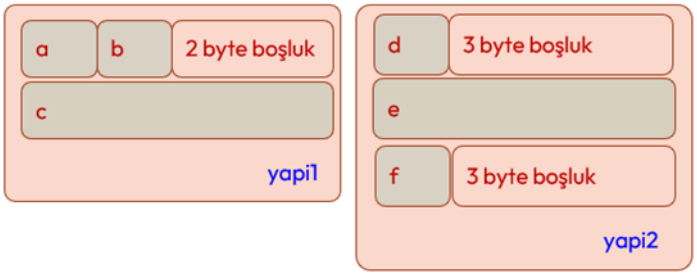
Yapı Dolgusu

C dilinde yapı dolgusu (structure padding), işlemci (CPU) mimarisi ile belirlenir. Dolgu (padding) işlemi ile üyelerinin bellekte doğal olarak hizalanması için belirli sayıda boş bayt eklenir. Bunun nedeni 32 veya 64 bitlik bir bilgisayarda işlemcinin tek seferde bellekten 4 bayt okumasından kaynaklanmaktadır.

```
#include <iostream>
using namespace std;
struct yapi1 {
   char a;
   char b;
   int c;
};
struct yapi2 {
   char d;
   int e;
   char f;
};
int main() {
   cout << "char bellek miktarı:" << sizeof(char) << endl;
        // char bellek miktarı: 1
   cout << "int  bellek miktarı:" << sizeof(int) << endl;
        // int  bellek miktarı: 4
   cout << "------------------------" << endl;
   cout << "Her iki yapı için olmasi gereken bellek miktarı:"
```

```
        << 2*sizeof(char)+sizeof(int) << endl;
        // Her iki yapı için olmasi gereken bellek miktarı: 6
   cout << "yapi1 için bellekte ayrılan miktar:" << sizeof(struct yapi1) << endl;
        // yapi1 için bellekte ayrılan miktar: 8
   cout << "yapi2 için bellekte ayrılan miktar:" << sizeof(struct yapi2) << endl;
        // yapi2 için bellekte ayrılan miktar: 12
}
```

Yukarıda verilen örnekte aynı bellek miktarına sahip elemanlar için yapının bütününe bakıldığında farklı miktarda bellek ayrılabileceği aşağıdaki şekilden anlaşılmaktadır;



Şekil 31. Yapı Dolgusu

Dolgu (padding) işlemi ile üyelerinin bellekte doğal olarak hizalanması için belirli sayıda boş bayt eklenir. Bunu tüm yapılar için engellemenin yolu; `#pragma pack(1)` ön işlemci yönergesini kaynak koda eklemektir.

```
#include <iostream>
using namespace std;
#pragma pack(1)
struct yapi1 {
   char a;
   char b;
   int c;
};
struct yapi2 {
   char d;
   int e;
   char f;
};
int main() {
   cout << "char bellek miktarı:" << sizeof(char) << endl;
         // char bellek miktarı: 1
   cout << "int  bellek miktarı:" << sizeof(int) << endl;
         // int  bellek miktarı: 4
   cout << "------------------------" << endl;
   cout << "Her iki yapı için olmasi gereken bellek miktarı:"
        << 2*sizeof(char)+sizeof(int) << endl;
         // Her iki yapı için olmasi gereken bellek miktarı: 6
   cout << "yapi1 için bellekte ayrılan miktar:" << sizeof(struct yapi1) << endl;
         // yapi1 için bellekte ayrılan miktar: 6
   cout << "yapi2 için bellekte ayrılan miktar:" << sizeof(struct yapi2) << endl;
         // yapi2 için bellekte ayrılan miktar: 6
}
```

Eğer yalnızca belirlenen yapı için bunun yapılması isteniyorsa yapı tanımına `__attribute__((packed))` özelliği eklenir;

```
#include <iostream>
using namespace std;
struct yapi1 {
   char a;
   char b;
```

```
   int c;
};
struct __attribute__((packed))  yapi2 {
   char d;
   int e;
   char f;
};
int main() {
   cout << "char bellek miktarı:" << sizeof(char) << endl;
         // char bellek miktarı: 1
   cout << "int  bellek miktarı:" << sizeof(int) << endl;
         // int  bellek miktarı: 4
   cout << "------------------------" << endl;
   cout << "Her iki yapı için olmasi gereken bellek miktarı:"
        << 2*sizeof(char)+sizeof(int) << endl;
         // Her iki yapı için olmasi gereken bellek miktarı: 6
   cout << "yapi1 için bellekte ayrılan miktar:" << sizeof(struct yapi1) << endl;
         // yapi1 için bellekte ayrılan miktar: 8
   cout << "yapi2 için bellekte ayrılan miktar:" << sizeof(struct yapi2) << endl;
         // yapi2 için bellekte ayrılan miktar: 6
}
```

Birlikler

Birlikler (union), Pascal dilindeki record case talimatına (statement) benzer. Birlik (union), yapı (struct) gibi tanımlanır.

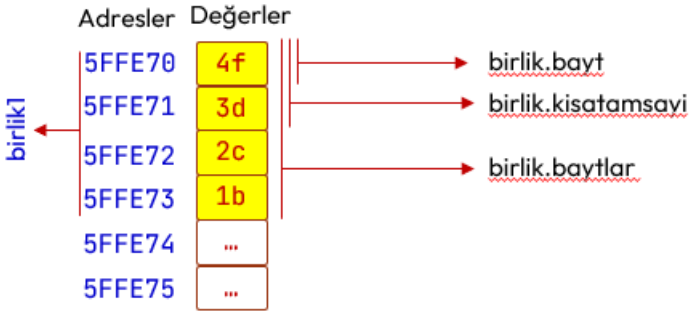
```
union yapı-kimliği {
    veri-tipi yapı-elemanı-kimliği1;
    veri-tipi yapı-elemanı-kimliği1;
    ...
} [değişken-kimliği1][,değişken-kimliği2];
```

Aralarındaki fark yapı elemanlarının her birine ayrı bellek bölgesi ayrılırken, birlik üyelerinin her biri aynı bellek bölgesini paylaşırlar. Birliğin belek boyutu, elemanlarından en fazla bellek kaplayanı kadardır. Aşağıda üyelerinin aynı bellek bölgesini paylaştığı görülmektedir.

```
#include <iostream>
#include <iomanip> //setbase()
using namespace std;
union tamsayiBirligi {
    char baytlar[4];
    int  tamsayi;
    char bayt;
    short int kisatamsayi;
} birlik1,birlik2;
int main() {
    union tamsayiBirligi birlik3,birlik4;
    birlik1.tamsayi=0x1B2C3D4F; //16lık sayılarda her çift rakam bir bayt olur
    cout << "sizeof tamsayiBirligi.baytlar:" << sizeof birlik1.baytlar << endl;//4
    cout << "sizeof tamsayiBirligi.bayt:" << sizeof birlik1.bayt << endl;//1
    cout << "sizeof tamsayiBirligi.tamsayi:" << sizeof birlik1.tamsayi
         << endl;//4
    cout << "sizeof tamsayiBirligi.kisatamsayi:" << sizeof birlik1.kisatamsayi
         << endl;//2
    cout << "---------------------------------------" << endl;
    cout << "sizeof tamsayiBirligi:" << sizeof birlik1 << endl;//4
    cout << "---------------------------------------" << endl;
    cout << "birlik1.tamsayi:" << setbase(16) << birlik1.tamsayi
         << endl; //1b2c3d4f
    cout << "birlik1.baytlar:" << setbase(16)
         << int(birlik1.baytlar[0]) << "-"
```

```
        << int(birlik1.baytlar[1]) << "-"
        << int(birlik1.baytlar[2]) << "-"
        << int(birlik1.baytlar[3]) << endl; //4f-3d-2c-1b
    cout << "birlik1.bayt:" << setbase(16) << int(birlik1.bayt) << endl;  //4f
    cout << "birlik1.kisatamsayi:" << setbase(16) << birlik1.kisatamsayi
        << endl; //3d4f
    cout << "-------------------------------------" << endl;
    birlik1.bayt=0x00;
    cout << "birlik1.tamsayi:" << setbase(16) << birlik1.tamsayi; //1b2c3d00
}
```

Şekil olarak da birlik1 değişkeninin bellek yerleşimi aşağıda verilmiştir;



Şekil 32. birlik1 Değişkeninin Bellek Yerleşimi

İSTİSNA YÖNETİMİ

Yapısal Programlamada Hata Yönetimi

Yapısal programlamada sorun olan, hata ayıklama kodu ile işlevsel kodun iç içe olması durumudur. Nesne yönelimli programlamanın getirdiği yenilikle bu sorun tarihe karışmıştır. Yapısal programlamada hatalı bir duruma yol açmamak ya da hatalı bir durumla karşılaşıldığında programdan çıkılır. Aşağıda örneği verilmiştir.

```
#include <stdio.h>
int faktoriyel(int sayi) {
  if (sayi<0) return -1;
  if(sayi <= 1) return 1;
  return sayi * faktoriyel(sayi - 1);
}
int main() {
  int okunan,deger,hata;
  do {
    printf("Bir Sayı Giriniz:");
    scanf("%d",&okunan);
    deger=faktoriyel(okunan);
    printf("%d nin faktöriyeli: %d\n", okunan, deger);
    if (deger==-1) {
       hata=100;
       break; // exit(100);
    }

  } while (okunan!=0);
  return hata;
}
/* Programın Çıktısı:
Bir Sayı Giriniz:-1
-1 nin faktöriyeli: -1

...Program finished with exit code 100
*/
```

İstisna Yönetimi

Nesne Yönelimli Programlamada ise istisna (exception), bir programın yürütülmesi sırasında ortaya çıkan beklenmeyen bir sorundur. İstisna, programın çalışması sırasında (run time) oluşur. Beklenmeyen durumlarda imal edilen istisna nesneleri (exception object) vardır. İstisna iki türlü ortaya çıkar;

- Eşzamanlı (synchronous): Giriş verilerindeki bir hata nedeniyle bir şeylerin ters gitmesi veya programın çalıştığı mevcut veri türünü işleme durumunda oluşan istisnalar (örneğin bir sayıyı sıfıra bölmek).
- Eşzamanlı olmayan (asynchronous): Disk arızası, klavye kesintileri vb. gibi yazılan programın kontrolü dışındaki istisnalar.

İstisnai durumları yönetmek için `throw`, `try` ve `catch` anahtar kelimeleri kullanılır;

- `try` talimatı: İçerisinde istisnai bir duruma düşüleceğimizi belirten bir kod bloğunu temsil eder. Bu bloğu bir veya daha fazla `catch` bloğu takip eder.
- `catch` talimatı: `try` bloğunda bir istisnai duruma düşüldüğünde yürütülecek bir kod bloğunu temsil eder. İstisnayı işlemek için kullanılan kod, `catch` bloğunun içine yazılır.

- Bir istisnai duruma `throw` anahtar sözcüğü kullanılarak da düşülebilir. Bir program bir `throw` ifadesiyle karşılaştığında hemen içinde bulunduğu `try` bloğu dışına çıkılır ve ilgili istisnayı işlemek için eşleşen bir `catch` bloğu bulmaya başlar.
- Bir `try` bloğu içinde birden fazla `throw` ifadesi bulunabilir.

```
try {
   /*
    ...
    İstisnai Durumun Ortaya Çıkabileceği Kod...
    ...
   */
   throw SomeExceptionType("İstisnai Durum Açıklaması");
   /*
    throw an exception: istisnai bir duruma düşme
   */
 } catch( ExceptionName e1 )  {
   /*
    catch bloğu try bloğundaki istisnayı yakalamak ve gerekli işlemi yapmak için kullanılır.
   */
 }
```

İstisna işlemeye niçin ihtiyaç duyarız?

- Hata işleme kodunun işlevsel koddan ayrılması için.
- Yöntemler yalnızca seçtikleri istisnaları işleyebilir. Bir yöntemde birden çok istisnai duruma düşülebilir. Ancak bunlardan bazılarını `catch` bloğunda işlemeyi seçebilir. Geri kalanlarını yöntemi çağıran yere bırakabilir.
- Hata Türlerinin Gruplanması: C++ dilinde hem temel tipler (int, float, string ...) hem de nesneler istisnai durum olarak belirlenebilir. Böylece bunların hiyerarşik olarak gruplaması yapılabilir.

Aşağıda sıfıra bölme hatasını işleyen bir program örneği verilmiştir;

```
#include <iostream>
#include <stdexcept>
using namespace std;
int main() {
   cout << "Burası her zaman icra edilir." << endl;
   try { //try BLOĞU: İstisna olabilecek bir bok bloğu
      int bolunen = 10;
      int bolen = 0;
      int sonuc;
      if (bolen == 0) {
         throw runtime_error("Sıfıra Bölmeye İzin Verilmez!");
         //Buradan sonra yazılacak kodlar icra edilmez.
         cout << "Burası hiçbir zaman icra edilmez." << endl;
      }
      sonuc = bolunen / bolen;
      cout << "Bölme Sonucu: " << sonuc << endl;
   } catch (const exception& e) {
      cout << "İstisna: " << e.what() << endl;
   }
   cout << "Burası da her zaman icra edilir." << endl;
   return 0;
}
/* Programın Çıktısı:
Burası her zaman icra edilir.
İstisna: Sıfıra Bölmeye İzin Verilmez!
Burası da her zaman icra edilir.

...Program finished with exit code 0
*/
```

Yandaki örnekte `try` bloğunda izin verilmeyen bölme işleminde istisnai bir durum fark ediliyor ve istisna nesnesi imal ediliyor ve `throw` ile (istisna havuzuna) fırlatılıyor (throw an exception). `catch` bloğunda ise (havuzdan) istisna nesnesi yakalanarak (catch an exception) işleniyor. Örnekte standart `runtime_error` istisna nesnesi kullanılmıştır. Bu nesneler ön tanımlı nesnelerdir. `stdexcept` başlığında aşağıdaki istisnalar ön tanımlıdır; `runtime-error`, `logic_error`, `invalid_argument`, `range_error`, `out_of_range`, `overflow_error`, `underflow_error`.

Bir `try` bloğunda birden fazla istisna nesnesi (istisna havuzuna) fırlatılabilir; Aşağıdaki örnekte `try` bloğunda birden fazla istisnai nesnesi fırlatılmıştır.

```
#include <iostream>
using namespace std;
int main()
{
   try {
      throw 10;
      throw 'A';
      throw "Hata";
   }
   catch (char* excp) {
      cout << "İstisna: " << excp;
   }
   catch (...) { //default exception üç nokta ile belirtilir.
      cout << "Diğer Catchlerde işlenmeyen istisnalar burada işlenir.\n";
   }
   return 0;
}
/* Programın Çıktısı:
Diğer Catchlerde işlenmeyen istisnalar burada işlenir.

...Program finished with exit code
*/
```

Programda ilk karşılaşılan `throw` ile (istisna havuzuna) fırlatılan nesne tamsayı nesnesidir. Bu durumda `catch` bloklarına sırasıyla bakılır ilk (havuzdan) yakalanacak olan dizgi (string) nesneleridir. Havuzda dizgi nesnesi olmadığından bu `catch` bloğu icra edilmez. Bir sonraki `catch` bloğuna bakılır ve böyle devam edilir. Eğer havuzdan yakalanacak istisna nesnesi önceki `catch` bloklarında tanımlı değilse üç nokta ile belirtilen `catch` bloğu icra edilir.

Aşağıda Kişi sınıfta yaş özelliğine ilişkin istisna içeren bir kod örneği verilmiştir;

```
#include <iostream>
#include <stdexcept>
using namespace std;
class Kisi {
private:
    int yas;
public:
    Kisi() {
        yas=15;
    }
    int getYas() {
        return yas;
    }
    void setYas(int pYas) {
      if (pYas < 0)
        throw range_error("Yaş değeri sıfırdan küçük olamaz!");
      else if (pYas > 120)
        throw range_error("Çok Büyük Yaş Değeri: >120!");
      yas = pYas;
    }
```

```
};
int main() {
    Kisi ali;
    try {
        ali.setYas(12);
        cout <<ali.getYas()<<endl;
        ali.setYas(-1); // Bu talimatta istisna nesnesi oluşur.
        //Buradan sonrası icra edilmez!
        cout <<ali.getYas()<<endl;
        ali.setYas(130);
        cout <<ali.getYas()<<endl;
    }
    catch (const exception& istisna) {
        cout << "İstisnai Bir Durum Yakalandı!" << endl;
        cout << "Durum Açıklaması: " << istisna.what()  << endl;
    }
}
/* Programın Çıktısı:
12
İstisnai Bir Durum Yakalandı!
Durum Açıklaması: Geçersiz Yaş Değeri: <0!

...Program finished with exit code 0
*/
```

Yukarıdaki kod örneğindeki gibi her zaman standart istisna nesneleri kullanmak zorunda değiliz. Bu durumda `exception` başlığını projemize dahil ederek kullanarak kendi istisna nesnemizi de oluşturabiliriz.

```
#include <iostream>
#include <exception>
using namespace std;

struct yasMaxException: public exception {
   const char* what () const throw () {
      return "Çok Büyük Yaş Değeri: >120!";
   }
};
struct yasMinException: public exception {
   const char* what () const throw () {
      return "Geçersiz Yaş Değeri: <0!";
   }
};

class Kisi {
private:
    int yas;
public:
    Kisi() {
        yas=15;
    }
    int getYas() {
        return yas;
    }
    void setYas(int pYas) {
      if (pYas < 0)
        throw yasMinException();
      else if (pYas > 120)
        throw yasMaxException();
      yas = pYas;
    }
```

```
};
int main() {
    Kisi ali;
    try {
        ali.setYas(12);
        cout <<ali.getYas()<<endl;
        ali.setYas(-1); // Bu talimatta istisna nesnesi oluşur.
        //Buradan sonrası icra edilmez!
        cout <<ali.getYas()<<endl;
        ali.setYas(130);
        cout <<ali.getYas()<<endl;
    }
    catch (const exception& istisna) {
        cout << "İstisnai Bir Durum Yakalandı!" << endl;
        cout << "Durum Açıklaması: " << istisna.what()  << endl;
    }
}
/* Programın Çıktısı:
12
İstisnai Bir Durum Yakalandı!
Durum Açıklaması: Geçersiz Yaş Değeri: <0!

...Program finished with exit code 0
*/
```

noexcept İşleci

İşlenen değerlendirmesinin bir istisna nesnesi imal edip etmeyeceğini belirleyen tekli işleç `noexcept` işlecidir. Çağırılan yöntem ya da fonksiyonların gövdelerinin incelenmediğini, dolayısıyla `noexcept` işlecinin yanlış sonuçlar verebileceğini unutulmamalıdır;

```
#include <iostream>
#include <stdexcept>
using namespace std;
void fonksiyon() { throw std::runtime_error("oops"); }
void bosfonksiyon() {}
struct yapi {};
int main() {
    cout << noexcept(fonksiyon()) << endl; // 0
    cout << noexcept(bosfonksiyon()) << endl; // 0
    cout << noexcept(1 + 1) << endl; // 1
    cout << noexcept(yapi()) << endl; // 1
}
```

Bu işleç, bir fonksiyon bildirilirken, fonksiyonun gövdesinde bir istisna nesnesi imal edip etmeyeceğini belirtmek için de kullanılır;

```
void f1() { throw std::runtime_error("oops"); }
void f2() noexcept(false) { throw std::runtime_error("oops"); }
void f3() {}
void f4() noexcept {}
void f5() noexcept(true) {}
void f6() noexcept {
    try {
        f1();
    } catch (const std::runtime_error&) {}
}
```

Bu örnekte, `f4`, `f5` ve `f6` fonksiyonlarının istisna imal etmeyeceğini beyan ettik. (Bir istisna `f6` fonksiyonunun yürütülmesi sırasında atılabilse de yakalanır ve fonksiyondan dışarı yayılmasına izin verilmez.) `f2` fonksiyonunun bir istisnayı imal edebileceğini beyan ettik. `f1` ve `f3` fonksiyonlarının

istisnaları imal edeceğini dolaylı olarak beyan ettik. `noexcept` belirteci atlandığında, `noexcept(false)` ile eşdeğerdir.

İSİM UZAYLARI VE NUMARALANDIRMA

İsim Uzayları

İsim uzayları (namespace), değişkenleri, metotları ve sınıfları bir isim altında toplayabileceğimiz bir tanımlamadır. Birden fazla kütüphane kullanırken isim çakışmalarını önlemek için kullanılır. Böylece aynı kimlikli sınıf, değişken ve fonksiyon birden fazla isim uzayında bulunabilir.

Örneğin, `fonksiyon()` kimlikli bir fonksiyonu olan bir kod yazıyor olabilirsiniz ve aynı fonksiyon aynı kimlikle başka bir başlıkta da mevcut olabilir. Bu durumda derleyicinin, kodunuz içinde hangi `fonksiyon()` fonksiyona atıfta bulunduğunuzu bilmesinin bir yolu yoktur. İsim uzayları, bu zorluğun üstesinden gelmek için tasarlanmıştır ve farklı kütüphanelerde bulunan aynı kimliğe sahip benzer fonksiyonları, sınıfları, değişkenleri vb. ayırt etmek için ek bilgi olarak kullanılır. İsim uzaylarına en iyi örnek, C++ standart kütüphanesidir (`std`). Bu nedenle, bir C++ programı yazarken genellikle `namespace std` kullanarak yönergeyi ekleriz;

İsim Uzayı Tanımlama

Bir isim uzayının tanımı, aşağıdaki gibi `namespace` anahtar sözcüğüyle tanımlanır;

```
namespace  isim-uzayı-kimliği
{
    // kod;
    // -değişken tanımlama (int a,b;)
    // -fonksiyon tanımlama (void topla();)
    // -sınıf tanımlama (class Kisi{ /* ... ?/ };)
}
```

Burada isim uzayına ilişkin bloğun noktalı virgül ile bitmediği gözden kaçırılmamalıdır. Çünkü isim uzayına ait bloğun görevi tanımlamaları gruplamaktır. İsin uzayının içindeki bir öğeye erişmek için kapsam çözümleme işleci (scope resolution operator) kullanılır. Ayrıca `using namespace` yönergesini (directive) kullanarak isim uzaylarını ile kapsam çözümleme işlecinin önek olarak eklenmesini de önleyebilirsiniz.

```
#include <iostream>
using namespace std; // std:: kullanmamak için

namespace birinci // birinci kimlikli isim uzayı
{
    int tamsayi=100; // Birinci isim uzayındaki tamsayi değişkeni
    void fonksiyon() {
        cout << "Birinci isim uzayındaki fonksiyon." << endl;
    }
}

namespace ikinci // ikinci kimlikli isim uzayı
{
    int tamsayi=200; // ikinci isim uzayındaki tamsayi değişkeni
    void fonksiyon() {
        cout << "İkinci isim uzayındaki fonksiyon." << endl;
    }
}

using namespace birinci;  //birinci:: kullanmamak için
int main () {
  fonksiyon(); // birinci isim uzayındaki fonksiyon çağrılır.
  cout << tamsayi << endl;

  ikinci::fonksiyon();
```

```
  cout << ikinci::tamsayi << endl;
}
/* Programın Çıktısı:
Birinci isim uzayındaki fonksiyon.
100
İkinci isim uzayındaki fonksiyon.
200

...Program finished with exit code 0
*/
```

İsim uzayları, aşağıdaki şekilde iç içe (nested) de yerleştirilebilir;

```
#include <iostream>
using namespace std; // std:: kullanmamak için

namespace birinci // birinci kimlikli isim uzayı
{
    int tamsayi=100; // Birinci isim uzayındaki tamsayi değişkeni
    void fonksiyon() {
        cout << "Birinci isim uzayındaki fonksiyon." << endl;
    }
    namespace ikinci // ikinci kimlikli isim uzayı
    {
        int tamsayi=200; // ikinci isim uzayındaki tamsayi değişkeni
        void fonksiyon() {
            cout << "İkinci isim uzayındaki fonksiyon." << endl;
        }
    }
}

using namespace birinci::ikinci;
int main () {
  birinci::fonksiyon(); // birinci isim uzayındaki fonksiyon çağrılır.
  cout << birinci::tamsayi << endl;

  fonksiyon(); // ikinci isim uzayındaki fonksiyon çağrılır.
  cout << tamsayi << endl;
}
/* Program çalıştığında:
Birinci isim uzayındaki fonksiyon.
100
İkinci isim uzayındaki fonksiyon.
200

...Program finished with exit code 0
*/
```

Bağımlı Argüman Arama

Açık bir ad alanı niteleyicisi olmadan bir işlevi çağırırken, derleyici, o işlevin parametre türlerinden biri de o ad alanındaysa, bir ad alanı içindeki bir işlevi çağırmayı seçebilir. Buna bağımlı argüman arama (argument dependent lookup- ADL) denir;

```
namespace Test
{
   int func(int i);
   class AClass {...};
   int func2(const AClass &data);
}
func(5); //Hata alınır. Hangi İsim Uzayından olduğu belli değil.
Test::AClass data;
```

```
Func2(data); //Hata alınmaz. Kullandığı argümana göre isim uzayı belli.
```

Bu durumda isim uzaylarının öğelerin başında kullanılması yerinde olacaktır.

İsim Uzaylarını Genişletme

Ad alanlarının kullanışlı bir özelliği de onları genişletebilmenizdir ya da yeni üyeler ekleyebilmenizdir;

```
namespace Test
{
    void fonk1() {}
}
//arada başka kodlar yer alabilir
namespace Test
{
    void fonk2() {}
}
```

Using Anahtar Kelimesi

1. `namespace` anahtar sözcüğüyle birleştirildiğinde bir `using` yönergesi yazarsınız;

```
namespace Test
{
    void func() {}
    void func2() {}
}
//...
Test::func2(); //kapsam çözümleme işleci kullanarak istediğimiz üyeyi kullanabiliriz
using namespace Test; //Test isim uzayını projemize dahil ediyoruz.
//Artık kapsam çözümleme işlecini kullanmamıza gerek kalmaz.
func();
func2();
```

2. Tüm isim uzayı yerine, isim uzayındaki seçili üyeleri projeye dahil etmek mümkündür;

```
using Test::func2;
func2(); //Başarılı bir şekilde projeye dahil edilmiştir.
func(); // Bu fonksiyon projeye dahil edilmemiştir.
```

3. Özellikle başlık (header) dosyalarında `using namespace` kullanmak çoğu durumda kötü görülür. Bu yapılırsa, isim uzayı dahil edilmiş başlığı içeren her dosya projeye aktarılır. Bu da çatışmalara yol açabilir. Örneğin; `AAA.h` Dosyamız;

```
namespace AAA
{
    class C;
}
```

   `BBB.h` Dosyamız;

```
namespace BBB
{
    class C;
}
```

   `CCC.h` Dosyamız;

```
using namespace AAA;
```

   `main.c` Dosyamız;

```
#include "BBB.h"
#include "CCC.h"
using namespace AAA;
C c; // Hata: BBB::C mi? AAA::C mi?
```

Numaralandırma Sınıfı

C++ dilinde kapsamlı numaralandırma (scopped enumaration) olarak da adlandırılan numaralandırma sınıfı (enumaration class), bir grup tamsayılardan oluşan (integral) değişmezden (literal) oluşan numaralandırılmış kullanıcı tanımlı bir veri tipidir. Bir bütünün parçalarını tamsayılar olarak ifade eden değişmezlere kullanıcı tanımlı adlar atamak istediğinizde kullanışlıdır.

```
enum class numaralandırmakimliği {
   adlandırılmış-tamsayı1 [=DEĞER1],
   adlandırılmış-tamsayı2 [=DEĞER2],
   ...
};
```

Sözde kodda `DEĞER1`, `DEĞER2` olarak belirtilenler aslında tamsayılara verilen isimlerdir. Yani tamsayı değişmezlerdir (integer literal). Bu değişmezler büyük harfle yazılırlar ve belirtilmedikçe ilkinin değeri `0`, ikincisinin değeri `1`, ... şeklinde ilerler. Örnek;

```
enum class Durum {
    OK = 0,     // 0
    Hata = 1,   // 1
    Uyarı = 2   // 2
};
```

Bu numaralandırma sınıfı öğeleri bir kod içerisinde yine kapsam çözümleme işleci (scope resolution operator) ile erişilir. İstenirse tamsayılardan oluşan numaralandırma başka bir tamsayı veri tipinde (`char`, `unsigned`, `short`, ...) de oluşturulabilir. Bu durumda numaralandırma sınıfı aşağıdaki şekilde tanımlanır.

```
enum class numaralandırmakimliği: farklı-bir-tamsayı-veritipi {
   adlandırılmış-tamsayı1 [=DEĞER1],
   adlandırılmış-tamsayı2 [=DEĞER2],
   ...
};
```

Örnek;

```
    enum class cinsiyetKodu:char {
        ERKEK = 'E',
        KADIN = 'K',
        BELIRSIZ = 'B'
    };
```

Farklı bir tiple numaralandırma sınıfı oluşturulmuş ise `static_cast` ile tip dönüşümü yapılmalıdır. Aşağıda buna ilişkin bir örnek verilmiştir.

```
#include <iostream>
using namespace std;
int main() {

    enum class cinsiyetKodu:char {
        ERKEK = 'E',
        KADIN = 'K',
        BELIRSIZ = 'B'
    };

    struct ogrenci {
        unsigned yas;
        char cinsiyet;
        float kilo;
        unsigned boy;
    } ogrenci1;

    ogrenci1.yas=19;
```

```
    ogrenci1.cinsiyet=static_cast<char>(cinsiyetKodu::ERKEK);
    ogrenci1.kilo=75.5;
    ogrenci1.boy=180;

    cout << "Öğrenci1'in:" << endl << "yaşı:" << ogrenci1.yas
         << ", cinsiyeti: " <<ogrenci1.cinsiyet
         << ", kilosu:" << ogrenci1.kilo
         << ", boyu:" << ogrenci1.boy << endl;

    struct ogrenci ogrenci2={25,static_cast<char>(cinsiyetKodu::KADIN),55.0,165};
    cout << "Öğrenci 2'nin:" << endl << "yaşı:" << ogrenci2.yas
         << ", cinsiyeti: " << ogrenci2.cinsiyet
         << ", kilosu:" << ogrenci2.kilo
         << ", boyu:" << ogrenci2.boy << endl;
}
/*Program Çıktısı:
Öğrenci1'in:
yaşı:19, cinsiyeti: E, kilosu:75.5, boyu:180
Öğrenci 2'nin:
yaşı:25, cinsiyeti: K, kilosu:55, boyu:165

...Program finished with exit code 0
*/
```

TİP DÖNÜŞÜMLERİ

Tip Dönüşümü Nedir?

Tip dönüşümü, bir veri tipini anlamını kaybetmeyecek şekilde başka uyumlu tip dönüştürmek anlamına gelir.

```
#include <iostream>
using namespace std;

int main() {
    int i = 10;
    char c = 'A';
    cout << (int)c << endl; // c karakterinin tamsayı kodu ekrana yazıldı: 65

    int sum = i + c; /* c karakteri char veri tipinden int veritipine
                        dönüştürüldü ve i değişkeni ile toplandı ardından
                        atama işleci ile sum değişkenine atandı */
    cout << sum; //75
}
```

Üstü Kapalı Tip Dönüşümleri

C++ dilinde üstü kapalı veri tipi dönüşümü (implicit type casting), zorlama (coercion) ya da standart tip dönüşümü (standart type casting) olarak bilinir. Otomatik tip dönüşümü olarak da adlandırılan ve programcının açık talimatlar vermesine gerek kalmadan derleyicinin bir veri tipini otomatik olarak başka bir uyumlu veri tipine dönüştürmesi işlemidir. Şu durumlarda gerçekleşir;

- Dönüşüm farklı veri tiplerinin değerleri üzerinde gerçekleştirilir.
- Farklı bir veri tipi bekleyen bir fonksiyona bir argüman geçirmeniz halinde gerçekleşir.
- Bir veri tipinin değerini başka bir veri tipinin değişkenine atama durumunda gerçekleşir.

Otomatik tip dönüşümde bellekte az yer kaplayan veri tipinden tanımlanmış değişkenler, bellekte çok yer kaplayan değişkenlere atanmaya (widening casting) çalışıldığında otomatik olarak çok yer kaplayanın veri tipine dönüştürülür:

```
bool→ char→ short int→ int→ unsigned int→ long→ unsigned→ long long→ float→ double→ long
double
```

Örnek;

```
//...
char c;   //1 bayt
int i;    //4 bayt
long l;   //8 bayt
float f;  //4 bayt
double d; //8 bayt
c=65;
i=c; //int veri tipine atanmadan önce; char, int veritipine dönüştürülür.
l=i; //long veri tipine atanmadan önce; int, long veritipine dönüştürülür.
f=12.50;
d=f; //double veri tipine atanmadan önce; float, double veritipine dönüştürülür.
//...
```

Ayrıca bellekte çok yer kaplayan veri tipinden tanımlanmış değişkenler, bellekte az yer kaplayan değişkenlere atanmaya (narrowing casting) çalışıldığında, tip dönüşümü otomatik olarak yapılır ancak veri kaybı söz konusu olur.

```
//...
char c;
```

```
int i;
float f;
i=4095; //i=0x00000FFF -> 0x00,0x00,0x0F,0xFF olarak 4 bayt
c=i;    /* c=0xFF -> c değişkenine int veritipinden char veritipine en
           anlamsız baytı atanır.*/

f=12.50;
/*
Kayan noktalı bir sayıdan tamsayıya atama yapılıyor ise tam kısmı tamsayıya çevrilir ve
sonrasında atama yapılır.
*/
i=f;    //i=12
//...
```

Bilinçli Tip Dönüşümü

Bazen belli işlemleri daha kontrollü yapmak amacıyla programcı yapılan işlemde işlenen veri tipini bir başka ver tipine kasıtlı olarak değiştirir. Bu tür dönüşümüne bilinçli tür dönüşümü (explicit type casting) denir ve tekli tip dönüştürme işleci (unary cast operator) ile yapılır.

Diğer tekli işleçler gibi ifadenin (expression) önünde kullanılır ve parantez içerisinde dönüştürülmek istenen veri tipi yazılarak `int j=(int) 12.8/4;` şeklinde kodlanır. Bu işleç, diğer `+`, `-`, `!`, `--` (pre-decrement), `++` (pre-increment) gibi işleçlerle aynı önceliklidir. Bu dönüşümler C dilinden gelen bilinçli dönüştürmedir. Aşağıda buna ilişkin bir örnek program verilmiştir;

```
#include<stdio.h>
int main() {
   float x = 9.9;
   int y = x+3.3;
   int z= (int)(x)+3.3; // x int veritipine dönüştürülmüş ve ardından toplama yapılmıştır.

   printf("x: %f\n",x);  // x: 9.900000
   printf("y: %d\n",y);  // y: 13
   printf("z: %d\n",z);  // z: 12
}
```

C++ dilinde ise bilinçli tip dönüştürme yukarıdakinin aksine aşağıdaki gibi yapılır;

1. Statik dönüşüm (static cast): Derleme zamanı (compile time) tip dönüşümleri için kullanılır.

```
#include <iostream>
using namespace std;

int main() {
    double x = 1.2;
    int sum = static_cast<int>(x + 1); // 2
    cout << sum;
}
```

2. Dinamik dönüşüm (dynamic cast): Çok biçimlilik (polymorphism) ve kalıtımda (inheritance) çalışma zamanı (run time) tip dönüşümü için kullanılır.

```
#include <iostream>
using namespace std;

class Taban {
public:
    virtual void fonksiyon() {
        cout<< "Taban sınıf fonksiyonu..." << endl;
    }
};
class Turemis : public Taban {
```

```
};

int main() {
    Taban* b = new Turemis();
    Turemis* d = dynamic_cast<Turemis*>(b);
    if (d != NULL) {
        cout << "Taban* Turemis* olarak dönüştürülebilir. d çalışır." <<endl;
        d->fonksiyon();
    }
    else
        cout << "Taban* Turemis* olarak DÖNÜŞTÜRÜLEMEZ. d çalışmaz." << endl;
}
/* Program çalıştığında:
Taban* Turemis* olarak dönüştürülebilir. d çalışır.
Taban sınıf fonksiyonu...

...Program finished with exit code 0
*/
```

3. Sabit dönüşüm (const cast): `const` veya `volatile` niteleyicilerini kaldırır veya ekler.

```
#include <iostream>
using namespace std;
class Kisi
{
private:
    int yas;
public:
    Kisi(int pYas):yas(pYas) {
    }

    // yaş alanını değiştiren bir const yasDegistir fonksiyonu
    void yasDegistir() const{
        ( const_cast<Kisi*> (this) )->yas = 30;
    }

    int getYas()  { return yas; }
};
int fonksiyon(int* ptr) {
    return (*ptr + 100);
}
int main(void) {
    Kisi ilhan(50);
    cout << "Eski Yaş: " << ilhan.getYas() << endl;
    ilhan.yasDegistir();
    cout << "Yeni Yaş: " << ilhan.getYas() << endl;

    const int val = 500;
    const int *ptr = &val;
    int *ptr1 = const_cast<int *>(ptr);
    cout << fonksiyon(ptr1);
}
/*
Eski Yaş: 50
Yeni Yaş: 30
600

...Program finished with exit code 0
*/
```

4. Yeniden yorumlama dönüşümü (reinterpret cast): Dönüştürme öncesi ve sonrası veri tipleri farklı olsa bile, bir veri tipindeki göstericiyi başka bir veri tipindeki göstericiye dönüştürmek için kullanılır. Gösterici tipinin ve göstericinin işaret ettiği verinin aynı olup olmadığı kontrol edilmez.

```
#include <iostream>
using namespace std;
int main() {
    int* p = new int(65);
    char* ch = reinterpret_cast<char*>(p);
    cout << *p << endl;
    cout << *ch << endl;
    cout << p << endl;
    cout << ch << endl;
}
/* Program Çalıştığında:
65
A
0x58af464ee2b0
A

...Program finished with exit code 0
*/
```

Bir tamsayının hangi 4 bayt bellek alanından oluştuğu *Değişken Tanımlama* başlığında anlatılmıştı. Aşağıda, tanımlanan bir tamsayının hangi baytlardan oluştuğunu gösteren bilinçli tip dönüşümlerini içeren program örneği verilmiştir;

```
#include <iostream>
#include <iomanip> // setbase(16): yazdıdılacak bir sonaki değişkenin 16 tabanında yazdırır
using namespace std;
int main(){
    int i=0x10203040;
    char* p= reinterpret_cast<char*>(&i);
    /* int gösteren adres, char içeriyor diye (char*) ifadesiyle
       bilinçli tip dönüşümü (imlplicit cast) yapılıyor. */
    cout << "i:" << setbase(10) << i << "-" <<setbase (16) << i << endl;
    cout << "1.bayt:" << setbase(10) << static_cast<int>(*p)<< "-"
                      << setbase (16) << static_cast<int>(*p) << endl;
    /* Yukarıdaki takimatta static_cast<int>(*p) ifedesiyle p ile gösterilen karakterin
       tamsayı olarak konsola yazdırılması için  bilinçli tip dönüşümü (imlplicit cast)
       yapılıyor. */
    cout << "2.bayt:" << setbase(10) << static_cast<int>(*(p+1)) << "-"
                      << setbase (16) << static_cast<int>(*(p+1)) << endl;
    cout << "3.bayt:" << setbase(10) << static_cast<int>(*(p+2)) << "-"
                      << setbase (16) << static_cast<int>(*(p+2)) << endl;
    cout << "4.bayt:" << setbase(10) << static_cast<int>(*(p+3)) << "-"
                      << setbase (16) << static_cast<int>(*(p+3)) << endl;
}
/* Program çalıştırıldığında:
i:270544960-10203040
1.bayt:64-40
2.bayt:48-30
3.bayt:32-20
4.bayt:16-10

...Program finished with exit code 0
*/
```

Tip dönüşümünün üstünlükleri;

- İşlemlerde Esneklik: Farklı veri tiplerini içeren işlemleri gerçekleştirmede esneklik sağlar. Programcının verileri bir tipten diğerine açıkça dönüştürmesini sağlayarak karışık tip aritmetiğini ve diğer işlemleri kolaylaştırır.
- Uyumluluk: Farklı veri tipleri arasında uyumluluğun sağlanmasına yardımcı olur. Programcının verileri belirli bir bağlamda kullanmadan önce uyumlu bir tipe dönüştürmesini sağlayarak veri uyumsuzluğu hatalarını önler.
- Hassasiyet Kontrolü: Hassasiyet kontrolünün kritik olduğu durumlarda, programcının özellikle sayısal işlemlerde veri tipleri arasında dönüşüm yaparak istenen hassasiyeti açıkça belirtmesini sağlar.
- Açıklık: Tip dönüşümü, programcının veri tipini değiştirme niyetini belirterek kodu daha açık hale getirir. Bu, kod okunabilirliğini artırabilir ve işlenen veri tipiyle ilgili kafa karışıklığını azaltabilir.

Tip dönüşümünün zayıf yönleri;

- Hassasiyet Kaybı: Tip dönüştürmenin en büyük dezavantajlarından biri hassasiyet kaybı potansiyelidir. Örneğin, kayan noktalı bir sayıyı tam sayıya dönüştürürken kesirli kısım kesilir ve bu da bilgi kaybına yol açar.
- Çalışma Zamanı Yükü: Bilinçli tip dönüştürme genellikle program yürütme sırasında dönüştürmenin gerçekleştirilmesi gerektiğinden çalışma zamanı yüküne neden olur. Bu ek işlem, özellikle tip dönüştürmenin sık olduğu durumlarda performansı etkileyebilir.
- Derleyici Uyarıları ve Hataları: Yanlış veya güvenli olmayan tip dönüştürme, derleyici uyarılarına veya hatalarına yol açabilir. Örneğin, uyumsuz tipler arasında dönüştürme yapmaya çalışmak veya geçersiz dönüştürme sözdizimi kullanmak, hata ayıklaması zor olabilecek sorunlara yol açabilir.
- Tanımsız Davranış Potansiyeli: Bazı durumlarda, tip dönüştürme, özellikle uyumsuz tipler arasında dönüştürme yaparken veya dönüştürülen değer hedef tipin aralığının dışında olduğunda tanımsız davranışa yol açabilir. Bu, programda öngörülemez durumlara neden olabilir.
- Kod Bakım Zorlukları: Bilinçli tip dönüşümüne kodun bakımı ve anlaşılması, özellikle karmaşık veya iç içe ifadelerde gerçekleştirildiğinde, daha zor hale gelebilir. Bu, artan kod karmaşıklığına ve azalan okunabilirliğe yol açabilir.
- Taşınabilirlik Endişeleri: Bilinçli tip dönüşümüne farklı platformlar veya derleyiciler arasında daha az taşınabilir olabilir. C'de tip dönüşümünün davranışı değişebilir ve belirli veri tipi boyutları hakkındaki varsayımlar tüm ortamlarda geçerli olmayabilir.

Tip Çıkarımı

Tip çıkarımı (type inference), bir ifadenin veri tipinin otomatik olarak belirlenmesi anlamına gelir. C++ dilinin 2011 sürümünden sonra, `auto` ve `decltype` gibi anahtar sözcükler bu amaçla dile dahil edilmiştir. Böylece bir programcının tür çıkarımını derleyicinin kendisine bırakmasına olanak sağlıyor. Tür çıkarımı yetenekleriyle, derleyicinin zaten bildiği şeyleri yazmak gerekmez.

Tüm tipler yalnızca derleyici aşamasında belirlendiği için, derleme için gereken süre biraz artar ancak programın çalışma süresini etkilemez.

C++ dilinde `auto` anahtar sözcüğü (auto keyword), bildirilen değişkenin tipinin ilk değer verme işleminden otomatik olarak çıkarır. Fonksiyonlar söz konusu olduğunda, dönüş tipleri `auto` ise bu, çalışma zamanında (run time) dönüş tipi belirlenir.

```
#include <iostream>
using namespace std;
int main() {
    // auto a; hata verir. Çünkü ilk değer verilmediğinden veri tipi belirlenemez!
    auto x = 4;
    auto y = 3.37;
    auto z = 3.37f;
    auto c = 'a';
    auto ptr = &x;
    auto pptr = &ptr; //pointer to a pointer
```

```
    cout << typeid(x).name() << endl  //i
         << typeid(y).name() << endl  //d
         << typeid(z).name() << endl  //f
         << typeid(c).name() << endl  //c
         << typeid(ptr).name() << endl  //pi
         << typeid(pptr).name() << endl; // ppi
}
```

C++ dilinde `auto` anahtar sözcüğü belirli bir tipe sahip bir değişken bildirimine olanak tanırken, `decltype` değişkenden tip çıkarmanıza olanak tanır; dolayısıyla `decltype`, geçirilen ifadenin türünü değerlendiren bir tip işlecidir (decltype operator).

```
int fun1() { return 10; }
char fun2() { return 'g'; }
int main() {
    decltype(fun1()) x;
    decltype(fun2()) y;

    cout << typeid(x).name() << endl; //i
    cout << typeid(y).name() << endl; //c

    char a = 65;
    decltype(a) b = a + 5;

    cout << typeid(b).name() << endl; //c
}
```

Lamda İfadeleri

Bir lamda ifadesi (lambda expression), basit fonksiyon nesneleri oluşturmak için özlü bir yol sağlar. 'Lamda' adı, *Alonzo Church* tarafından 1930'larda mantık ve hesaplanabilme hakkındaki soruları araştırmak için icat edilen matematiksel bir form olan lamda hesabından gelir. Lamda hesabı, fonksiyonel bir programlama dili olan LISP'in temelini oluşturmuştur.

Bir lamda ifadesi genellikle çağrılabilir bir nesne alan fonksiyonlara argüman olarak kullanılır. Lamda ifadesi, yalnızca argüman olarak geçirildiğinde kullanılacak olan gövdesi tanımlı bir fonksiyon oluşturmaktan daha basit olabilir. Bu gibi durumlarda, lamda ifadeleri genellikle tercih edilir çünkü işlev nesnelerini satır içi (inline) tanımlamaya izin verirler.

Bir lamda tipik olarak üç bölümden oluşur; bir yakalama listesi `[]`, isteğe bağlı bir parametre listesi `()` ve bir gövde `{}`, bunların hepsi boş olabilir;

`[]` yakalama listesidir. Varsayılan olarak, çevreleyen kapsamın değişkenlerine bir lamda tarafından erişilemez. Bir değişkeni yakalamak, onu lamda içinde, bir kopya veya bir referans olarak erişilebilir hale getirir. Yakalanan değişkenler lamdanın bir parçası haline gelir; fonksiyon argümanlarının aksine, lamda çağrılırken geçirilmeleri gerekmez;

```
int a = 0;
auto f = []() { return a*9; }; // Hata: 'a' erişilemez
auto f = [a]() { return a*9; }; // 'a' değerinden yakalanı
auto f = [&a]() { return a++; }; // 'a' referansından yakalanır
auto call_b = f(); // Lamda fonksiyonu çağrılır
```

`()` parametre listesidir ve normal fonksiyonlardakiyle hemen hemen aynıdır. Lamda hiçbir argüman almazsa, bu parantezler atlanabilir (lamdayı değiştirilebilir olarak bildirmeniz gerekmediği sürece). Bu iki lamda eşdeğerdir;

```
auto call_foo = [x](){ x.foo(); };
auto call_foo2 = [x]{ x.foo(); };
```

Parametre listesi gerçek tipler yerine yer tutucu tipi olan `auto` tipini kullanabilir. Bunu yaparak, bu argüman bir fonksiyon şablonunun şablon parametresi gibi davranır. Aşağıdaki lamdalar, genel kodda bir vektörü sıralamak istediğinizde eşdeğerdir;

```
auto sort_cpp11 = [](std::vector<T>::const_reference lhs, std::vector<T>::const_reference rhs)
                    { return lhs < rhs; };
auto sort_cpp14 = [](const auto &lhs, const auto &rhs) { return lhs < rhs; };
```

`{}` normal fonksiyonlardakiyle aynı olan gövdedir. Bir lamda ifadesinin sonuç nesnesi, diğer fonksiyon nesnelerinde olduğu gibi, `()` işleci kullanılarak yapılır:

```
int multiplier = 5;
auto timesFive = [multiplier](int a) { return a * multiplier; };
std::out << timesFive(2); // 10
multiplier = 15;
std::out << timesFive(2); // 2*5 = 10
```

Varsayılan olarak, bir lamda ifadesinin dönüş tipi türetilir. Aşağıdaki durumda dönüş tipi `bool` olur.

```
[](){ return true; };
```

Aşağıdaki sözdizimini kullanarak dönüş türünü elle da belirtebilirsiniz:

```
[]() -> bool { return true; };
```

Lamda ifadeleriyle basit bir C++ programı aşağıdaki bibi yazılabilir;

```
#include <iostream>
#include <iomanip>
using namespace std;

auto main() -> int
{
    int const   satir  = 3;
    int const   sutun  =4;
    int const   matris[satir][sutun] =
    {
        { 1, 2, 3, 4},
        { 5, 6, 7, 8},
        { 9,10,11,12}
    };
    for( int y = 0; y < satir; ++y )
    {
        for( int x = 0; x < sutun; ++x )
        {
            cout << setw( 4 ) << matris[y][x];
        }
        cout << endl;
    }
}
 /*Program Çıktısı:
   1   2   3   4
   5   6   7   8
   9  10  11  12
 */
```

Lamda İfadeleriyle Yakalanan Nesneler

Lamdadaki değerle yakalanan nesneler varsayılan olarak değişmezdir (immutable). Bunun nedeni, oluşturulan kapatma nesnesinin işleci olan ()'i varsayılan olarak `const` olmasıdır.

```
auto func = [c = 0](){++c; std::cout << c;}; /* ++c lamda'nın durumunu değiştirmeye çalıştığı
için derlenemez.*/
```

Değiştirilmeye, `mutable` anahtar sözcüğü kullanılarak izin verilebilir; bu, yakın nesnenin işleci olan ()'i sabit olmayan hale getirir:

```
auto func = [c = 0]() mutable {++c; std::cout << c;};
```

Dönüş tipiyle birlikte kullanıldığında, `mutable` ondan önce gelir;

```
auto func = [c = 0]() mutable -> int {++c; std::cout << c; return c;};
```

Dönüş tipi belirlenirken bazı durumlar göz önüne alınmalıdır;

```
auto l = [](int value) {
    return value > 10; // Returns bool
};
auto l = [](int value) {
    if (value < 10) {
        return 1; // Hata: lamda dönüş tipini belirleyemedi int?

    } else {
        return 1.5; // Hata: lamda dönüş tipini belirleyemedi double?
    }
};
auto l2 = [](int value) -> double {// Dönüş tipi 'double'
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

explicit Sıfatı

C++ dilinde explicit anahtar kelimesi, tiplerin üstü kapalı olarak dönüştürülmemesi için yapıcıları (constructor) nitelendirmek için kullanılır. Aşağıdaki kodu göz önüne alalım;

```
#include <iostream>
using namespace std;
class Karmasik {
private:
    double gercek;
    double sanal;
public:
    Karmasik(double r = 0.0, double i = 0.0) :
           gercek(r), sanal(i)
    {
    }
    bool operator == (Karmasik c) //Bir başka karşamış sayıya eşit mi?
    {
        return (gercek == c.gercek && sanal == c.sanal);
    }
};
int main()
{
    Karmasik k(3.0, 0.0);

    if (k == 3.0) cout << "Aynı";
    else cout << "Farklı";
}
/*Program Çıktısı:
Aynı
*/
```

Aşırı yüklenen eşitlik işlecinde kontrol ifadesindeki `&&` işlecinin solundaki ifade doğru ise sağdakine bakılmaz. Bu nedenle program çıktısı `Aynı` olacaktır. Bunu önlemek için explicit anahtara kelimesi yapıcı önünde kullanılır.

```
    explicit Karmasik(double r = 0.0, double i = 0.0) :
           gercek(r), sanal(i)
    {
    }
```

Bu durumda program derlenince aşağıdaki derleme hatası alınır;

```
no match for ‘operator==’ (operand types are ‘Karmasik’ and ‘double’)
```

Hala double değerlerini `Karmasik` tipe dönüştürebiliriz, ancak şimdi açıkça tip dönüşümü yapmamız gerekiyor;

```
    if (k == (Karmasik) 3.0) cout << "Aynı";
    else cout << "Farklı";
```

ŞABLONLAR

Şablon Nedir?

Şablonlar (template), farklı veri tipleriyle çalışan fonksiyonlar, sınıflar, algoritmaları tekrar tekrar yazmamak için bir programlama türü olan jenerik programlamanın (generic programming) temelini oluşturur. Yani jenerik programlamada veri tiplerine bağlı olmadan fonksiyon, sınıf ve algoritma oluşturmamızı sağlar.

Fonksiyon Şablonları

Fonksiyon şablonları (function template), aynı mantığı yeniden yazmadan bir fonksiyonun farklı veri türleri üzerinde çalışmasını sağlayan bir fonksiyon planı tanımlar. Bu plan aşağıdaki şekilde tanımlanır;

```
template <typename değişken-kimliği> foksiyon-bildirimi;
```

Burada `typename`, işlenecek bir tipe verilen bir kimliktir. Yani bir şablonun bildiriminde bir parametre tanıtır. Fonksiyon şablonuna aşağıdaki örnek verilebilir;

```
#include <iostream>
#include <string>
using namespace std;

template <typename Degisken> void Degistir(Degisken& a, Degisken& b) {
   Degisken temp= a;
   a=b;
   b=temp;
}

int main () {
   int i = 10;
   int j = 20;
   Degistir(i, j); // Derleyici parametreleri int veri tipinde kabul etti
   cout << "Degiştir(i, j) sonrası i: " << i <<" , j: " << j << endl;

   double f1 = 15.0;
   double f2 = 25.5;
   Degistir(f1, f2); // Derleyici parametreleri double veri tipinde kabul etti
   cout << "Degiştir(f1, f2) sonrası f1: " << f1 <<" , f2: " << f2 << endl;

   string s1 = "Hello";
   string s2 = "World";
   Degistir(s1, s2); // Derleyici parametreleri string veri tipinde kabul etti
   cout << "Degiştir(s1, s2) sonrası s1: " << s1 <<" , s2:"  << s2 << endl;
}
/* Program Çıktısı:
Degiştir(i, j) sonrası i: 20 , j: 10
Degiştir(f1, f2) sonrası f1: 25.5 , f2: 15
Degiştir(s1, s2) sonrası s1: World , s2: Hello

...Program finished with exit code 0
*/
```

Görüldüğü üzere her veri tipi için değiştir fonksiyonu yazmak yerine bir adet şablon fonksiyon ile bütün ihtiyacımız karşılanmıştır.

Çok Değişkenli Fonksiyon Şablonları

Değişken sayıda argüman alabilen yada çok değişkenli fonksiyon şablonları (variadic function template), herhangi bir değişken (sıfır veya daha fazla) sayıda argüman alabilen sınıf veya fonksiyon şablonlarıdır. C++ dilinde şablonlar, yalnızca bildirim sırasında belirtilmesi gereken sabit sayıda parametreye sahip olabilir. Ancak *Douglas Gregor* ve *Jaakko Järvi* tarafından ortaya koyulan değişkenli şablonlar (variadic template) bu sorunun üstesinden gelmeye yardımcı olur.

```
#include <iostream>
using namespace std;

template <typename T> void yaz(const T& t) {
    cout << t << endl;
}

template <typename Ilk, typename... KalanParametreler>
    void yaz(const Ilk& ilk, const KalanParametreler&... kalan) {
       cout << ilk << ", ";
       yaz(kalan...); // özyinelemeli çağrı (recursive call)
}

int main() {
    yaz(1, 2, 3.14, "Merhaba ", "C++", "Öğreniyorum");
    yaz(10, 20, 30);
    yaz(1, 2, 3, 4, 5);
}
/* Program Çıktısı:
1, 2, 3.14, Merhaba , C++, Öğreniyorum
10, 20, 30
1, 2, 3, 4, 5

...Program finished with exit code 0
*/
```

Sınıf Şablonları

Fonksiyonlara benzer şekilde, sınıf şablonları aynı zamanda herhangi bir veri türüyle çalışabilen sınıflar oluşturmak için bir plan tanımlar. Aşağıdaki şekilde tanımlanır;

```
template <veritipi değişken-kimliği[,veritipi değişken-kimliği-2]>
class şablon-sınıf-kimliği {
  private:
    değişken-kimliği örnek-değişkeni;
    [değişken-kimliği-2 örnek-değişkeni2;]
    ... .. ...
  public:
    değişken-kimliği fonksiyonKimligi(değişken-kimliği arg);
    ... .. ...
};
```

Burada, `sınıf-kimliği`, bir sınıf şablona konu olan yer tutucu sınıf kimliğidir. Virgülle ayrılmış bir liste kullanarak birden fazla genel veri tipi tanımlanabilir.

Bir şablon sınıftan bir nesne imal etme aşağıdaki gibi yapılır;

```
şablon-sınıf-kimliği <veritipi> object-sınıf-kimliği;
```

Aşağıda bir şablon sınıf örneği verilmiştir;

```
#include <iostream>
using namespace std;
```

```
template <class veriTipi>
class Sayi {
   private:
    veriTipi num; // num kimlikli bir alan (field) tanımlandı

   public:
    Sayi(veriTipi n) : num(n) {}   // constructor

    veriTipi getNum() {
        return num;
    }
};

int main() {
    Sayi<int> intAlanliSinifNesnesi(5);
    Sayi<double> doubleAlanliSinifNesnesi(15.75);

    cout << "int Sayı<int>.getnum() = "
         << intAlanliSinifNesnesi.getNum() << endl;
    cout << "double Sayı<double>.getnum() = "
         << doubleAlanliSinifNesnesi.getNum() << endl;
}
```

Görüldüğü üzere hem `int` hem de `double` için ayrı ayrı sınıf yazmadan tek bir şablon sınıf ile tüm ihtiyacımız karşılanmıştır.

Şablon kullanmanın çeşitli üstünlükleri vardır;

- Yeniden Kullanılabilir Kod: Şablonlar, tüm veri tipleriyle çalışan genel kod yazmanıza olanak tanır ve böylece her gerekli tip için aynı kodu yazma ihtiyacını ortadan kaldırır. Bu da kod tekrarını ve büyümesini azaltarak geliştirme süresinden tasarruf sağlar.
- Azaltılmış Bakım: Bir şablon güncellendiğinde tüm örneklemelerdeki değişiklikleri otomatik olarak yapılmış olur. Bu, hata düzeltme, düzeltmeleri yapma ve tüm örneklemelerin faydalarını görme açısından üstünlük sağlar.
- Gelişmiş Performans: Şablon örneklemeleri derleme zamanında gerçekleşir ve çalışma zamanı hatalarını azaltır. Derleyici, kodu belirli veri tipleri için optimize eder.
- İyi Organize Edilmiş Kaynak Kod: Şablonlar algoritma mantığı veri türünden ayırdığından, geliştirme senaryosu açısından daha da iyi olan modüler kod oluşturmaya yardımcı olur. Bir kodun farklı uygulamalarını aramayı azaltmaya yardımcı olur.

Akıllı Göstericiler

Akıllı göstericiler (smart pointer) sınıf şablonlarıdır. `std::unique_ptr`, dinamik olarak depolanan bir nesnenin ömrünü yöneten bir sınıf şablonudur. Dinamik nesne herhangi bir zamanda yalnızca bir `std::unique_ptr` örneğine aittir.

```
std::unique_ptr<int> ptr = std::make_unique<int>(20); /* Benzersiz bir göstericiye ait 20
değerine sahip dinamik bir int oluşturur */
```

Sadece `ptr` değişkeni dinamik olarak tahsis edilmiş bir tamsayıya (`int`) bir gösterici tutar. Bir nesneye sahip olan benzersiz bir gösterici kapsam dışına çıktığında, sahip olunan nesne silinir, yani nesne sınıf türündeyse onun yıkıcısı çağrılır ve o nesnenin belleği serbest bırakılır.

```
std::unique_ptr<int> ptr = std::make_unique<int>(59); /* Benzersiz bir göstericiye ait 59
değerine sahip dinamik bir int oluşturur */
std::unique_ptr<int[]> ptr2 = std::make_unique<int[]>(15); /* 15 adet int tutan diziye ait
benzersiz bir gösterici */
```

Akıllı göstericinin içeriğinin sahipliğini `std::move` kullanarak başka bir göstericiye aktarabilirsiniz, bu durum orijinal akıllı işaretçinin `nullptr`'yi işaret etmesine neden olur.

```
std::unique_ptr<int> ptr = std::make_unique<int>();
*ptr = 1; // gösterilen değer 1 yapıldı

std::unique_ptr<int> ptr2 = std::move(ptr);
int a = *ptr2; // 'a' is 1
int b = *ptr; // Hata! 'ptr' = 'nullptr'
```

`std::shared_ptr` sınıf şablonu ise bir nesnenin sahipliğini diğer paylaşılan göstericilerle paylaşır.

```
std::shared_ptr<Foo> firstShared = std::make_shared<Foo>(/*args*/);
```

Aynı nesneyi paylaşan birden fazla akıllı gösterici oluşturmak için, ilk paylaşılan gösterici takma adla adlandıran başka bir `shared_ptr` oluşturmamız gerekir. Bunu yapmanın 2 yolu vardır:

```
std::shared_ptr<Foo> secondShared(firstShared); // Birinci yol
std::shared_ptr<Foo> secondShared;
secondShared = firstShared; // 2. Yol. Atama ile
```

Akıllı gösterici tıpkı bilinen göstericiler gibi çalışır. Buda `*` işlecini (dereference operator) kullanabileceğiniz anlamına gelir. `->` işleci de aynı şekilde çalışır:

```
secondShared->test();
```

Ne yazık ki, `make_shared<>` kullanarak Dizileri tahsis etmenin doğrudan bir yolu yoktur. C++17 ile, `shared_ptr` dizi tipleri için özel destek kazandı. Artık `array-deleter`'ı açıkça belirtmek gerekmiyor ve paylaşılan işaretçi [] dizi dizin operatörü kullanılarak başvurudan kaldırılabilir:

```
std::shared_ptr<int[]> sh(new int[10]);
sh[0] = 42;
```

Paylaşılan göstericiler, sahip olduğu nesnenin bir alt nesnesine işaret edebilir;

```
struct Foo { int x; };
std::shared_ptr<Foo> p1 = std::make_shared<Foo>();
std::shared_ptr<int> p2(p1, &p1->x);
```

Standart Şablon Kütüphanesi

Standart şablon kütüphanesi (Standart Template Library), C++ dilinin en güçlü kütüphanesidir. Jenerik programlama (generic programming) ile birçok algoritma programcıya bu kütüphane ile sunulur.

Buraya kadar anlatılan şablon (template) kavramını anlamışsanız C++ dili Standart Şablon Kütüphanesi, vektörler, listeler, kuyruklar ve yığınlar gibi birçok popüler ve yaygın olarak kullanılan algoritmayı ve veri yapısını uygulayan şablonlarla genel amaçlı sınıflar ve fonksiyonlar sağlamak için güçlü bir şablon sınıfları kümesidir[19]. Bu küme dört başlıkta incelebilir;

- Konteynerler: Belirli bir türdeki nesne kümeleri yani koleksiyonlarını yönetmek için kullanılır. Deque, list, vector, map gibi çeşitli farklı konteyner türleri vardır. Konteynerler, tuttukları verilerden bağımsız dinamik veri yapılarıdır (dynamic data structure).
- Algoritmalar: Konteynerler üzerinde etki eder. Konteynerlerin içeriklerinin başlatılmasını, sıralanmasını, aranmasını ve dönüştürülmesini gerçekleştireceğiniz araçları sağlarlar.
- Yineleyiciler: Nesne koleksiyonlarının elemanları arasında gezinmek için kullanılır. Bu koleksiyonlar, kapsayıcılar veya kapsayıcıların alt kümeleri olabilir.
- Fonksiyon Nesneleri: Bir fonksiyon sanki bir fonksiyonmuş gibi çağrılabilen bir nesnedir. Normal fonksiyonlar gibi çağrılabilir. Bu yetenek, fonksiyon çağrı işleci (function call operator) olan `()`'in aşırı yüklenmesiyle elde edilir. Bu kelimelerin kısaltılmış hali olan functor olarak adlandırılırlar.

Aşağıda bir diziye benzeyen ancak büyümesi durumunda kendi depolama gereksinimlerini otomatik olarak karşılayan vektör konteynerini kullanan program örneği verilmiştir;

[19] https://www.tutorialspoint.com/cplusplus/cpp_stl_tutorial.htm sayfasından çoğu derlenmiştir.

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
   vector<double> vec; // int değerler tutacak vector konteyneri
   int i;
   cout << "Vector Uzunluğu = " << vec.size() << endl;
   for(i = 0; i < 5; i++) { // 5 değer vektöre koyuluyor.
      vec.push_back(i*1.5);
   }
   cout << "Vektörün Son Uzunluğu = " << vec.size() << endl;

   for(i = 0; i < 5; i++) {// vektör elemanlarına erişiliyor
      cout << "vec[" << i << "] = " << vec[i] << endl;
   }

   // yineleyiciler üzerinden vektör elemanlarına erişim:
   vector<double>::iterator v = vec.begin();
   while( v != vec.end()) {
      cout  << *v << endl;
      v++;
   }
}
/*Program Çıktısı:
Vector Uzunluğu = 0
Vektörün Son Uzunluğu = 5
vec[0] = 0
vec[1] = 1.5
vec[2] = 3
vec[3] = 4.5
vec[4] = 6
0
1.5
3
4.5
6

...Program finished with exit code 0
*/
```

Programı incelediğimizde `double` olan veri tipini değiştirme ihtiyacımız varsa sadece iki satırda değişiklik yapmamız yeterlidir. Program içinde yer alan;

- `push_back()` üye fonksiyonu vektörün sonuna değer ekler ve gerektiği gibi boyutunu genişletir.
- `size()` fonksiyonu vektörün boyutunu görüntüler.
- `begin()` fonksiyonu vektörün başlangıcını gösteren bir yineleyici döndürür.
- `end()` fonksiyonu vektörün sonunu gösteren bir yineleyici döndürür.

Konteyner Şablonları

Konteynerler, vektörler, listeler, kümeler ve haritalar gibi veri veya nesne küme/koleksiyonlarını depolamak ve yönetmek için kullanılan veri yapılarıdır (data structures). Dinamik veri yapıları yapısal programlama sürecinde çokça geliştirilen ve artık standart hale gelen koleksiyonlardır. C++ dilinde koleksiyonlar, artık hazır bir kütüphanedir.

Ardışık konteynerler (sequential container), elemanları belirli bir doğrusal düzende depolarlar. Elemanları ve sırasını ve düzenlemesini yönetmek için işlevlere doğrudan erişim sağlarlar;

- Dizi (array): Bunlar sabit boyutlu eleman koleksiyonlarıdır.

- Vektör (vector): Gerektiğinde boyutunu değiştirebilen dinamik bir dizidir.
- İki Uçlu Kuyruk (deque): Her iki uçtan hızlı ekleme ve silmeye izin veren çift uçlu kuyruk.
- Liste (list): Çift yönlü yinelemeye izin veren çift bağlantılı liste.
- İleri Liste (forward list): Verimli ekleme ve silmeye izin veren ancak yalnızca bir yönde gezinmeye izin veren tek bağlantılı listedir.
- Dizgi (string): Diğer ardışık konteynerlere benzeyen dinamik bir karakter dizisidir.

Konteyner kütüphanesi, programcıların kuyruklar, listeler ve yığınlar gibi yaygın veri yapılarını kolayca uygulamasına olanak tanıyan sınıf şablonları ve algoritmaları içerir. Bu konteynerlerden hangisini ne zaman kullanacağımız çözülecek problemle ilgilidir [20].

İlişkili konteynerler (associated container), elemanları anahtarlara (key) göre hızlı bir şekilde geri almaya izin veren sıralı bir düzende depolar. Bu konteynerler, her bir elemanın benzersiz bir anahtarla ilişkilendirildiği bir anahtar-değer çifti yapısında çalışır. Bu anahtar, karşılık gelen değere hızlı erişim için kullanılır;

- Küme (set): Belirli bir düzende sıralanmış benzersiz elemanların bir koleksiyonudur.
- Harita (map): Anahtarların benzersiz olduğu anahtar-değer çiftlerinin bir koleksiyonudur.
- Çoklu Küme (multiset): Bir kümeye benzer, ancak yinelenen elemanların izin verir.
- Çoklu Harita (multimap): Bir haritaya benzer, ancak yinelenen anahtarlara izin verir.

Sıralanmamış ilişkisel konteynerler (unordered associated container), elemanları sırasız bir şekilde depolar ve anahtarlara (key) dayalı olarak verimli erişim, ekleme ve silmeye olanak tanır. Elemanların sıralı bir düzenini korumak yerine verileri düzenlemek için karma kullanırlar;

- Sıralanmamış Küme (unordered set): Belirli bir sırası olmayan, benzersiz elemanlardan oluşan bir koleksiyondur.
- Sıralanmamış Harita (unordered map): Belirli bir sırası olmayan, anahtar-değer çiftlerinden oluşan bir koleksiyondur.
- Sıralanmamış Çoklu Küme (unordered multiset): Belirli bir sırası olmayan, yinelenen elemanlara izin verir.
- Sıralanmamış Çoklu Harita (unordered multimap): Belirli bir sırası olmayan, yinelenen anahtarlara izin verir.

Konteyner adaptörleri (container adapter), mevcut konteynerler için farklı bir ara yüz sağlar;

- Yığın (stack): Son Giren İlk Çıkar (last in first out-LIFO) ilkesini izleyen bir veri yapısıdır.
- Kuyruk (queue): İlk Giren İlk Çıkar (first in first out-FIFO) ilkesini izleyen bir veri yapısıdır.
- Öncelikli Kuyruk (priority queue): Elemanların önceliğe göre kaldırıldığı özel bir kuyruk türüdür.

Algoritma Şablonları

Standart Şablon Kütüphanesi'ndeki algoritmalar (algotrithm), konteynerler üzerinde işlem yapmak için özel olarak tasarlanmış büyük bir fonksiyon kümesidir. Bu fonksiyonlar, konteynerlerin iç yapılarını bilmeye gerek kalmadan konteynerler arasında geçiş yapmak için kullanılan yineleyiciler (iterator) kullanılarak uygulanır.

Sıra değiştirmeyen algoritmalar;

- for_each: Bir aralıktaki her bir elemanı bulmak için bir fonksiyonda kullanılır.
- count: Bir aralıktaki değerin oluşum sayısını sayar.
- find(): Bir aralıktaki değerin ilk oluşumunu bulur.
- find_if: Bir yordamı karşılayan ilk elemanı bulur.
- find_if_not: Bir yordamı karşılamayan ilk elemanı bulur.
- equal: İki aralığın eşit olup olmadığını kontrol eder.

[20] https://docs.google.com/drawings/d/1c-qvy499kxaYXM70DM34rOnEBCzgQLopyNDEXdaK0eU/edit

– search: Bir dizi içinde bir alt diziyi arar.

Sıra değiştiren algoritmalar;

– copy(): Elemanları bir aralıktan diğerine kopyalar.
– copy_if: Bir yordamı karşılayan elemanları başka bir aralığa kopyalar.
– copy_n: Belirli sayıda elemanı bir aralıktan diğerine kopyalar.
– move: Elemanları bir aralıktan diğerine taşır.
– transform(): Bir aralığa bir fonksiyon uygular ve sonucu depolar.
– remove: Belirli bir değere sahip elemanları bir aralıktan kaldırır.
– remove_if: Bir yordamı karşılayan elemanları kaldırır.
– unique: Ardışık yinelenen elemanları kaldırır.
– reverse(): Bir aralıktaki elemanların sırasını tersine çevirir.
– swap(): Eelemanları değiştirir.

Sıralama algoritmaları;

– sort: Bir aralıktaki elemanları sıralar.
– stable_sort: Eşdeğer elemanların göreli sırasını koruyarak elemanları sıralar.
– partial_sort: Bir aralığın bir kısmını sıralar.
– nth_element: Aralığı, n'inci elemanın son konumunda olacak şekilde bölümlere ayırır.

Arama algoritmaları;

– binary_search: Sıralanmış bir aralıkta bir elemanın bulunup bulunmadığını kontrol eder.
– lower_bound: Sırayı korumak için bir elemanın eklenebileceği ilk konumu arar.
– upper_bound: Belirtilen değerden büyük olan ilk elemanın konumunu arar.
– binary_search: Sıralanmış bir aralıkta bir elemanın bulunup bulunmadığını kontrol eder.
– lower_bound: Sırayı korumak için bir elemanın eklenebileceği ilk konumu bulur.
– upper_bound: Belirtilen değerden büyük olan ilk elemanın konumunu arar.
– equal_range: Eşit elemanların aralığını döndürür.

Öbek (heap) algoritmaları;

– make_heap: Bir aralıktan bir öbek oluşturur.
– push_heap: Bir öbeğe bir eleman ekler.
– pop_heap: Bir öbekten en büyük eleman kaldırır.
– sort_heap: Bir öbekteki elemanları sıralar.

Küme algoritmaları;

– set_union: İki kümenin birleşimini hesaplar.
– set_intersection: İki kümenin kesişimini hesaplar.
– set_difference: İki küme arasındaki farkı hesaplar.
– set_symmetric_difference: İki küme arasındaki simetrik farkı hesaplar.

Sayısal algoritmalar;

– accumulate: bir aralığın toplamını (veya diğer işlemleri) hesaplar.
– inner_product: İki aralığın iç çarpımını hesaplar.
– adjacent_difference: Bitişik elemanlar arasındaki farkları hesaplar.
– partial_sum: bir aralığın kısmi toplamlarını hesaplar.

Diğer algoritmalar;

– generate: Bir fonksiyon tarafından üretilen değerlerle bir aralığı doldurur.
– shuffle: Bir aralıktaki elemanları rastgele karıştırır.
– partition: Eelemanları bir öngörüye göre iki gruba ayırır.

Yineleme Şablonları

Standart Şablon Kütüphanesi'ndeki yineleyiciler (iterator), bir konteyner içindeki elemanlara gösterici görevi gören ve verilere erişmek ve bunları düzenlemek için tekdüze bir ara yüz sağlayan nesnelerdir. Algoritmalar ve konteynerler arasında bir köprü görevi görürler;

- Giriş Yineleyicileri (input iterator): Elemanlara salt okunur erişime izin verirler.
- Çıkış Yineleyicileri (output iterator): Elemanlara salt yazılır erişime izin verirler.
- İleri Yineleyiciler (forward iterator): Artırılabilen okuma ve yazmaya izin verirler.
- Çift Yönlü Yineleyiciler (bidirectional iterators): Artırılabilir ve azaltılabilirler.
- Rastgele Erişim Yineleyicileri (Random Access Iterator): Aritmetik işlemleri desteklerler ve elemanlara doğrudan erişebilirler.

Fonksiyon Nesneleri

Fonksiyon nesnesi (function object), fonksiyon çağrı işleci (functor) ile bir fonksiyonmuş gibi çağrılabilen bir nesnedir. Fonksiyon nesneleri, bir fonksiyon veya fonksiyon göstericisiymiş gibi ele alınabilen nesnelerdir. Aşağıda bir dizinin elemanlarını 1 artıran bir dönüşüm programı verilmiştir;

```
#include <bits/stdc++.h> //transform()
using namespace std;

int artirim(int x) { return (x+1); }

int main(){
	int dizi[] = {1, 2, 3, 4, 5};
	int adet = sizeof(dizi)/sizeof(dizi[0]);
	/*
	    Aşağıdaki dönüşüm fonksiyonu ile
	    dizi elemanları 1 artırılıyor
	*/
	transform(dizi, dizi+adet, dizi, artirim);
	for (int i=0; i<adet; i++)
		cout << dizi[i] <<" ";
	return 0;
}
/* Program Çıktısı:
2 3 4 5 6

...Program finished with exit code 0
*/
```

Aşağıda bir fonksiyon nesnesi üzerinden aynı işlemi yapan bir program örneği verilmiştir;

```
#include <bits/stdc++.h> // transform
using namespace std;

class artirim// Functor
{
private:
	int sayi;
public:
	artirim(int n) : sayi(n) { }

	/* Aşağıda verilen işleç önyüklemesi ile
	arr_sayi kadar artırım işlemi yapılacaktır. */
	int operator () (int pMiktar) const {
		return sayi + pMiktar;
	}
};
```

```
int main() {
	int dizi[] = {1, 2, 3, 4, 5};
	int adet = sizeof(dizi)/sizeof(dizi[0]);
	int miktar = 5;
	transform(dizi, dizi+adet, dizi, artirim(miktar));
	for (int i=0; i<adet; i++)
		cout << dizi[i] << " ";
}
/*Program Çıktısı:
6 7 8 9 10

...Program finished with exit code 0
*/
```

Aritmetik functor (aritmetic functor), aritmetik operatörler temel matematiksel işlemleri gerçekleştirmek için kullanılır. Toplama (+), Çıkarma (-), Çarpma (*), Bölme (/), Kalan (%), Negatif (−) işleç (operator) işlevlerini yerine getirirler.

Karşılaştırma fonksiyon nesneleri, özellikle konteynerlerde sıralama veya arama yapmak için değerleri karşılaştırmak amacıyla kullanılır. Küçüktür (<), Büyüktür (>), Küçüktür veya Eşittir (≤), Büyük veya Eşit (≥), Eşit (=), Eşit Değil (! =) işleç işlevlerini yerine getirirler.

Mantıksal fonksiyon nesneleri, Boole mantığını içeren senaryolar için mantıksal işlemleri gerçekleştirir. Mantıksal VE (&&), Mantıksal VEYA (||) ve Mantıksal DEĞİL (!) işleç işlevlerini yerine getirirler.

Bit düzeyi fonksiyon nesneleri, bit düzeyi işlemleri gerçekleştirir. Bit düzeyi VE (&), VEYA (|) ve ÖZEL VEYA (^) işleç işlevlerini yerine getirirler.

std::array

C dilinden bildiğimiz dizilere benzer ancak her türlü veri tipiyle oluşturulabilir. Bu sınıf şablonunun iki parametresi vardır; `class T` Dizi elemanlarının veri tipini belirtir, `std::size_t N` ise Dizide kaç eleman olduğunu belirtir.

Dizi tanımlama ve ilk değer verme (initialize);

T'nin sayılarla ifade edilen (scalar) veri tiplerini barındırması halinde aşağıdaki şekilde ilk değer verilebilir;

1. Dizilere ilk değer verme yöntemini kullanarak

```
std::array<int, 3> a { 0, 1, 2 };
// buna eşdeğer ifade:
std::array<int, 3> a = { 0, 1, 2 };
```

2. Kopya yapıcı (copy constructor) kullanarak

```
std::array<int, 3> a { 0, 1, 2 };
std::array<int, 3> a2(a);
// buna eşdeğer ifade
std::array<int, 3> a2 = a;
```

3. Tasıma yapıcısı (move constructor) kullanarak;

```
std::array<int, 3> a = std::array<int, 3>{ 0, 1, 2 };
```

T'nin sayılarla ifade edilmeyen (non scalar) veri tiplerini barındırması halinde aşağıdaki şekilde ilk değer verilebilir;

```
struct A { int values[3]; };
```

4. Tırnaklı parantez ile ilk değer verilebilir

```
std::array<A, 2> a{ 0, 1, 2, 3, 4, 5 };
// buna eşdeğer ifade:
```

```
std::array<A, 2> a = { 0, 1, 2, 3, 4, 5 };
```

5. Tırnaklı parantezleri alt elemanlarda kullanarak;

```
std::array<A, 2> a{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };
// buna eşdeğer ifade:
std::array<A, 2> a = { A{ 0, 1, 2 }, A{ 3, 4, 5 } };
```

6. Tamamıyla tırnaklı parantez kullanarak;

```
std::array<A, 2> a{{ { 0, 1, 2 }, { 3, 4, 5 } }};
// buna eşdeğer ifade:
std::array<A, 2> a = {{ { 0, 1, 2 }, { 3, 4, 5 } }};
```

7. Kopya yapıcı (copy constructor) kullanarak;

```
std::array<A, 2> a{ 1, 2, 3 };
std::array<A, 2> a2(a);
// buna eşdeğer ifade:
std::array<A, 2> a2 = a;
```

8. Taşıma yapıcısı (move constructor) kullanarak;

```
std::array<A, 2> a = std::array<A, 2>{ 0, 1, 2, 3, 4, 5 };
```

Dizi elemanlarına erişim;

1. `at(pos)` fonksiyonu ile: `pos` konumundaki elemana sınır denetimiyle bir referans döndürür. `pos`, konteynerin aralığında değilse, `std::out_of_range` türünde bir istisna atılır.

```
#include <array>
int main()
{
    std::array<int, 3> arr;
    arr.at(0) = 3; //0. Elemana 2 atandı
    arr.at(1) = 4; //1. Elemana 2 atandı
    arr.at(2) = 6; //2. Elemana 2 atandı
    int a = arr.at(0); // a değişkenine 0. Dizi elemanı atandı
    int b = arr.at(1); // a değişkenine 1. Dizi elemanı atandı
    int c = arr.at(2); // a değişkenine 2. Dizi elemanı atandı
}
```

2. `operator[pos]` işleci ile: `pos` konumundaki elemana sınır denetimi olmadan bir başvuru döndürür. `pos`, konteynerin aralığında değilse, bir çalışma zamanı hatası (segmentation violation) oluşabilir. Bu yöntem, klasik dizilere eşdeğer eleman erişimi sağlar ve bu nedenle `at(pos)`'tan daha verimlidir.

```
#include <array>
int main()
{
    std::array<int, 3> arr;
    arr[0] = 3;
    arr[1] = 4;
    arr[2] = 6;
    int a = arr[0];
    int b = arr[1];
    int c = arr[2];
}
```

3. `std::get<pos>` ile: `std::array` sınıfının üyesi olmayan bu fonksiyon, sınır denetimi olmaksızın derleme zamanı sabit konumu `pos`'taki elemana bir referans döndürür. `pos`, konteynerin aralığında değilse, bir çalışma zamanı (segmentation violation) hatası oluşabilir.

```
#include <array>
int main()
{
```

```
    std::array<int, 3> arr;
    std::get<0>(arr) = 3;
    std::get<1>(arr) = 4;
    std::get<2>(arr) = 6;
    int a = std::get<0>(arr);
    int b = std::get<1>(arr);
    int c = std::get<2>(arr);
}
```

4. `front()` fonksiyonu ile: Konteynerdeki ilk elemana bir referans döndürür. Boş bir konteynerde `front()`'u çağırmak tanımsızdır.

```
#include <array>
int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };
    int a = arr.front();
}
```

5. `back()` fonksiyonu ile: Konteynerdeki son elemana bir referans döndürür. Boş bir konteynerde `back()`'u çağırmak tanımsızdır.

```
#include <array>
int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };
    int a = arr.back();
}
```

6. `data()` fonksiyonu ile: Eleman depolaması olarak hizmet eden altta yatan diziye gösterici döndürür.

```
#include <iostream>
#include <array>
void yaz(const int* p, std::size_t size){
    for (std::size_t i = 0; i < size; ++i)
        std::cout << p[i] << ' ';
}
int main (){
    std::array<int, 3> a { 0, 1, 2 };
    yaz(a.data(), 3);
}
```

Dizi elemanları üzerinde yineleme (iteration);

```
#include <iostream>
#include <array>
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    for (auto i : arr)
        std::cout << i << std::endl;
}
```

Dizi boyutunu kontrol etme;

```
#include <iostream>
#include <array>
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    std::cout << arr.size() << std::endl;
}
```

Dizi elemanlarını tek seferde değiştirme;

```
#include <iostream>
```

```
#include <array>
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    arr.fill(100);
}
```

std::vector

Bir vektör, otomatik olarak işlenen depolamaya sahip dinamik bir dizidir. Bir vektördeki elemanlara, bir dizideki elemanlar kadar verimli bir şekilde erişilebilir; avantajı, vektörlerin boyut olarak dinamik olarak değişebilmesidir. Depolama açısından vektör verileri (genellikle) dinamik olarak tahsis edilmiş belleğe yerleştirilir, bu nedenle bazı küçük ek yükler gerektirir; tersine C dizileri ve `std::array`, beyan edilen boyuta göre otomatik depolama kullanır ve bu nedenle herhangi bir ek yüke sahip değildir.

Vektör elemanlarına erişim;

Vektör elemanlarına `[]` işleci ve `at()` fonksiyonu ile erişebiliriz.

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> v{ 1, 2, 3 };
    // [] işleci:
    int a = v[1];
    v[1] = 4;
    // at() fonksiyonu:
    int b = v.at(2);
    v.at(2) = 5;
    //int c = v.at(3); // Hata 4. Eleman Yok!
    for (std::size_t i = 0; i < v.size(); ++i) {
        v[i] = 1;
    }
}
```

Ayrıca `front()` ve `back()` fonksiyonları ile de vektör elemanlarına erişilebilir;

```
std::vector<int> v{ 4, 5, 6 };
int a = v.front();
v.front() = 3;
int b = v.back(); // v.back() eşdeğerdir: v[v.size() - 1]
v.back() = 7;
```

Vektöre eleman ekleme `push_back()`, vektörden eleman çıkarma `pop_back()` ve boş olduğunu kontrol etme `empty()` fonksiyonları ile yapılabilir;

```
#include <iostream>
#include <vector>
int main ()
{
    std::vector<int> v;
    int toplam (0);
    for (int i=1;i<=10;i++) v.push_back(i); //vektör oluştutulup elemanlar ekleniyor
    while (!v.empty()) //vektörün boş olup olmadığı kontrol edilip ilerleniyor
    {
        toplam += v.back();
        v.pop_back(); //son elemanı vektörden çıkar
    }
    std::cout << "toplam: " << toplam << std::endl;
}
```

Vektörde `data()` yöntemi, `std::vector`'un elemanlarını dahili olarak depolamak için kullandığı ham belleğe bir gösterici döndürür. Bu gösterici, genellikle vektör verilerini C tarzı bir dizi bekleyen eski koda geçirirken kullanılır.

```
std::vector<int> v{ 1, 2, 3, 4 };
int* p = v.data();
*p = 4; // v[0]=4
++p;
*p = 3; // v[1]=3
p[1] = 2;
*(p + 2) = 1;
```

std::map, std::multimap ve std::pair

Haritalar, benzersiz anahtarlara sahip anahtar-değer çiftleri (key-value pair) içeren sıralı bir ilişkisel konteynerlerdir. Anahtarlar, `Compare()` karşılaştırma işlevi kullanılarak sıralanır. Arama, kaldırma ve ekleme işlemleri olan `search()`, `insert()`, `delete()` için logaritmik karmaşıklığa sahiptir.

- `std::map` veya `std::multimap`'ten herhangi birini kullanmak için `<map>` başlık dosyası projeye dahil edilmelidir.
- `std::map` ve `std::multimap` her ikisi de elemanlarının anahtarların artan sırasına göre sıralanmış halde tutar.
- `std::multimap` durumunda, aynı anahtarın değerleri için sıralama yapılmaz.
- `std::map` ve `std::multimap` arasındaki temel fark, `std::map`'in aynı anahtar için yinelenen değerlere izin vermemesidir.

Çiftler aşağıdaki gibi oluşturulup karşılaştırılabilir;

```
std::pair<int, int> p1 = std::make_pair(1, 2);
std::pair<int, int> p2 = std::make_pair(2, 2);
if (p1 == p2)
    std::cout << "eşit";
else
    std::cout << "eşit değil!"
```

Çiftler bir konteyner içindeyken de küçük işleci ile sıralanabilir;

```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
#include <string>
int main()
{
    std::vector<std::pair<int, std::string>> v = { {2, "Ali"}, {2, "Mustafa"}, {1, "İlhan"} };
    std::sort(v.begin(), v.end());
    for(const auto& p: v) {
        std::cout << "(" << p.first << "," << p.second << ") ";
    }
}
/*Program Çıktısı:
(1,İlhan) (2,Ali) (2,Mustafa)
*/
```

Haritalarda elemanlar anahtar değer ikilisidir ve aşağıdaki şekilde yineleyiciler ile erişilir;

```
#include <iostream>
#include <map>
int main()
{
    std::map < int, std::string > siralama {
                                                std::make_pair(2, "ahmet"),
```

```
                                        std::make_pair(1, "ali") };

    siralama[1]="mustafa";
    std::cout << siralama.at(2) << std::endl; // ahmet
    auto it = siralama.begin();
    std::cout << it->first << " : " << it->second << std::endl; // "1 : mustafa"
    it++;
    std::cout << it->first << " : " << it->second << std::endl; // "2 : ahmet"

    std::map < std::string , int> notlar {
                                        std::make_pair("ahmet",100),
                                        std::make_pair("ali",90),
                                        std::make_pair("mustafa",85)
                                        };
    notlar["mustafa"]=95;
    std::cout << notlar.at("ali") << std::endl; // 90
    auto it2 = notlar.rbegin(); //tersten
    std::cout << it2->first << " : " << it2->second << std::endl; // "mustafa: 95"
    it2++;
    std::cout << it2->first << " : " << it2->second << std::endl; // "ali: 90"
}
```

Aşağıdaki örnekte ise `std::multimap` örneği de verilmiştir;

```
#include <iostream>
#include <map>
int main()
{
    std::multimap < int, std::string > mmp {
                                        std::make_pair(2, "ahmet"),
                                        std::make_pair(1, "ali"),
                                        std::make_pair(2, "mustafa")

    };
    auto it = mmp.begin();
    std::cout << it->first << " : " << it->second << std::endl; //"1 : ali"
    it++;
    std::cout << it->first << " : " << it->second << std::endl; //"2 : ahmet"
    it++;
    std::cout << it->first << " : " << it->second << std::endl; //"2 : mustafa"

    std::map < int, std::string > mp {
                                    std::make_pair(2, "ahmet"),
                                    std::make_pair(1, "ali"),
                                    std::make_pair(2, "mustafa")
                                    // Aynı anathardan var!
    };
    auto it2 = mp.rbegin(); //tersten
    std::cout << it2->first << " : " << it2->second << std::endl; //"2 : ahmet"
    it2++;
    std::cout << it2->first << " : " << it2->second << std::endl; //"1 : ali"
}
```

Haritaya anahtar değerler aşağıdaki şekilde eklenebilir;

```
std::map< std::string, size_t > meyvesayisi;
meyvesayisi.insert({"üzüm", 20});
meyvesayisi.insert(make_pair("portakal", 30));
meyvesayisi.insert(pair<std::string, size_t>("muz", 40));
meyvesayisi.insert(map<std::string, size_t>::value_type("çilek", 50));
```

`insert()` fonksiyonu bir yineleyici ve bir bool değerden oluşan bir çift döndürür. Döndürülen değer true ise haritaya çift eklenmiştir. Değilse eklenmeye çalışılan çift zaten haritada vardır.

```
auto success = meyvesayisi.insert({"üzüm", 20});
if (!success.second) { // zaten üzüm var
    success.first->second += 20; // üzüm müktarı değiştirlebilir
}
meyvesayisi["karpuz"]=10; //haridata olmayan karpuz eklenmişir.
meyvesayisi.insert({{"şeftali", 1}, {"kayısı", 1}, {"limon", 1}, {"mango", 7}});
```

Konteynerlere başlatma listeleriyle de eleman eklenebilir;

```
std::map< std::string, size_t > meyvelistesi{ {"zeytin", 0}, {"elma", 0}, {"kavun", 0}};
meyvesayisi.insert(meyvelistesi.begin(), meyvelistesi.end());
```

Haritaya eklenecek çiftler yukarıdaki örneklerde olduğu gibi `make_pair()` ve `emplace()` yöntemleriyle hazırlanabilir;

```
meyvesayisi.emplace("Harnup", 200);
```

Haritalarda bir anahtarın ilk oluşumunun yineleyicisini almak için `find()` fonksiyonu kullanılabilir. Anahtar mevcut değilse `end()` döndürür;

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
auto it = mmp.find(6);
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl; // 6, 5
else
    std::cout << "Aranan Anahtar Bulunamadı!" << std::endl;
it = mmp.find(66);
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl;
else
    std::cout << "Aranan anahtar bulunamadı!" << std::endl;
```

Haritalarda bir anahtarın mevcut olduğunu bulmanın bir başka yolu da `count()` fonksiyonunu kullanmaktır. Bu fonksiyon, belirli bir anahtarla ilişkili değer sayısını sayar;

```
std::map< int , int > mp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
if(mp.count(3) > 0)
    std::cout << "Anahtar bulundu!" << std::endl;
else
    std::cout << "Anahtar Bulunamadı!" << std::endl;
```

Haritaları tanımlama ve ilk değer verme;

```
std::multimap < int, std::string > mmp { std::make_pair(2, "C Lang"),
                                         std::make_pair(1, "CPP Lang"),
                                         std::make_pair(2, "Java Lang") };

std::map < int, std::string > mp { std::make_pair(2, "C lang"),
                                   std::make_pair(1, "CPP Lang"),
                                   std::make_pair(2, "Java Lang") // Eklenmeyecek!
                                 };
// Yineleyici kullanarak;
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {6, 8}, {3, 4}, {6, 7} };
auto it = mmp.begin();
std::advance(it,3); //baştan üçüncü çifte git {6, 5}
std::map< int, int > mp(it, mmp.end()); // {6, 5}, {8, 9}, {6, 8}, {3, 4}, {6, 7}

//std::pair dizisi kullanarak;
std::pair< int, int > arr[10];
arr[0] = {1, 3};
arr[1] = {1, 5};
```

```
arr[2] = {2, 5};
arr[3] = {0, 1};
std::map< int, int > mp(arr,arr+4); //{0 , 1}, {1, 3}, {2, 5}

//std::pair vektörü kullanarak
std::vector< std::pair<int, int> > v{ {1, 5}, {5, 1}, {3, 6}, {3, 2} };
std::multimap< int, int > mp(v.begin(), v.end()); // {1, 5}, {3, 6}, {3, 2}, {5, 1}
```

Haritaların boş olup olmamasına bağlı olarak true veya false döndüren bir üye `empty()` fonksiyonuna sahiptir. Ayrıca sahip olunan eleman sayısı da `size()` üye fonksiyonuna da sahiptir.

```
std::map<std::string , int> rank {{"facebook.com", 1} ,{"google.com", 2}, {"youtube.com", 3}};
if(!rank.empty()){
    std::cout << "Hatitadadaki eleman sayısı: " << rank.size() << std::endl;
}
else{
    std::cout << "Harita Boş!" << std::endl;
}
```

Bir de karma harita (hash map) vardır. Düzenlenmemiş harita (unordered map) olarak da adlandırılan bu haritalar, normal bir haritaya benzer şekilde anahtar-değer çiftlerini depolar. Ancak elemanları anahtara göre sıralamaz. Bunun yerine, anahtar için bir karma (hash) değer, ihtiyaç duyulan anahtar-değer çiftlerine hızlı bir şekilde erişmek için kullanılır;

```
#include <string>
#include <unordered_map>
std::unordered_map<std::string, size_t> meyveSayisi;
```

Haritadaki elemanlar aşağıdaki şekilde silinebilir;

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
mmp.clear(); //hepsini sil

auto it = mmp.begin();
std::advance(it,3); // baştan üçüncü elemana git {6, 5}
mmp.erase(it); // {1, 2}, {3, 4}, {8, 9}, {3, 4}, {6, 7}
```

Belli anahtarlara sahip elemanlar `erase()` üye fonksiyonu ile silinebilir;

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
mmp.erase(6); // {1, 2}, {3, 4}, {3, 4}, {8, 9}
```

std::list ve std::forwardlist

Konteynerin herhangi bir yerinden öğelerin hızlı bir şekilde eklenmesini ve kaldırılmasını istiyorsak `std::forward_list` bir konteynerdir. Hızlı rastgele erişim desteklenmez. Bu konteyner, `std::list` ile karşılaştırıldığında, çift yönlü yineleme gerekmediğinde daha fazla alan tasarrufu sağlar.

```
#include <forward_list>
#include <list>
#include <string>
#include <iostream>
template<typename T> std::ostream& operator<<(std::ostream& s, const std::forward_list<T>& v)
{
    s << "Forward List";
    s.put('[');
    char comma[3] = {'\0', ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
```

```
}
template<typename T> std::ostream& operator<<(std::ostream& s, const std::list<T>& v) {
    s << "List";
    s.put('[');
    char comma[3] = {'\0', ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
}
int main() {
std::forward_list<std::string> words1 {"İlhan", "Ahmet", "Mehmet", "Mustafa", "Abdullah"};
std::cout << "İsimler1: " << words1 << '\n';

std::list<std::string> words2 {"İlhan", "Ahmet", "Mehmet", "Mustafa", "Abdullah"};
std::cout << "İsimler2: " << words2 << '\n';
}
/* Program Çıktısı:
İsimler1: Forward List[İlhan, Ahmet, Mehmet, Mustafa, Abdullah]
İsimler2: List[İlhan, Ahmet, Mehmet, Mustafa, Abdullah]
*/
```

std::set ve std::multiset

Küme (`std::set`), elemanları sıralanmış ve benzersiz olan bir tür konteynerdir. Ancak `std::multiset` birden fazla eleman aynı elemana sahip olabilir. Lamda ifadeleri C++ 14 ile hayatımıza girmiştir.

```
#include <iostream>
#include <set>
#include <stdlib.h>
struct custom_compare final
{
    bool operator() (const std::string& left, const std::string& right) const
    {
        int nLeft = atoi(left.c_str());
        int nRight = atoi(right.c_str());
        return nLeft < nRight;
    }
};
int main ()
{
    std::set<std::string> sut({"1", "2", "5", "23", "6", "290"});
    std::cout << "Ön tanımlı sıralama std::set<std::string> :" << std::endl;
    for (auto &&data: sut)
        std::cout << data << ", ";
    std::set<std::string, custom_compare> sut_custom(
                                          {"1", "2", "5", "23", "6", "290"},
                                          custom_compare{});
    std::cout << std::endl << "Özel Sıralama :" << std::endl;
    for (auto &&data : sut_custom)
        std::cout << data << ", ";
    auto compare_via_lamda = [](auto &&lhs, auto &&rhs){ return lhs > rhs; };
    using set_via_lamda = std::set<std::string, decltype(compare_via_lamda)>;
    set_via_lamda sut_reverse_via_lamda({"1", "2", "5", "23", "6", "290"},
                                          compare_via_lamda);
    std::cout << std::endl << "Lamda Sıralaması :" << std::endl;
    for (auto &&data : sut_reverse_via_lamda)
        std::cout << data << ", ";
    return 0;
}
```

```
/*g++ main.cpp -std=C++14 ile derlenen Program Çıktısı:
Ön tanımlı sıralama std::set<std::string> :
1, 2, 23, 290, 5, 6,
Özel Sıralama :
1, 2, 5, 6, 23, 290,
Lamda Sıralaması :
6, 5, 290, 23, 2, 1,
*/
```

std::optional

`std:optional` olarak tanımlananlar isteğe bağlı/belki tipleri olarak da bilinir. İçeriği mevcut olabilen veya olmayabilen bir veri tipi temsil etmek için kullanılır.

C++17'den önce, `nullptr` değerine sahip göstericilere sahip olmak genellikle bir değerin yokluğunu temsil ediyordu. Bu, dinamik olarak tahsis edilmiş ve zaten göstericiler tarafından yönetilen büyük nesneler için iyi bir çözümdür. Ancak, bu çözüm nadiren göstericiler tarafından dinamik olarak tahsis edilen veya yönetilen int gibi küçük veya ilkel türler için iyi çalışmaz. `std::optional` bu yaygın soruna uygulanabilir bir çözüm sağlar.

```
#include <iostream>
#include <optional>
#include <string>
struct Hayvan {
    std::string adi;
};

struct Person {
    std::string adi;
    std::optional<Hayvan> evcilhayvan;
};
int main() {
    Person person;
    person.adi = "Ali";
    if (person.evcilhayvan) {
        std::cout << person.adi << " Adlı Kişinin " << person.evcilhayvan->adi
                  << "Evcil Hayvanı Var" << std::endl;
    }
    else {
        std::cout << person.adi << " adlı kişinin evcil hayvanı yok!" << std::endl;
    }
}
/* g++ -std=c++17 main.cpp olarak derlenen Program Çıktısı:
Ali adlı kişinin evcil hayvanı yok!
*/
```

Fonksiyondan geri dönüş değeri olarak da `std::optional` kullanılabilir;

```
std::optional<float> divide(float a, float b) {
    if (b!=0.f) return a/b;
    return {};
}
```

`std:optional` olarak tanımlı bir değeri işlemek için `value_or` kullanılır;

```
void print_name( std::ostream& os, std::optional<std::string> const& name ) {
    std::cout "Name is: " << name.value_or("<name missing>") << '\n';
}
```

std::function ve std::bind

Herhangi bir fonksiyona kılıf çekmek için kullanılır. C tipi fonksiyon göstericisi kullanmak yerine C++ dilinde `std::function` kullanılır.

```
#include <iostream>
#include <functional>
std::function<void(int , const std::string&)> myFuncObj;
void theFunc(int i, const std::string& s) {
    std::cout << s << ": " << i << std::endl;
}
int main(int argc, char *argv[]) {
    myFuncObj = theFunc;
    myFuncObj(10, "hello world");
}
```

Argümanlarla bir fonksiyonu geri çağırmamız gereken bir durumda `std::bind` ile kullanılan `std::function` aşağıda gösterildiği gibi çok güçlü bir tasarım yapısı verir.

```
#include <iostream>
#include <functional>
class Ocak {
public:
    std::function<void(int, const std::string&)> YumurtaKaynadi = nullptr;
    void YumurtaKaynat() {
        if (YumurtaKaynadi) {
            YumurtaKaynadi(10, "Yumurta Pişti");
        }
    }
};
class Kisi {
public:
    Kisi() {
        auto yumurtaPisinceHaberVer = std::bind(&Kisi::eventHandler, this,
                                                std::placeholders::_1,
                                                std::placeholders::_2);
        ocakNesnesi.YumurtaKaynadi = yumurtaPisinceHaberVer;
    }
    void eventHandler(int i, const std::string& s) {
        std::cout << i << " dakikada "<< s <<std::endl;
    }
    void OcaktaYumurtaKaynat() {
        ocakNesnesi.YumurtaKaynat();
    }
    Ocak ocakNesnesi;
};
int main(int argc, char *argv[]) {
    Kisi ali;
    ali.OcaktaYumurtaKaynat();
}
/*Program Çıktısı:
10 dakikada Yumurta Pişti
*/
```

Aşağıda `std:function` ile çeşitli fonksiyonların çağrılmasını gösteren bir program verilmiştir;

```
#include <iostream>
#include <functional>
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
```

```
using namespace std::placeholders;
double c_function(int x, float y, double z) {
    double res = x + y + z;
    std::cout << "c_function çağrıldı: "
    << x << "+" << y << "+" << z
    << "=" << res
    << std::endl;
    return res;
}
struct yapi
{
    double yapi_function(int x, float y, double z) {
        double res = x + y + z;
        std::cout << "yapi::yapi_function çağrıldı: "
        << x << "+" << y << "+" << z
        << "=" << res
        << std::endl;
        return res;
    }
    double farkli_yapi_function(int x, double z, float y, long xx) {
        double res = x + y + z + xx;
        std::cout << "farkli_yapi_function çağrıldı: "
        << x << "+" << y << "+" << z << "+" << xx
        << "=" << res
        << std::endl;
        return res;
    }
    double operator()(int x, float y, double z) {
        double res = x + y + z;
        std::cout << "yapi::operator() çağrıldı: "
        << x << "+" << y << "+" << z
        << "=" << res
        << std::endl;
        return res;
    }
};
int main(void) {
    using function_type = std::function<double(int, float, double)>;
    yapi ornekyapi;
    std::vector<function_type> bindings;

    function_type var1 = c_function;
    bindings.push_back(var1);
    function_type var2 = std::bind(&yapi::yapi_function, ornekyapi, _1, _2, _3);
    bindings.push_back(var2);
    function_type var3 = std::bind(&yapi::farkli_yapi_function, ornekyapi, _1, _3, _2, 11l);
    bindings.push_back(var3);
    function_type var4 = ornekyapi; //operator()
    bindings.push_back(var4);

    function_type var5 = [](int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "lamda çağrıldı: "
        << x << "+" << y << "+" << z
        << "=" << res
        << std::endl;
        return res;
    };
    bindings.push_back(var5);
```

```
    std::cout << "Vektöre saklana fonksiyonlar test ediliyor: x = 1, y = 2, z = 3"
    << std::endl;
    for (auto f : bindings)
        f(1, 2, 3);
}
```

std::tupple

Farklı tiplerdeki değerler de dahil olmak üzere herhangi bir sayıda değeri tek bir dönüş nesnesine toplamak için `std::tuple` kullanılır.

```
std::tuple<int, int, int, int> foo(int a, int b) { // or auto (C++14)
    return std::make_tuple(a + b, a - b, a * b, a / b);
           /*dört farklı int değeri geri döndürülüyor */
}
```

C++ 2017 uyarlamasından sonra tırnaklı parantez olan başlatma listesi kullanılır;

```
std::tuple<int, int, int, int> foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}
```

Döndürülen `tuple`'dan değerleri almak zahmetlidir ve `std::get` şablon fonksiyonunun kullanımını gerektirir:

```
auto mrvs = foo(5, 12);
auto add = std::get<0>(mrvs);
auto sub = std::get<1>(mrvs);
auto mul = std::get<2>(mrvs);
auto div = std::get<3>(mrvs);
```

Eğer tipler fonksiyon dönmeden önce bildirilebiliyorsa, o zaman `std::tie` bir `tuple`'ı mevcut değişkenlere açmak için kullanılabilir:

```
int add, sub, mul, div;
std::tie(add, sub, mul, div) = foo(5, 12);
```

`std::tie` ile döndürülen değerlerden birine ihtiyaç duyulmuyorsa `std::ignore` kullanılabilir:

```
std::tie(add, sub, std::ignore, div) = foo(5, 12);
```

Yapılandırılmış bağlamalar `std::tie`'dan kaçınmak için kullanılabilir:

```
auto [add, sub, mul, div] = foo(5,12);
```

DİZGİLER

Dizgi Tanımlama

Dizgiler (string) metinler olup bellekte son karakteri sıfır olan karakter dizileri olarak tutulurlar. Dizgiler, boş bir karakter `'\0'` ile sonlanan karakter dizisi olarak saklanır. Metinleri tutan değişkenler uzantısı olduğu C dilinin belirlediği şekilde tanımlanabilir;

```
    char metin[]="Merhaba";  /* değişkenine metin değişmezi(string literal) ile
                                ilk değer verildi */
    char metin2[]={'M','e','r','h','a','b','a','\0'};
    char* metinGostericisi="Merhaba";
```

Aşağıda C dilinden geldiği şekliyle metinlerin tanımlanıp kullandığı bir örnek program verilmiştir;

```
#include <iostream>
using namespace std;
int main() {
    char metin[] = "Merhaba";
    cout << metin << endl;
}
```

C dilinde karakter dizileri olarak kullanılan dizgiler (string) yerine C++ dilinde konteyner şablonlarından biri olan `std::string` kullanılır. C++ dilinde dizgiler, karakter dizilerini temsil eden nesnelerdir. Standart dizgi sınıfı, metin ve diğer karakter dizgileriyle çalışır ve C dilindeki açık karakter dizileri yerine basit, güvenli ve çok yönlü bir alternatiftir. C++ dizgi sınıfı, `std` isim uzayının bir parçasıdır ve 1998'de standartlaştırılmıştır.

```
    string metin=("Merhaba");
    string metin2="Merhaba";
    string metin3; metin3="Merhaba";
```

Yukarıda verilen program `std::string` ile aşağıdaki gibi yazılabilir;

```
#include <iostream>
int main() {
    std::string str("Merhaba");
    std::cout << str;
}
```

std::string

Dizgiler belirlenen bir ayraç metni ile belirteç (token) denilen küçük parçalara `strtok()` fonksiyonu ile ayrılabilir;

```
std::string str{ "Pijamalı hasta yağız şoföre çabucak güvendi." };

vector<std::string> tokens;
for (auto i = strtok(&str[0], " "); i != NULL; i = strtok(NULL, " "))
tokens.push_back(i);
```

Bir `std::string` verilerine `const char*` olarak erişim sağlamak için `c_str()` üye yöntemi kullanılır;

```
//Göstericiler str nesnesinin altında yatan karakter dizisini gösterir;
std::string str("This is a string.");
const char* cstr = str.c_str(); // cstr göstericisi bu metni gösterir: "This is a string.\0"
const char* data = str.data();  // data göstericisi bu metni gösterir: "This is a string.\0"

//Göstericilerin str nesnesinden ömür boyu bağını çözmek için kopyasını oluşturun
std::string str("This is a string.");
std::unique_ptr<char []> cstr = std::make_unique<char[]>(str.size() + 1);
```

```
// Yukarıdaki satırın eşdeğeri: char* cstr_unsafe = new char[str.size() + 1];
std::copy(str.data(), str.data() + str.size(), cstr);
cstr[str.size()] = '\0'; // NULL karakter sonuna eklenir.
// delete[] cstr_unsafe;
std::cout << cstr.get();
```

C++17 ile birlikte `std:string` altında yatan `const char` üzerinden işlem yapmak yerine `std::string_view` üzerinden işlemler yapılır. Değiştirilemeyen dizgi verileri gerektiren işlevler için üstünlük sağlar;

```
//1)Alt metin elde etmek için kopya oluşturarak;
std::string str = "Çoook uzun biiiir metiiiiiin";
//'string::substr' returns a new string (expensive if the string is long)
std::cout << str.substr(15, 10) << '\n';

//2)Hiçbir kopya oluşturmadan;
std::string_view view = str;
// string_view::substr returns a new string_view
std::cout << view.substr(15, 10) << '\n';
```

`std::string`, `char` tipindeki elemanlarla oluşturulmuştur. `std::wstring` ise uluslararası karakterleri temsil eden `wchar_t` tipindeki elemanlarla oluşturulmuştur. Bunlar arasında çeviri yapılabilir;

```
#include <iostream>
#include <string>
#include <codecvt>
#include <locale>
int main() {
    std::locale::global(std::locale(""));
    std::wcout.imbue(std::locale());
    setlocale(LC_ALL, "Turkish");
     const wchar_t message_turkish[] = L"ıİüÜğĞiİşŞçÇöÖ"; //Türkçe Karakterler
    //std::wcout << message_turkish << std::endl;

    std::string input_str = "İlhan-this is a -string-";
    std::wstring input_wstr = L"İlhan-this is a -wide- string";
    // conversion
    std::wstring str_turned_to_wstr =
    std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes(input_str);
    std::wcout << str_turned_to_wstr << std::endl;

    std::string wstr_turned_to_str =
    std::wstring_convert<std::codecvt_utf8<wchar_t>>().to_bytes(input_wstr);
    std::cout << wstr_turned_to_str << std::endl;

    wstr_turned_to_str =
    std::wstring_convert<std::codecvt_utf8<wchar_t>>().to_bytes(message_turkish);
    std::cout << wstr_turned_to_str << std::endl;
}
```

İki `std::string`, `==`, `!=`, `<`, `<=`, `>` ve `>=` işleçleri kullanarak sözlükteki olarak karşılaştırılabilir;

```
#include <iostream>
#include <string>
#include <locale>
int main() {
    std::locale::global(std::locale(""));
    std::wcout.imbue(std::locale());
    setlocale(LC_ALL, "Turkish");
    std::string sozcuk1 = "İlhan";
    std::string sozcuk2 = "İlhan";
    if (sozcuk2==sozcuk1)
        std::cout << sozcuk1 << "=" << sozcuk2 << std::endl;
    std::wstring wsozcuk1 = L"İlhan";
```

```
    std::wstring wsozcuk2 = L"İlhan";
    if (wsozcuk1==wsozcuk2)
        std::wcout << wsozcuk1 << "=" << wsozcuk2 << std::endl;
}
```

Dizgenin bir bölümü bir başka metin ile `replace()` üye yöntemi ile değiştirilebilir;

```
std::string str = "Hello foo, bar and world!";
std::string alternate = "Hello foobar";
//1)
str.replace(6, 3, "bar"); //"Hello bar, bar and world!"
//2)
str.replace(str.begin() + 6, str.end(), "nobody!"); //"Hello nobody!"
//3)
str.replace(19, 5, alternate, 6, 6); //"Hello foo, bar and foobar!"
Version ≥ C++14
//4)
str.replace(19, 5, alternate, 6); //"Hello foo, bar and foobar!"
//5)
str.replace(str.begin(), str.begin() + 5, str.begin() + 6, str.begin() + 9);
//"foo foo, bar and world!"
//6)
str.replace(0, 5, 3, 'z'); //"zzz foo, bar and world!"
//7)
str.replace(str.begin() + 6, str.begin() + 9, 3, 'x'); //"Hello xxx, bar and world!"
Version ≥ C++11
//8)
str.replace(str.begin(), str.begin() + 5, { 'x', 'y', 'z' }); //"xyz foo, bar and world!"
```

Dizgenin bir kısmı `substr()` üye yöntemi ile alınabilir;

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(11); // "bar and world!"
```

Dizgenin altında yatan karakter dizisine `[]` işleci ile `at()`, `front()` ve `back()` üye fonksiyonları ile erişilebilir;

```
std::string str("Hello world!");
char c1 = str[6]; // 'w'
char c2 = str.at(7); // 'o'
char c3 = str.front(); // 'H'
char c4 = str.back(); // '!'
```

Dizginin altında yatan karakter dizisi üzerinde yineleme yapılabilir;

```
std::string str = "Hello World!";
//1) Foreach
for (auto c : str)
    std::cout << c;
//2) Geleneksel
for (std::size_t i = 0; i < str.length(); ++i)
    std::cout << str[i];
```

Metinleri diğer veri tiplerine de dönüştürebiliriz;

```
std::string ten = "10";
int num1 = std::stoi(ten);
long num2 = std::stol(ten);
long long num3 = std::stoll(ten);
float num4 = std::stof(ten);
double num5 = std::stod(ten);
long double num6 = std::stold(ten);
```

Metinleri dönüştürürken sayı tabanları da kullanılabilir;

```
std::string ten = "10";
std::string ten_octal = "12";
std::string ten_hex = "0xA";
int num1 = std::stoi(ten, 0, 2); // Returns 2
int num2 = std::stoi(ten_octal, 0, 8); // Returns 10
long num3 = std::stol(ten_hex, 0, 16); // Returns 10
long num4 = std::stol(ten_hex); // Returns 0
long num5 = std::stol(ten_hex, 0, 0); // Returns 10 as it detects the leading 0x
```

Metinler `+` ve `+=` işleçleri birleştirilebilir;

```
std::string hello = "Hello";
std::string world = "world";
std::string helloworld = hello + world; // "Helloworld"
hello += world; // "Helloworld"
```

Metinler birinin sonuna diğeri gelecek şekilde `append()` üye fonksiyonu ile birleştirilebilir;

```
std::string app = "test and ";
app.append("test"); // "test and test"
```

Bir karakteri veya başka bir dizeyi bulmak için `std::string::find()` kullanabilirsiniz. İlk eşleşmenin ilk karakterinin konumunu döndürür. Eşleşme bulunamazsa, `std::string::npos` döndürür. `find_first_of()` ile karakterlerin ilk oluşumu bulunur, `find_first_not_of()` ile karakterlerin ilk yokluğunu bulunur, `find_last_of()` ile karakterlerin son oluşumu bulunur, `find_last_not_of()` ile karakterlerin son yokluğu bulunur;

```
std::string str = "Hello world!";
auto it = str.find("world");
if (it != std::string::npos)
    std::cout << "Found at position: " << it << '\n';
else
    std::cout << "Not found!\n";
```

Klavyeden Metin Okuma

Bir girdi nesnesinden dizgi okumak için en yaygın yol, C++ dininde akıştan veri çıkarma işleci (stream extraction operator) olan `>>` ile `std::cin` nesnesini kullanmaktır.

```
#include <iostream>
using namespace std;
int main() {
    string metin;
    cout << "Bir metin Giriniz: ";
    cin >> metin;
    cout<< "Girilen Metin: " << metin << endl;
}
/* Program Çalıştığında:
Bir metin Giriniz: Ilhan Ozkan
Girilen Metin: Ilhan

...Program finished with exit code 0
*/
```

Program incelendiğinde girilen ilk sözcük metin değişkenine atanmıştır. Eğer girilen tüm satırı bu metne aktarmak istersek `<string>` başlık dosyasındaki `getline()` metodunu kullanmalıyız;

```
#include <iostream>
using namespace std;
int main() {
    string metin;
    cout << "Bir metin Giriniz: ";
```

```
    getline(cin, metin);
    cout<< "Girilen Metin: " << metin << endl;
}
/* Program Çalıştığında:
Bir metin Giriniz: Ilhan Ozkan
Girilen Metin: Ilhan Ozkan

...Program finished with exit code 0
*/
```

Uluslararası karakterlerle çalışılan aşağıdaki örnek verilebilir;

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    const wchar_t message_turkish[] = L"ıIüÜğĞiİşŞçÇöÖ"; //Türkçe "Diğer Karakterler
    std::locale::global(std::locale(""));
    std::wcout.imbue(std::locale());
    setlocale(LC_ALL, "Turkish");
    wcout << message_turkish << endl;

    wchar_t message[100];
    wcout << "Tükçe Bir Metin Giriniz:";
    wcin.getline(message,100);
    wcout << "Girilen metin:" << message <<endl;

    wstring message2;
    wcout << "Tükçe Bir Başka Giriniz:";
    getline(wcin, message2);
    wcout << "Girilen metin:" << message2;
}
```

Parametre Olarak Dizgiler

Bir diziyi bir fonksiyona geçirdiğimiz gibi, C++ dilinde de dizgiler de karakter dizileri olarak fonksiyonlara geçirilebilir.

```
#include <iostream>
using namespace std;
void metniKonsolaYaz(string pMetin) {
    cout << "Parametre olarak geçirilen metin: " << pMetin << endl;
    return;
}
int main() {
    string metin = "Merhaba Ilhan Ozkan.";
    metniKonsolaYaz(metin);
}
/* Program Çalıştığında:
Parametre olarak geçirilen metin: Merhaba Ilhan Ozkan.

...Program finished with exit code 0
*/
```

Karakter Göstericileri

C++ dilinde göstericiler, adreslerin sembolik gösterimleridir. Göstericileri kullanarak dizginin ilk karakterini, aslında karakter dizisi olarak tutulan metnin ilk karakterini başlangıç adresini elde edebiliriz. Aşağıda gösterildiği gibi, verilen dizgiye göstericiler aracılığıyla erişilebilir ve yazdırılabiliriz;

```
#include <iostream>
using namespace std;
int main() {

    string s = "Merhaba Ilhan";
    char* p = &s[0];
    while (*p != '\0') {
        cout << *p;
        p++;
    }
    cout << endl;
}
/* Program çalıştığında;
Merhaba Ilhan

...Program finished with exit code 0
*/
```

Dizgiler ve Karakter Dizisi

Bir dizgi ile bir karakter dizisi arasındaki temel fark, dizgilerin `mutable` olmayan değiştirilemez nesne (immutable object), karakter dizilerinin ise boyutunun değiştirilemez olmasıdır. Aralarındaki farklar aşağıdaki tabloda verilmiştir;

Dizgi	Karakter Dizisi
Dizgiler, dizgi akışları (stream) olarak temsil edilebilen nesneleri tanımlar.	`'\0'` olarak kodlanan `NULL` karakteri, karakterlerden oluşan bir karakter dizisini sonlandırır.
Dizgilerde dizi bozulması meydana gelmez çünkü dizeler nesneler olarak temsil edilir.	Bir dizinin tip ve boyut kaybına dizinin bozulması (decay of array) denir. Bu durum genellikle diziyi değer veya gösterici ile fonksiyona geçirdiğimizde ortaya çıkar. Karakter dizisinde böyle bir durum ortaya çıkabilir.
Bir dizgi sınıfı, dizgileri işlemek için çok sayıda fonksiyon sağlar.	Karakter dizileri, dizeleri işlemek için C++ dilinde yerleşik işlevler sunmaz.
Dizgilere bellek dinamik olarak tahsis edilir.	Karakter dizisine boyutu kadar veri bellekte (data segment) veya yığın bellekte (stack segment) yer statik olarak tahsis edilir.

Tablo 20. Dizgi ile Karakter Dizisi Karşılaştırması

Dizgi Fonksiyonları

C++ dilinde dizgileri kopyalamak ve birleştirmek için `strcpy()` ve `strcat()` işlevleri gibi dize işleme için kullanılan bazı yerleşik işlevler sağlar. Bunlardan bazıları şunlardır

Fonksiyon	Açıklama
`length()`	Metnin karakter uzunluğunu döndürür.
`swap()`	İki metni yer değiştirmek için kullanılır.
`size()`	Metnin boyutunu bulmak için kullanılır. Metne bellekte kaç karakter yer ayrılmış onu döner.
`resize()`	Metnin uzunluğunu belirtilen karakter sayısına kadar yeniden boyutlandırmak için kullanılır.
`find()`	Metnin içinde parametre olarak geçirilen bir başka metni arar.
`push_back()`	Parametre olarak geçirilen karakteri metnin sonuna eklemek kullanılır.
`pop_back()`	Metinden son karakteri çıkarmak için kullanılır.
`clear()`	Metnin tüm karakterlerini silmek için kullanılır.
`strncmp()`	Metin ile parametre olarak geçirilen bir başka metnin en fazla ilk num kadar karakterini karşılaştırır.
`strncpy()`	strcpy() fonksiyonuna benzerdir, ancak en fazla n bayt kopyalanır.
`strrchr()`	Metindeki bir karakterin son bulunduğu yeri bulur.

`strcat()`	Kaynak metnin bir kopyasını hedef metnin sonuna ekler.
`find()`	Bir metnin içerisinde belirli bir alt metni aramak için kullanılır ve alt metnin ilk karakterinin konumunu döndürür.
`replace()`	first-last aralığındaki eski değere eşit olan her bir elemanı yeni değerle değiştirmek için kullanılır
`substr()`	Verilen bir metinden bir alt metin oluşturmak için kullanılır.
`compare()`	İki metni karşılaştırmak ve sonucu tam sayı biçiminde döndürmek için kullanılır.
`erase()`	Bu fonksiyon bir metnin belli bir kısmını silmek için kullanılır.
`rfind()`	Metnin son bulunduğu konumu bulmak için kullanılır.
`capacity()`	Bu fonksiyon, derleyici tarafından dizgiye tahsis edilen kapasiteyi döndürür.
`shrink_to_fit()`	Bu fonksiyon dizginin kapasitesini en az (minimum) olacak şekilde azaltır.

Tablo 21. Çok Kullanılan Dizgi Fonksiyonları

Bunların yanında metni tutan dizgi değişkeni kullanılarak metin karakterleri üzerinde işlem yapmak için yineleme (itertation) yöntemlerini kullanabiliriz. Yineleme nesneleri diziler gibi birden çok aynı nesneyi tutan veri yapılarında elemanlar üzerinde hareket etmeyi sağlayana nesnelerdir.

Fonksiyon	Açıklama
`begin()`	Bu fonksiyon, dizginin başlangıcına işaret eden bir yineleyici (`iterator`) nesne döndürür.
`end()`	Bu fonksiyon, dizginin sonunu işaret eden bir yineleyici nesne döndürür.
`rbegin()`	Bu fonksiyon, dizginin başlangıcına işaret eden ters yineleyici (`reverse_iterator`) nesne döndürür.
`rend ()`	Bu fonksiyon, dizginin sonunu işaret eden ters yineleyici nesne döndürür.
`cbegin ()`	Bu fonksiyon, dizginin başlangıcına işaret eden bir sabit yineleyici (`const_iterator`) döndürür.
`cend()`	Bu fonksiyon dizginin sonunu işaret eden bir `const_iterator` döndürür.
`crbegin()`	Bu fonksiyon dizginin sonunu işaret eden bir sabit bir ters yineleyici (`const_reverse_iterator`) döndürür.
`crend()`	Bu fonksiyon, dizginin başlangıcına işaret eden bir yineleyici (`const_reverse_iterator`) döndürür.

Tablo 22. Dizgi Metni Yineleme Fonksiyonları

Yineleme fonksiyonlarına ilişkin örnek aşağıda verilmiştir;

```
#include <iostream>
using namespace std;

int main() {
    string::iterator itr; //string yineleme nesnesi
    string::reverse_iterator rit; //string ters yineleme nesnesi

    string metin = "Merhana Ilhan.";

    itr = metin.begin();
    cout << "Yineleyicinin işaret ettiği karakter: " << *itr<< endl;
    itr = metin.end() - 1;
    cout << "Yineleyicinin işaret ettiği son metin karakteri: " << *itr << endl;

    rit = metin.rbegin();
    cout << "Ters yineleyicinin işaret ettiği karakter:: " << *rit << endl;
    rit = metin.rend() - 1;
    cout << "Ters yineleyicinin işaret ettiği on metin karakteri:: " << *rit << endl;
}
/*Program çalıştığında:
Yineleyicinin işaret ettiği karakter: M
Yineleyicinin işaret ettiği son metin karakteri: .
```

```
Ters yineleyicinin işaret ettiği karakter:: .
Ters yineleyicinin işaret ettiği on metin karakteri:: M

...Program finished with exit code 0
*/
```

AKIŞLAR

Dosya ve Akış Nedir?

Yazdığımız programlarda sadece girdi ve çıktı işlemleri yaparsak, program çalıştığı sürece veriler var olur, program sonlandırıldığında o verileri tekrar kullanamayız. Bunun için elektrik kesildiğinde bellekteki veriler kaybolacağından harici hafıza ortamında (external memory) verileri saklamak gerekir.

Kısaca dosya (file), verilerin bir araya geldiği (collection of data) ikincil saklama ortamıdır (seconder storage). Bu ikincil ortam; Bir bilgisayar belleği (memory) olabileceği gibi, verilerin elektrikler kesildiğinde kaybolmayacağı sabit disk (hard disc), ya da bir başka ortama/bilgisayara veri gönderen (send) veya alan (receive) modem ve benzeri bir cihaz (device) olabilir.

Veriler dosyalara aynı nehirde birbirinin peşi sıra giden tekneler gibi blok-blok ya da bayt-bayt gönderilir ya da dosyadan alınırlar. Bu nedenle dosyalara veri taşıyan ve veri getiren nesnelere nehir veya akış (stream) adı verilir. Dosyaya giden nehre hangi veriyi koyarsak onu nehrin sonunda onu dosyaya koyar. Benzer şekilde nehirden hangi sırada veri gelirse o sırada verileri nehirden alırız.

C++ dilinde giriş çıkış işlemleri akışlar aracılığıyla yapılır. Bu işlemler için soyut sınıflar şunlardır:

- Metin okumak için `std::istream`.
- Metin yazmak için `std::ostream`.
- Karakterleri okumak veya yazmak için `std::streambuf`.
- Biçimlendirilmiş girdi için `>>` işleci kullanır.
- Biçimlendirilmiş çıktı için `<<` işleci kullanır.
- Akışlar, örneğin biçimlendirmenin ayrıntıları ve harici kodlamalar ile dahili kodlama arasındaki çeviri için `std::locale` kullanır.

Şimdiye kadar, standart girdiden yani klavyeden okuma ve standart çıktıya yani konsola yazma için sırasıyla `cin` ve `cout` nesnelerini sağlayan akışların (stream) içinde tanımlı olduğu `iostream` başlığını kullandık.

Standart Akışlar

C++ dili programcıya, girdi ve çıktı işlemleri (input output operation) için hazır ve sürekli açık akış (stream) sunar. Üç metin akış nesnesi önceden `iostream` başlık dosyasında tanımlanmıştır;

Standart Akışlar	Akış	Açıklama
Standart Giriş	`cin`	Standart giriş akışıyla ilişkilendirilir, geleneksel klavye girişini okumak için kullanılır. Program başlangıcında, akış yalnızca ve yalnızca akışın etkileşimli bir cihaza başvurmadığı belirlenirse tamamen arabelleğe alınır.
Standart Çıktı	`cout`	Standart çıktı akışıyla ilişkilendirilir, geleneksel ekrana çıktıyı yazmak için kullanılır. Program başlangıcında, akış yalnızca ve yalnızca akışın etkileşimli bir cihaza başvurmadığı belirlenirse tamamen arabelleğe alınır.
Standart Hata	`cerr`	Standart hata akışıyla ilişkilendirilir, hata çıktısını kullanıcı ekranına yazmak için kullanılır. `cerr` verileri hemen çıktı olarak verir, yani verileri bir arabelleğe kaydetmez. Bu, hata mesajları için idealdir, çünkü bunların hemen görüntülenmesi gerekir.

Tablo 23. C++ Dilinde Standart Giriş Çıkış Akışları

Standart Almayan Akışlar

Ancak veriler üzerinde yapılan işlemleri kalıcı olarak diske saklamak veya kalıcı olan diskten verileri okumak için dosyalar kullanılır. Dosyalarla işlem yapmak için üç nesne kullanılır;

- `ofstream` sınıfı, verileri çıktıya taşıyan akışı (output stream) temsil eder ve dosyalar oluşturmak ve dosyalara bilgi yazmak için kullanılır.
- `ifstream` sınıf, verileri girdiden alan ve bize getiren akışı (input stream) temsil eder ve dosyalardan bilgi okumak için kullanılır.
- `fstream` sınıfı ise genel olarak hem veri gönderilen hem de veri alınan bir nehirdir. Hem `ofstream` hem de `ifstream` özelliklerine sahiptir; yani dosyalar oluşturabilir, dosyalara bilgi gönderebilir ve dosyalardan veri alabilir.

C++'da dosya işlemeyi gerçekleştirmek için, kaynak kodunuza `<iostream>` veya `<fstream>` başlık dosyalarını `#include` ön işlemci yönergesi (preprocessor direvtive) ile dahil edilmesi gerekir.

Bir dosyadan okuyabilmeniz veya dosyaya yazabilmeniz için önce dosyanın açılması gerekir. Bir dosyayı yazmak için açmak için `ofstream` veya `fstream` sınıflarından biri kullanılabilir. Yalnızca okuma amacıyla bir dosyayı açmak için `ifstream` nesnesi kullanılır. Aşağıda `fstream`, `ifstream` ve `ofstream` sınıflarının üyesi olan `open()` yöntemi ile dosyanın nasıl açılacağı gösterilmiştir;

```
void open(const char* dosya-adı, ios::openmode işlem-modu);
```

Akışın `open()` yönteminin ilk argümanı açılacak dosyanın adını ve konumunu belirtirken, ikinci argümanı ise dosyanın hangi modda açılacağını tanımlar. Dosya açma modlarından ikisini veya daha fazlasını bit düzeyi VEYA (bitwise or) işlemiyle birleştirebilirsiniz.

İşlem Modu	Açıklama
`ios::app`	Gönderilen bütün veri, dosyanın sonuna (append) eklenir.
`ios::ate`	Dosyanın sonunda (at the end) gidilerek hem okuma hem de yazma amacıyla dosya açılır.
`ios::in`	Dosya okumak için yani sadece veri girdisi (input) için kullanılır.
`ios::out`	Dosya yazmak için yani dosyaya sadece veri göndermek (output) için kullanılır.
`ios::trunc`	Mevcut bir dosya varsa açmadan önce içi boşaltılır (truncate). .

Tablo 24. Dosya Açma İşlem Modları

Örneğin `cikti.txt` adlı bir dosyayı her seferinde içini boşaltarak sadece yazmak için akış nesnesi aşağıdaki gibi açılabilir ve akış nesnesiyle açılan dosyaya aynı `std::cout` nesnesinde olduğu gibi akışa veri ekleme (stream insertion operator) işleci olan `<<` ile veri ekleyebilir veya `std::cin` nesnesinde oluğu gibi akıştan veri çıkarma işleci (stream extraction operator) olan `>>` ile veri okuyabilirsiniz.

```
ofstream outfile;
outfile.open("cikti.txt", ios::out | ios::trunc );
if(outfile.is_open()){
    os << "Hello World!";
}
```

Benzer şekilde, aynı dosyayı hem okuma ve hem de yazma amacıyla aşağıdaki gibi açabilirsiniz;

```
fstream  afile;
afile.open("cikti.txt", ios::out | ios::in );
if (!afile.is_open()) {
    throw CustomException(afile, "Dosya Açılamadı!");
}
```

Çıktı dosyası akışında `<<` işleci yerine, üye fonksiyon olan `write()` da kullanılabilir:

```
if(afile.is_open()){
    char metin[] = "İlhan";
    os.write(metin, 3); // Metinden 3 characters akışa yazılıyor
}
```

Dosyadan veri okuma aşağıdaki şekilde yapılabilir;

```
/*Metin.txt Dosasında aşağıdakilerin olduğunu varsayalım;
İlhan Özkan 25 4 6 1987
Mehmet Yıldırım 15 5 24 1976
```

```
*/
std::ifstream is("metin.txt");
std::string adi, soyadi;
int yas, dogumAyi, dogumGunu, dogumYili;
while (is >> adi >> soyadi >> yas >> dogumAyi >> dogumGunu >> dogumYili)
   cout << adi << " " << soyadi;
```

Bir C++ programından çıkıldığında otomatik olarak tüm akışları temizler, ayrılmış belleği serbest bırakır ve açık tüm dosyaları kapatır. Ancak bir programcının programı sonlandırmadan önce açık olan tüm dosyaları kapatması her zaman iyi bir uygulamadır.

Aşağıda `fstream`, `ifstream` ve `ofstream` nesnelerinin üyesi olan `close()` yönteminin kullanılarak dosyaları kapatabiliriz.

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main () {
    string adiSoyadi;
    int yas;
    char cinsiyet;

    cout << "Adınızı Giriniz: ";
    getline(cin, adiSoyadi);
    cout << "Yaşınızı Giriniz: ";
    cin >> yas;
    cout << "Cinsiyetinizi Giriniz (K-E): ";
    cin >> cinsiyet;

    //Program bilgilerini dosyaya yazalım:
    ofstream outfile;  // akış nesnesi tanımlandı.
    outfile.open("output.txt"); //dosya açıldı
    outfile << "Bu dosya Yaş ve Cinsiyet Bilgilerini İçerir" << endl;
    outfile << "Adı:" << adiSoyadi << endl;
    outfile << "Yaşı:" << yas << endl;
    outfile << "Cinsiyeti:" << cinsiyet << endl;
    outfile.close(); // dosya kapandı

    //Yardığımız dosyayı okuyalım:
    string satir;
    ifstream infile; // akış nesnesi tanımlandı.
    infile.open("output.txt"); //dosya açıldı
    getline(infile, satir);
    cout << "Dosyadan Okunan Satır:" << endl << satir << endl;
    getline(infile, satir);
    cout << "Dosyadan Okunan Bir Sonraki Satır:" << endl << satir << endl;
    infile.close(); // dosya kapandı
}
/*Program Çalıştırıldığında:
Adınızı Giriniz: İlhan Özkan
Yaşınızı Giriniz: 50
Cinsiyetinizi Giriniz (K-E): E
Dosyadan Okunan Satır:
Bu dosya Yaş ve Cinsiyet Bilgilerini İçerir
Dosyadan Okunan Bir Sonraki Satır:
Adı:İlhan Özkan

...Program finished with exit code 0
```

```
*/
```

Program çalıştığında oluşan `output.txt` metin dosyasının içeriği aşağıda verilmiştir;

```
Bu dosya Yaş ve Cinsiyet Bilgilerini İçerir
Adı:İlhan Özkan
Yaşı:50
Cinsiyeti:E
```

Buraya kadar işlediğimiz akışlar konsola yazdığımız veya klavyeden okuduğumuz gibi metin olarak çalışır. Yani oluşan dosyalar gözle okunabilir metinlerdir.

Çalışılan dosyalar her zaman metin dosyası olamayabilir. Bir nesneyi bellekte olduğu gibi yani ikili olarak aynen dosyaya saklamak isteyebiliriz. Bu durumda akışları ikili dosya yazıp okuyacak şekilde açıp işlem yapmalıyız. Bu durumda dosya açma modu olarak `ios::binary` kullanılmalıdır.

Aşağıda ikili olarak oluşturulan bir dosya örneği verilmiştir.

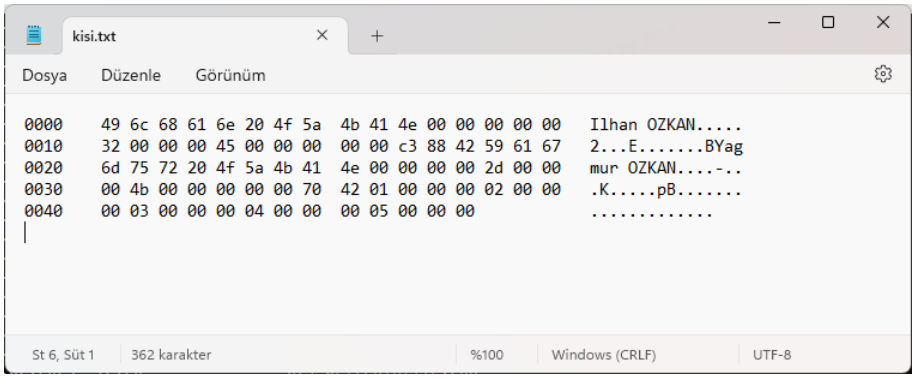
```
#include <iostream>
#include <fstream>
using namespace std;
struct kisi {
    char adiSoyadi[16];
    int yas;
    char cinsiyet;
    float kilo;
};
typedef struct kisi Kisi;

int main() {
    Kisi kisi1={"Ilhan OZKAN",50,'E',100.0};
    Kisi kisi2={"Yagmur OZKAN",45,'K',60.0};
    int dizi[5]={1,2,3,4,5};

    ofstream os("kisi.bin", std::ios::binary);
    if (!os) {
        cerr << "Dosya Açılamadı!" << std::endl;
        return 1;
    }
    os.write(reinterpret_cast<char*>(&kisi1), sizeof(Kisi));
    os.write(reinterpret_cast<char*>(&kisi2), sizeof(Kisi));
    os.write(reinterpret_cast<char*>(&dizi), sizeof(dizi));
    os.close();
}
```

Oluşan `kisi.bin` dosyasına ilk önce iki adet kişi yapısı kaydedilmiştir. Sonrasında 5 tamsayıdan oluşan dizi kaydedilmiştir. Bellekte saklandığı şekliyle dosyaya kaydedildiğinden içeriği ikili olduğundan ikili dosyaları açan görüntüleyiciler ile açılıp görülebilir.

Bir ikili dosyayı onaltılık rakamlara çevirip metin olarak görüntülemek için Windows ortamında aşağıdaki komut verilebilir;

```
C:\Users\ILHANOZKAN>certutil -encodehex kisi.bin kisi.txt
Input Length = 77
Output Length = 367
CertUtil: -encodehex command completed successfully.
```

Artık dönüştürülen dosyayı (`kisi.txt`) istediğimiz metin düzenleyicisinde açıp okuyabiliriz.



Şekil 24. Programın Oluşturduğu İkili Dosya İçeriği

Aynı dosyayı ikili olarak okuyan program aşağıdaki şekilde yazılabilir;

```
#include <iostream>
#include <fstream>
using namespace std;
struct kisi {
    char adiSoyadi[16];
    int yas;
    char cinsiyet;
    float kilo;
};
typedef struct kisi Kisi;

int main() {
    Kisi kisi1, kisi2;
    int dizi[5];

   ifstream is("kisi.bin", std::ios::binary);
   if (!is) {
      cerr << "Dosya Açılamadı!" << std::endl;
      return 1;
   }
   is.read(reinterpret_cast<char*>(&kisi1), sizeof(Kisi));
   cout << "Okunan Kisi Yapısı:" << endl
        << "Kişi Adı:" << kisi1.adiSoyadi << endl
        << "Kişi Yaşı:" << kisi1.yas << endl
        << "Kişi Cinsiyeti:" << kisi1.cinsiyet << endl
        << "Kişi Kilo:" << kisi1.kilo << endl;
   is.read(reinterpret_cast<char*>(&kisi2), sizeof(Kisi));
   cout << "Okunan Kisi Yapısı:" << endl
        << "Kişi Adı:" << kisi2.adiSoyadi << endl
        << "Kişi Yaşı:" << kisi2.yas << endl
        << "Kişi Cinsiyeti:" << kisi2.cinsiyet << endl
        << "Kişi Kilo:" << kisi2.kilo << endl;
   is.read(reinterpret_cast<char*>(&dizi), sizeof(dizi));
   cout << "Okunan Dizi:" << dizi[0] << "," << dizi[1] << ","
        << dizi[2] << "," << dizi[3] << "," << dizi[4] << endl;
   is.close();
}
/* Program Çalıştığında:
Okunan Kisi Yapısı:
Kişi Adı:Ilhan OZKAN
Kişi Yaşı:50
Kişi Cinsiyeti:E
Kişi Kilo:100
Okunan Kisi Yapısı:
Kişi Adı:Yagmur OZKAN
```

```
Kişi Yaşı:45
Kişi Cinsiyeti:K
Kişi Kilo:60
Okunan Dizi:1,2,3,4,5

...Program finished with exit code 0
*/
```

Eğer sadece dizili okumak isteseydik dosya konum göstericisini okuyacağımız yere konumlandırmak gerekir;

```
#include <iostream>
#include <fstream>
using namespace std;
struct kisi {
    char adiSoyadi[16];
    int yas;
    char cinsiyet;
    float kilo;
};
typedef struct kisi Kisi;

int main() {
    Kisi kisi1, kisi2;
    int dizi[5];

   ifstream is("kisi.bin", std::ios::binary);
   if (!is) {
      cerr << "Dosya Açılamadı!" << std::endl;
      return 1;
   }
   is.seekg(2*sizeof(Kisi),ios::beg);
   // dizi iki kişi kaydı sonrasında kaydedilmişti.
   is.read(reinterpret_cast<char*>(&dizi), sizeof(dizi));
   cout << "Okunan Dizi:" << dizi[0] << "," << dizi[1] << ","
        << dizi[2] << "," << dizi[3] << "," << dizi[4] << endl;
   is.close();
}
```

Hem `istream` hem de `ostream` başlığı ikili olarak açılan dosya için dosya konum göstericisini yeniden konumlandırmak için üye fonksiyonları sağlar. Bunlar `istream` için konumu öğrenmede (seek get) kullanılan `seekg()`ve `ostream` için yeniden konumlandırma (seek put) için `seekp()` fonksiyonlarıdır.

Bu fonksiyonlarda konumlandırmanın yönü, bir akışın başlangıcına göre konumlandırma için `ios::beg` (varsayılan), bir akıştaki geçerli konuma göre konumlandırma için `ios::cur` veya bir akışın sonuna göre konumlandırma için `ios::end` kullanılır.

```
streamObject.seekg( n ); // dosya başından itibaren n inci bayt ilerle (varsayılan ios::beg)
streamObject.seekg( n, ios::cur ); // mevcut konumdan itibaren n bayt ilerle
streamObject.seekg( n, ios::end ); // dosya sonundan itibaren n bayt geri gel
streamObject.seekg( 0, ios::end ); // pdosya sonuna ilerle.
```

std::cerr Nesnesini Dosyaya Yönlendirme

Hataları izlemek için kullanılan `std::cerr` nesnesi genellikle konsola hata çıkışı vermek için kullanılsa da bir dosyaya da yönlendirilebilir. Bu, bir programda hataları günlüğe kaydetmeniz gerektiğinde yararlı olabilir.

```
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main() {

    ofstream errorLog("log.txt");
    cerr.rdbuf(errorLog.rdbuf());
    cerr << "Hata mesajları bu dosyaya yazılacak!." << endl;
    errorLog.close();
}
```

stringstream Sınıfı

C++ dilinde `stringstream` sınıfı birden fazla dizgiyi aynı anda girdi olarak almak için kullanılır. Kısaca dosya yerine bir metni akış olarak kullanır. Aşağıda örnek bir uygulama verilmiştir;

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main() {
    string metin = " Merhaba Ilhan OZKAN. Nasılsınız? ";
    stringstream metinAkisi(metin);
    string sozcuk;
    while (metinAkisi >> sozcuk) {
        cout << sozcuk << endl;
    }
}
/* Program Çalıştığında:
Merhaba
Ilhan
OZKAN.
Nasılsınız?

...Program finished with exit code 0
*/
```

Dosyanın tamamın bir `stringstream` nesnesine konulabilir;

```
std::ifstream f("metin.txt");
if (f)
{
    std::stringstream buffer;
    buffer << f.rdbuf();
    f.close();
}
```

TASARIM DESENLERİ

Tasarım Deseni Nedir?

Nesne yönelimli programlamanın ortaya çıkışıyla beraber, daha önceden yazılmış hazır bileşenlerin (component) yeniden kullanımı (reusing) konusunda oldukça önemli ilerlemeler sağlamıştır. Bunun paralelinde tecrübelerin (experience) yeniden kullanımı konusunda da çeşitli çalışmalar yapılmış ve desen (pattern) kavramı ortaya çıkmıştır. Desenlerin aksine anti-desen (anti-pattern) ise başarısızlık tecrübelerini anlatır.

Yazılım geliştirme öncesi, geliştirme aşması ve sonrasında daha önce yaşanmış çeşitli problemlere karşı birçok kimse tarafından çeşitli çözümler getirilmiştir. Desenler, daha önce geliştirme yapan programcıların yanlış yaparak doğru yapmayı öğrendikleri başarılı tecrübelerin anlatılması için bir araçtır. Gerçekleştirilen bu çözümlerin, daha sonradan yaşanacak benzer nitelikli problemlerde kullanılması amacıyla desenler kullanılır. Desen kavramının temelleri Mimar Christopher Alexander'ın 1970 sonlarında başlattığı çalışmalara dayanmaktadır[21].

1987 yılında uluslararası "Nesne Yönelimli Programlama, Sistemler, Diller ve Uygulamalar" konferansına kadar desenlerle ilgili bir çalışma ortaya çıkmamıştır[22]. Bu tarihten sonra ise başta *Grady Booch*, *Richard Helm*, *Erich Gamma* ve *Kent Beck* olmak üzere pek çok bilim adamı desenlerle ilgili makale ve sunumlar yayınlamışlardır. 1994 yılında *Erich Gamma, Richard Helm, Ralph Johnson* ve *John Vlissides* tarafından yayınlanan "Tasarım Desenleri: Tekrar kullanılabilir Nesne Yönelimli Yazılımın Temelleri" kitabı, tasarım desenlerin yazılımda kullanılmasında dönüm noktası olmuştur[23]. Yazılımlarda kullanılan desenler yedi ana başlıkta toplanırlar. Bunlar;

1. Mimari desenler (architectural pattern)
2. Analiz desenleri (analysis pattern)
3. Tasarım desenleri (design pattern)
4. Kodlama desenleri (implementation pattern)
5. Test desenleri (test pattern)
6. Çözüm desenleri (solution pattern)
7. Veri desenleri (data pattern)

Bu desenlerin en önemlisi ve ilk gelişenlerden birisi, tasarım desenleridir (design pattern). Tasarım desenleri aslında, Nesne Yönelimli Tasarım Prensiplerini, yazılımlara uygulamanın bir başka ifadesidir. Bu nedenle çok önemlidir. Tasarım desenleri ise üç ana başlıkta toplanmaktadırlar.

1. Nesne imali ile ilgili desenler (creational patterns)
2. Yapısal desenler (structural patterns)
3. Davranışlarla ilgili desenler (behavioral patterns)

Desenlerin çok sık kullanıldığı programlama yöntemine günümüzde desen yönelimli programlama (pattern oriented programming) adı verilmektedir[24]. İzleyen sayfalarda verilen desen UML diyagramlarının birçoğu Erich Gamma'nın Design Pattern CD yayınından alınmıştır[25].

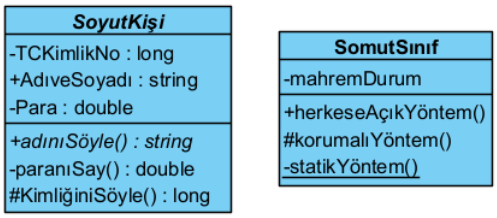
Bölümün devamında verilecek tasarın desenlerinin UML diyagramları visual-paradigm adresinden alınmıştır[26]. Bu bölümde verilen örnekleri iyi anlamak için

Sınıf diyagramları (class diagram), yazılımı geliştirilecek olan sisteme ait sınıfları (class) ve bu sınıflar arasındaki ilişkiyi (relationship) gösterir. Sınıf diyagramları durağandır (static). Yani sadece sınıfların

[21] http://gee.cs.oswego.edu/dl/ca/ca/ca.html
[22] OOPSLA, Object Oriented Programming, Systems, Languages, and Applications
[23] Design Patterns: Elements of Reusable Object-Oriented Software, ISBN 0-201-63361-2
[24] www.mathcs.sjsu.edu/faculty/pearce/patterns.html
[25] Design Pattern CD, Addison-Wesley,1998
[26] https://circle.visual-paradigm.com/category/uml-diagrams/gof-design/

birbiriyle etkileşimini (what interacts) gösterir, bu etkileşimler sonucunda ne olduğunu (what happens) göstermez.

Sınıf diyagramlarında sınıflar, bir dikdörtgen ile temsil edilir. Bu dikdörtgen üç parçaya ayrılır. En üstteki birincisine sınıf ismi yazılır. Ortadaki ikincisinde nitelikler (attribute), en alttaki ve sonuncusunda ise ve işlemler (operation) yer alır.



Şekil 33. Örnek Bir Sınıf Diyagramı

Sınıf isminin italik yazılması, sınıfın soyut (abstract class) sınıf olduğunu gösterir. Benzer şekilde işlemlerin yer aldığı kısımda, yöntemlerin italik yazılması halinde, ilgili yöntemin soyut yöntem (abstract method) olduğunu gösterir.

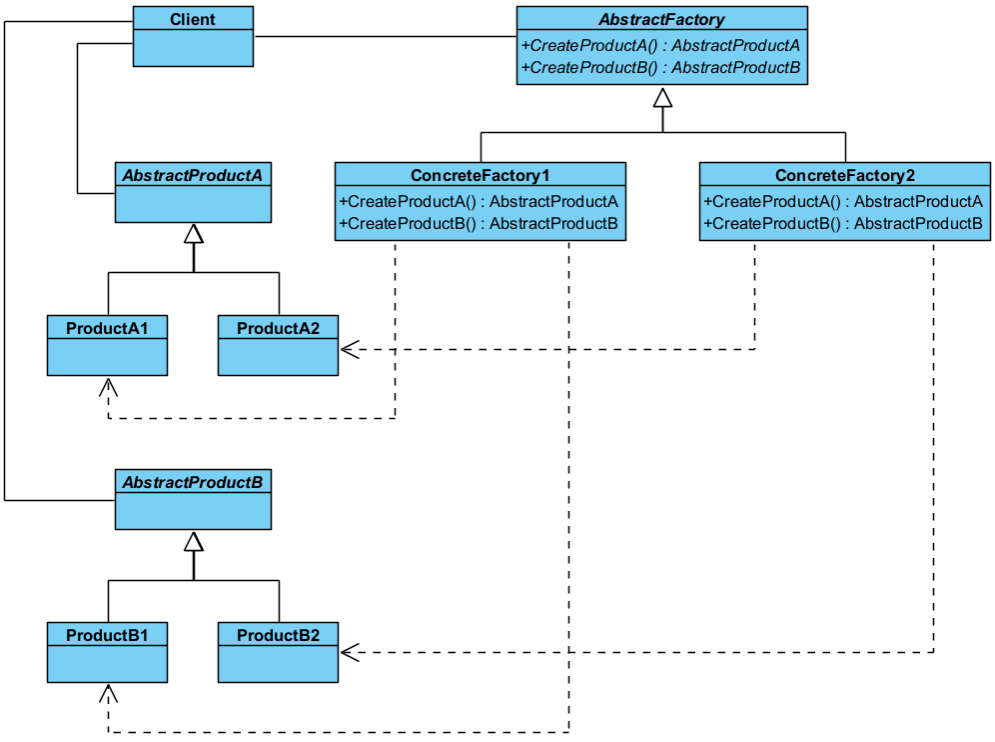
Sınıflarda erişim belirleyiciler (access modifier), özel karakterlerle birbirinden ayrılır. Burada "-" karakteri mahrem (private), "+" karakteri umuma açık (public), "#" karakteri ise korumalı (protected) olduğunu anlatmaktadır. Statik üyeler (static member), altı çizili olarak gösterilir.

Sınıflar Arası İlişkiler başlığını iyi özümsemek gerekir.

Nesne İmalatı ile İlgili Desenler

Nesne imalatı ile ilgili desenler (creational patterns), nesne (object) örnekleme ile ilgili ortaya çıkabilecek çeşitli problemlere çözüm getirmektedirler. Bu desenlerde nesneler dinamik olarak imal edilirler, çünkü değişim yönetiminde statik nesneler kullanılmazlar.

Soyut Fabrika



Şekil 34. Soyut Fabrika Deseni UML Diyagramı

Soyut fabrika deseninde (abstract factory pattern)amaç, nesnelerin imal edilmesi ve kullanılması ile ilgili olan kısımlarının birbirinden ayrılmasıdır. Bu amaca, birbiriyle ilgili ya da birbirine bağımlı olabilecek nesnelerin imal edilmesinde, nesnelerin somut (concrete) sınıflarını değil soyut (abstract) sınıfları kullanılarak ulaşılır. Somut sınıflar istemciden izole edilir ve programcıya somut sınıfları daha kolay değiştirme için olanak sağlar. Bu baslıktaki desenlerde, nesnelerin (objects) yapılandırılıp imal edildiği (creation) yer için fabrika (factory) kavramı, üretilen nesneler için ise ürün (product) kavramı kullanılmaktadır. Aşağıdaki UML diyagramında görüldüğü üzere istemci yalnızca soyut sınıflara bağımlı olduğundan somut ürüne olan bağımlılık azaltılmıştır (loosely coupling).

Şimdi bu deseni kodlamaya başlayalım. İlk önce `AbstractProductA` ve alt sınıflarını kodlayalım.

```
#include <iostream>
using namespace std;

class AbstractProductA {
public:
    AbstractProductA(string pAdi):productName(pAdi) {
    };
    virtual string getUrunAdi(){
        return productName;
    };
private:
    string productName;
};
class ConreteProductA1: public AbstractProductA {
public:
    ConreteProductA1(string pAdi): AbstractProductA(pAdi) { }
};
class ConreteProductA2: public AbstractProductA {
public:
    ConreteProductA2(string pAdi): AbstractProductA(pAdi) { }
};
```

Ardından `AbstractProductB` ve alt sınıflarını kodlayalım. Bu sınıftan imal edilecek ürünler `AbstractProductA` ürünleriyle etkileşim yaptığı varsayılarak bir `interact` yöntemi eklenmiştir.

```
class AbstractProductB {
public:
    AbstractProductB(string pAdi):productName(pAdi) {
    };
    virtual string getUrunAdi(){
        return productName;
    };
    void interact(AbstractProductA* a) {
        cout << this-> productName
             << " ürünü " << a->getUrunAdi()
             << " ile etkileşim halinde..." << endl;
    }
private:
    string productName;
};
class ConreteProductB1: public AbstractProductB {
public:
    ConreteProductB1(string pAdi): AbstractProductB(pAdi) { }
};
class ConreteProductB2: public AbstractProductB {
public:
    ConreteProductB2(string pAdi): AbstractProductB(pAdi) { }
};
```

Ardından ürünleri imal edecek fabrika sanal sınıfı ve somut sınıflarını tanımlayalım;

```
class AbstractFactory {
public:
    virtual AbstractProductA* createProductA()=0;
    virtual AbstractProductB* createProductB()=0;
};
class ConreteFactory1: public AbstractFactory {
public:
    AbstractProductA* createProductA() override {
        return new ConreteProductA1("A1 ürünü");
    }
    AbstractProductB* createProductB() override {
        return new ConreteProductB1("B1 Ürünü");
    };
};
class ConreteFactory2: public AbstractFactory {
public:
    AbstractProductA* createProductA() override {
        return new ConreteProductA2("A2 ürünü");
    }
    AbstractProductB* createProductB() override {
        return new ConreteProductB2("B2 Ürünü");
    };
};
```

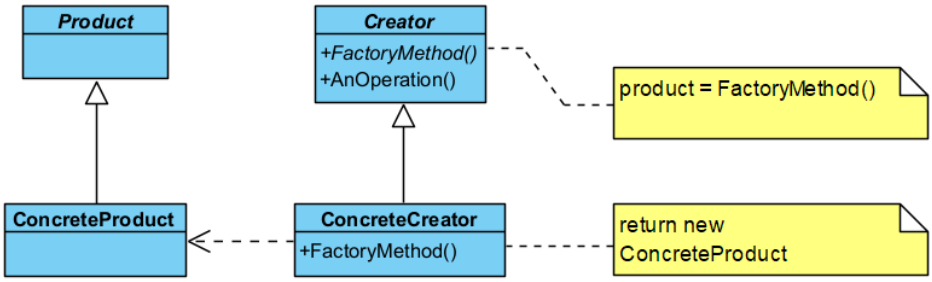
Son olarak fabrika ve ürün sınıflarını sanal olarak kullanan istemci programı kodlayalım;

```
int main() {// Client
    AbstractFactory* factory1=new ConreteFactory1();
    AbstractFactory* factory2=new ConreteFactory2();
    AbstractProductA* productA=factory1->createProductA();
    cout << "Fabrika 1 tarafından İmal Edilen Ürün:" << productA->getUrunAdi() << endl;
    AbstractProductB* productB=factory2->createProductB();
    cout << "Fabrika 2 tarafından İmal Edilen Ürün:" << productB->getUrunAdi() << endl;

    productB->interact(productA);

    delete productB;
    delete productA;
    delete factory2;
    delete factory1;
}
/* Program Çalıştığında;
Fabrika 1 tarafından İmal Edilen Ürün:A1 ürünü
Fabrika 2 tarafından İmal Edilen Ürün:B2 Ürünü
B2 Ürünü ürünü A1 ürünü ile etkileşim halinde...

...Program finished with exit code 0
*/
```

Fabrika Yöntemi

Fabrika yöntemi deseninde (factory method pattern) amaç, imal edilecek nesnenin sınıfını kesin olarak belirtmeden nesne yaratma işleminin gerçekleştirilmesidir. Bunu yapmak için fabrika yöntemi adında soyut (abstract) bir yöntem tanımlanır, fakat nesneleri imal etme (instantiation) işlemi alt sınıflara bırakılır. Yani sanal bir yapıcı (virtual constructor) kullanılır.

Soyut fabrika deseninde olduğu gibi dinamik nesne imal etme ile ilgili `new` anahtar kelimesi, fabrika yöntemi içinde kullanıldığından, istemcinin, ürünün yapıcısına (construtor) bağımlılığı ortadan kalkar. Aşağıdaki UML diyagramında görüldüğü üzere istemcinin somut ürünün yapıcısına bağımlılığı kaldırılmış ve nesne imal etme süreci kontrol altına alınmıştır.



Şekil 35. Fabrika Yöntemi Deseni UML Diyagramı

Şimdi bu deseni kodlamaya başlayalım. İlk önce `Pruduct` ve alt sınıflarını kodlayalım.

```
#include <iostream>
using namespace std;

class Product {
public:
    Product(string pAdi):productName(pAdi) {
    };
    virtual string getUrunAdi(){
        return productName;
    };
private:
    string productName;
};
class ConreteProduct: public Product {
public:
    ConreteProduct(string pAdi): Product(pAdi) { }
};
```

Ardından ürün imal edecek olan imalatçı `Creator` sınıfını kodluyoruz;

```
class Creator {
public:
    virtual Product* factoryMethod() =0;
    virtual void anOperation() =0;
};
class ConreteCreator: public Creator {
public:
    Product* factoryMethod() override {
        return new ConreteProduct("Fabrika Yöntemi Ürünü");
    };
    void anOperation() override {
        cout << "Fabrika Yöntemini Kullanan Sınıf, Başka İşlemler de yapabilir..." << endl;
    }
};
```

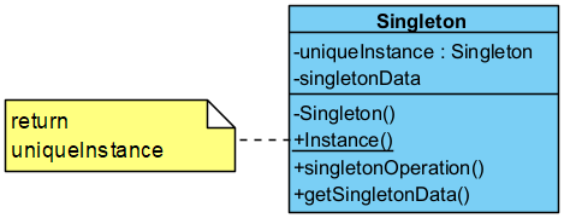
Son olarak ürün ve imalatçı sınıflarını sanal olarak kullanan istemci programı kodlayalım;

```
int main() {// Client
    Creator* imalatci=new ConreteCreator();
    Product* urun=imalatci->factoryMethod();

    cout << "İmalatçı tarafından imal edilen ürün:" << urun->getUrunAdi() << endl;
    imalatci ->anOperation();
    delete urun;
    delete imalatci;
}
/* Program Çıktısı:
İmalatçı tarafından imal edilen ürün:Fabrika Yöntemi Ürünü
Fabrika Yöntemini Kullanan Sınıf Başka, İşlemler de yapabilir...
```

```
...Program finished with exit code 0
*/
```

Tekil Nesne

Tekil Nesne deseninde (singleton pattern) amaç, bir sınıftan yalnızca bir örnek (instance) imal etmektir. Bir sınıfın yalnızca bir nesnesinin olmasına izin verilir, birden fazla olmasına izin verilmez. Yaratılan nesneye genel olarak evrensel (global) olarak erişilir.



Şekil 36. Tekil Deseni UML Diyagramı

Verilen UML diyagramı incelendiğinde, `Singleton` sınıfının `instance` yöntemi her seferinde aynı nesneye ilişkin referans geri döndürülür. Bu desen aynı zamanda, nesneye ilk değerlerin verilerek oluşturulduğu, imal edilme sürecini ve ilk kullanımını izole (encapsulate) eder. Böylece istemcilerin her biri aynı nesne ile muhatap olurlar.

Şimdi `Singleton` sınıfını kodlayalım;

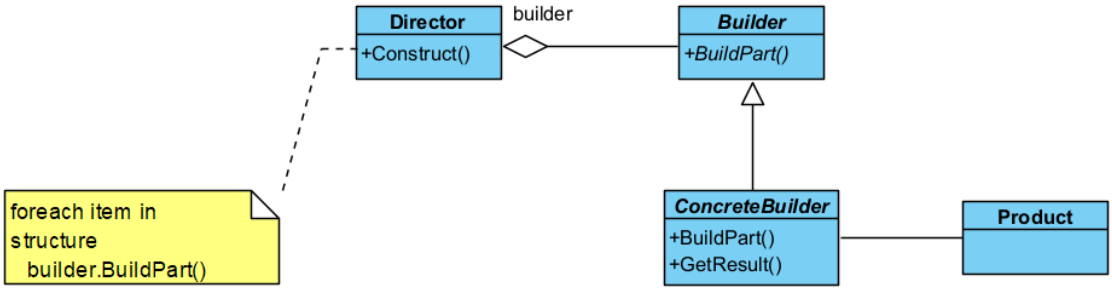
```
#include <iostream>
using namespace std;

class Singleton {
private:
    Singleton() { // Her seferinde başka nesne oluşmasın diye mahrem tanımlanmış
        singletonData=10;
    }
    static Singleton* uniqueInstance;
    int singletonData;
public:
    Singleton(Singleton &other) = delete; // Tekil nesne klonlanabilir olmamalıdır.
    void operator=(const Singleton &) = delete;
        // Tekil nesne atanabilir olmamalıdır.
    static Singleton *Instance() {
        /*
          Bu statik yyöntem;
          Tekil nesneye erişimi kontrol eden statik yöntemdir.
          İlk çalıştırmada, tekil bir nesne imal eder ve onu statik alana
          yerleştirir.
          Sonraki çalıştırmalarda, statik alanda depolanan nesneyi döndürür.
        */
        if(uniqueInstance==nullptr)
            uniqueInstance = new Singleton();
        return uniqueInstance;
    }
    void anOperation() {
        cout << "Tekil nesnenin davranışları da olabilir..." << endl;
    }
    int getSingletonData() const{
        return singletonData;
    }
};

Singleton* Singleton::uniqueInstance= nullptr; // ilk değer verilmeli.
```

Bu sınıfı kullanan istemci aşağıdaki gibi yazılabilir;

```
int main() {
    Singleton* singleton1 = Singleton::Instance();
    Singleton* singleton2 = Singleton::Instance();

    if (singleton1==singleton2)
        cout << "singleton1 ve singleton2 AYNI nesnedir." << endl;
    else
        cout << "singleton1 ve singleton2 FARKLI nesnedir." << endl;

    singleton1->anOperation();
    cout << "Tekil nesnenin verisi:" << singleton1->getSingletonData() << endl;
    delete singleton1;
}
/* Program Çıktısı:
singleton1 ve singleton2 AYNI nesnedir.
Tekil nesnenin davranışları da olabilir...
Tekil nesnenin verisi:10

...Program finished with exit code 0
*/
```

Kurucu

Kurucu deseninde (builder pattern) amaç, çok karmaşık bir nesnenin imal edilmesiyle ilgili işlemleri bir başka sınıfa bırakmaktır. Böylece nesnenin gösterimi ve kullanılması ilişkin kodlar ile yaratılmasına ilişkin kodlar birbirinden ayrılmış olur.



Şekil 37. Kurucu Deseni UML Diyagramı

Yukarıda verilen UML diyagramı incelendiğinde, Ürün imalatını yönetecek olan `Director` nesnesi, karmaşık nesneyi oluşturacak küçük ürünlerin (product) yapımını `Builder` nesnesine bırakır. `Director` nesnesi, oluşturulan bu küçük parçaları `Construct` yöntemi ile bir araya getirir.

İlk önce Product sınıfını tanımlayalım. Ürün birden fazla parçadan oluştuğu için burada temsilen `vector` kullanılmıştır.

```
#include <iostream>
#include <vector> // std::vector
#include <algorithm> //vector::iterator
using namespace std;
class Product {
public:
    Product(string pAdi):productName(pAdi) {
    };
    string getUrunAdi(){
        return productName;
    };
    void addPart(string pPart){
        parts.push_back(pPart);
    }
    void showParts() {
```

```
        cout << productName << " Ürününün parçaları:" << endl;
        vector<string>::iterator part = parts.begin();
        while( part != parts.end()) {
            cout  << *part << endl;
            part++;
        }
    }
private:
    string productName;
    vector<string> parts;
};
```

Buna ek olarak `Builder` ve alt sınıfını tanımlayalım;

```
class Builder {
public:
    virtual void buildPartKisim1()=0;
    virtual void buildPartKisim2()=0;
    virtual void buildPartKisim3()=0;
    virtual Product* getResult()=0;
};
class concreteBuilder:public Builder {
public:
    concreteBuilder() {
        product=new Product("Kitap");
    }
    void buildPartKisim1() override {
       product->addPart("Bölüm1: ...");
    };
    void buildPartKisim2() override {
       product->addPart("Bölüm2: ...");
    };
    void buildPartKisim3() override {
       product->addPart("Bölüm3: ...");
    };
    Product* getResult() override {
        return product;
    }
    ~concreteBuilder() {
        delete product;
    }
private:
    Product* product;
};
```

Şimdi de `Director` sınıfını tanımlayalım;

```
class Director {
public:
    void construct(Builder* builder){
        builder->buildPartKisim1();
        builder->buildPartKisim2();
        builder->buildPartKisim3();
    }
};
```

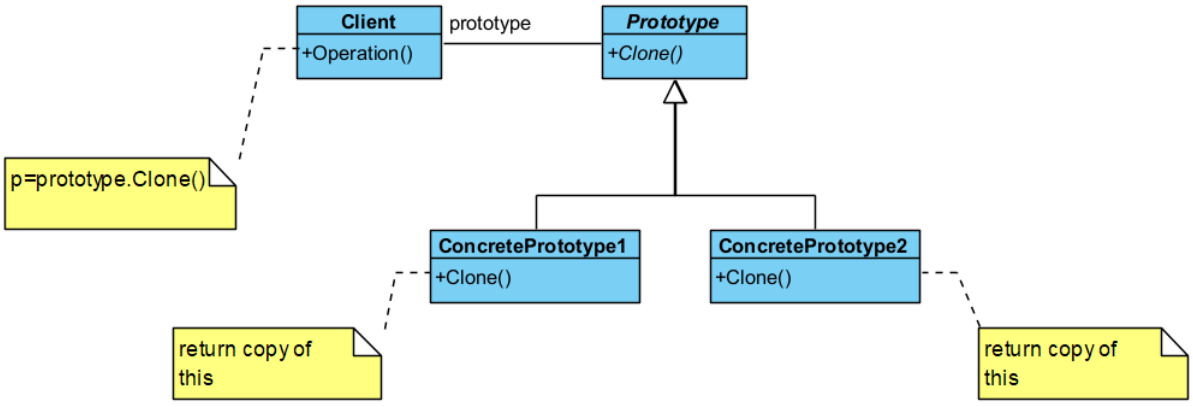
Son olarak istemciyi tanımlayalım;

```
int main() { // Client
    Director* director = new Director();
    Builder* builder = new concreteBuilder();
    director->construct(builder);
```

```
    Product* product = builder->getResult();
    product->showParts();
    delete director;
    delete builder;
}
/*Programın Çıktısı:
Kitap Ürününün parçaları:
Bölüm1: ...
Bölüm2: ...
Bölüm3: ...

...Program finished with exit code 0
*/
```

Prototip

Prototip deseni (prototype pattern), hâlihazırda mevcut olan nesnenin bir kopyasını çıkarmak için kullanılır. Bu desen, imal edilmesi güç olan nesnelere karar verilerek, bu nesneleri tekrar imal etmek için bir sürü işlem yapılması yerine, mevcut nesneden bir kopya çıkararak kullanılmasını sağlamaktadır.



Şekil 38. Prototip Deseni UML Diyagramı

Klonlama yoluyla var olan nesnelerin bir şablonuna dayalı yeni nesneler oluşturmak için kullanılan bu deseni kodlaması aşağıdaki şekilde yapılabilir;

```
#include <iostream>
using namespace std;
class Prototype {
protected:
    int data1;
    int data2;
public:
    void showData() {
        cout << "Object Data: data1=" << data1 <<",data2=" << data2 << endl;
    }
    virtual Prototype* clone()=0;
};
class ConcretePrototype1 : public Prototype {
//Birinci Şablon: Yalnızca data1 alanını kopyalayan prototip nesne imal eder
public:
    ConcretePrototype1(int pData1) {
        data1=pData1;
    };
    Prototype* clone() override {
        return new ConcretePrototype1(*this);
    }
};
class ConcretePrototype2 : public Prototype {
```

```
//İkinci Şablon: Yalnızca data2 alanını kopyalayan prototip nesne imal eder
public:
    ConcretePrototype2(int pData2) {
        data2=pData2;
    };
    Prototype* clone() override {
        return new ConcretePrototype2(*this);
    }
};
void operation() {
    Prototype* object1 = new ConcretePrototype1(40);
    object1->showData();
    Prototype* clone1 = object1->clone();
    clone1->showData();
    Prototype* object2 = new ConcretePrototype2(30);
    object2->showData();
    Prototype* clone2= object2->clone();
    clone2->showData();
    delete object1;
    delete clone1;
    delete object2;
    delete clone2;
}
int main() {  //İstemci
    operation();
}
/* Program Çıktısı:
Object Data: data1=40,data2=0
Object Data: data1=40,data2=0
Object Data: data1=0,data2=30
Object Data: data1=0,data2=30

...Program finished with exit code 0
*/
```

Nesneleri kopyalama söz konusu olduğunda, genel olarak üç tür kavramdan bahsedilir;

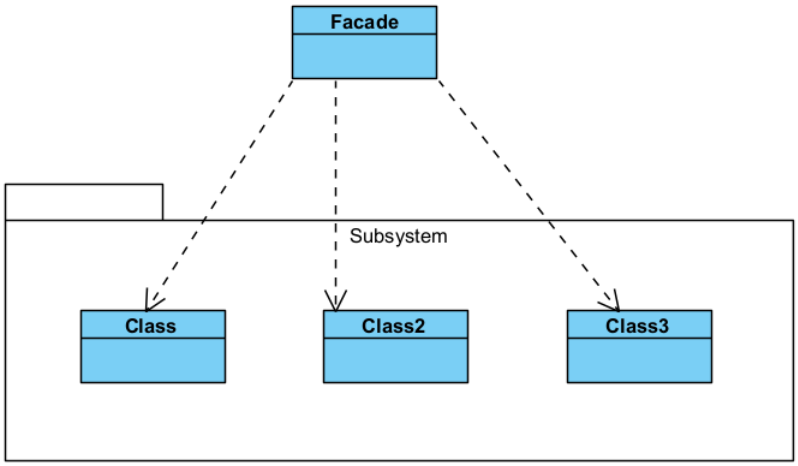
- Üstünkörü kopyalamada (shallow copy) yeni nesne eskisini referans gösterir. Bu durumda birinci nesnenin bir alanı (field) değiştiğinde ikincisinin ki de değişmiş olur. Yani birinde olan değişiklik diğerini etkiler. Bu kopyalamanın diğer adı da bit düzeyi kopyalamadır (bitwise copy).
- Derinlemesine kopyalamada (deep copy) ise birinci nesnenin aynısı bir başka bellek bölgesinde oluşturulur ve ikinci nesne yeni bellek bölgesini referans gösterir. Bu kopyalamanın diğer adı da üye düzeyi kopyalamadır (memberwise copy). Kopyalama sonrasında bir nesne üzerinde olan değişiklik yeni nesneyi etkilemez.
- Üçüncü tip bir kopyalama ise tembel kopyalamadır (lazy copy). Yukarıda bahsi geçen iki kopyalamanın iç içe girmiş şeklidir.

Yapısal Desenler

Yapısal desenler (structural pattern), sınıflar ve nesnelerin çok olduğu daha karmaşık programlarda getirilen yapısal çözüm yöntemleridir.

Vitrin

Vitrin deseninde (facade pattern) alt sistemlerin değişmesi halinde tüm sistemin yeniden derlenme olasılığını bertaraf etmek amacıyla alt sistemlere erişim tek bir sınıf üzerinden yapılır. Çok katmanlı yazılım mimarisinde, alt katmanların içinde olan değişiklikler sistem genelini etkilemez. Çünkü katmanlar birbirlerinin ön yüzünü (facade) görür ve kullanırlar.



Şekil 39. Vitrin Deseni UML Diyagramı

Bir sistem içindeki bir dizi ara yüze tek bir ara yüz üzerinden erişim sağlayacağımız program için bir dişi alt sistem kodunu yazalım;

```
#include <iostream>
using namespace std;
class SubSystem1 {
public:
    void operation11()  {
        cout << "Operation1 for subsystem1..." << endl;
    }
    void operation12()  {
        cout << "Operation2 for subsystem1..." << endl;
    }
    void operation13()  {
        cout << "Operation3 for subsystem1..." << endl;
    }
};
class SubSystem2 {
public:
    void operation21()  {
        cout << "Operation1 for subsystem2..." << endl;
    }
    void operation22()  {
        cout << "Operation2 for subsystem2..." << endl;
    }
    void operation23()  {
        cout << "Operation3 for subsystem2..." << endl;
    }
};
class SubSystem3 {
public:
    void operation31()  {
        cout << "Operation1 for subsystem3..." << endl;
    }
    void operation32()  {
        cout << "Operation2 for subsystem3..." << endl;
    }
    void operation33()  {
        cout << "Operation3 for subsystem3..." << endl;
    }
};
```

Şimdi de `Facade` sınıfını kodlayalım;

```
class Facade {
private:
```

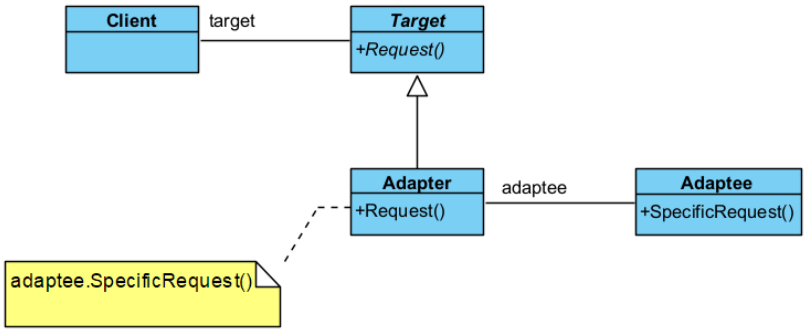
```
  SubSystem1 *subsystem1;
  SubSystem2 *subsystem2;
  SubSystem3 *subsystem3;
public:
  Facade() {
    this->subsystem1 = new SubSystem1();
    this->subsystem2 = new SubSystem2();
    this->subsystem3 = new SubSystem3();
  }
  ~Facade() {
    delete subsystem1;
    delete subsystem2;
    delete subsystem3;
  }
  void operation() {
    this->subsystem1->operation11();
    this->subsystem2->operation22();
    this->subsystem3->operation31();
    this->subsystem3->operation33();
    this->subsystem3->operation32();
  }
};
```

Son olarak istemci kodumuzu yazalım;

```
int main() { // Client
    Facade* facade = new Facade();
    facade->operation();
    delete facade;
}
/*Program Çalışdığında:
Operation1 for subsystem1...
Operation2 for subsystem2...
Operation1 for subsystem3...
Operation3 for subsystem3...
Operation2 for subsystem3...

...Program finished with exit code 0
*/
```

Adaptör

Adaptör deseni (adapter pattern), yabancı bir ara yüzü istemcinin anlayacağı bir ara yüze dönüştürür. Yani farklı ara yüzlere sahip sınıfların, iletişim ve etkileşim kurabilecekleri ortak bir nesne oluşturarak birlikte çalışmalarına izin verir. Bu desen, yabancı ara yüze yaptırılacak işin bir şekilde yapıldığı ve zaten yapılan bu işlemin istemcinin isteğine uygun hale getirildiği düşünülmelidir.



Şekil 40. Adaptör Deseni UML Diyagramı

Aşağıdaki örnekte `Target` nesnesi, istemci tarafından gelen isteği, bu işi yapabilme yeteneğine sahip `Adaptee` nesnesine yaptıracaktır. İşte burada `Adapter` nesnesi devreye girerek `Target` nesnesine gelen istekleri, `Adaptee` nesnesinin yapabileceği şekilde uyarlar.

İlk önce yabancı ara yüzü temsil eden `Adaptee` sınıfını kodluyoruz;

```
#include <iostream>
using namespace std;
class Adaptee {
public:
    void specificRequest() {
        cout << "Adaptee.SpecificRequest() yöntemi çağrıldı." << endl;
    }
};
```

Daha sonra soyut olan `Target` ve somut olan `Adapter` sınıfını kodluyoruz;

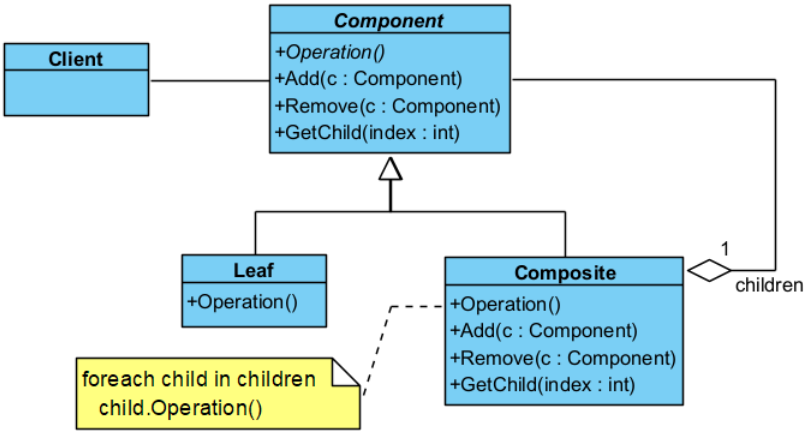
```
class Target {
public:
    virtual void request()=0;
};
class Adapter: public Target {
private:
    Adaptee* adaptee;
public:
    Adapter() {
        adaptee= new Adaptee();
    }
    ~Adapter() {
        delete adaptee;
    }
    void request() override {
        adaptee->specificRequest();
    };
};
```

Son olarak istemci tarafını kodluyoruz;

```
int main() { // Client
    Target* target = new Adapter();
    target->request();
    delete target;
}
/* Program Çıktısı:
Adaptee.SpecificRequest() yöntemi çağrıldı.

...Program finished with exit code 0
*/
```

Bileşik

Bileşik deseni (composite pattern), özyinelemeli (recursive) ya da hiyerarşik yapıya sahip nesne oluşturmak gerektiğinde yapısal olarak nasıl bir kodlama yöntemi izleneceğini söyler. Bu desen, her nesnenin bağımsız olarak veya aynı ara yüz üzerinden iç içe geçmiş nesneler kümesi olarak ele alınabileceği nesne hiyerarşilerinin oluşturulmasını kolaylaştırır. Aşağıda verilen UML diyagramı incelendiğinde; `Leaf`, ağaç yapısı içinde kullanılabilecek, parçalara ayrılamayan en küçük bileşen olan en ilkel (primitive) yapıdır. `Component`, bu ilkel yapılardan oluşturulacak karmaşık `Composite` nesnesine ilkel yapıların eklenip çıkarılmasını sağlayacak bileşendir.



Şekil 41. Bileşik Deseni UML Diyagramı

Şimdi sanal Component sınıfı ile somut sınıflarını tanımlayalım;

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
class Component {
private:
    Component *ebeveyn;
public:
    void setEbeveyn(Component *pEbeveyn) {
        this->ebeveyn = pEbeveyn;
    }
    Component *getEbeveyn() const {
        return this->ebeveyn;
    }
    virtual bool IsComposite() const {
        return false;
    }
    virtual void addChild(Component *component) {}
    virtual void removeChild(Component *component) {}
    virtual string displayOperation() const = 0;
};
class Leaf : public Component {
public:
    string displayOperation() const override {
        return "Yaprak";
    }
};
class Composite : public Component {
private:
    list<Component*> cocuklar;
public:
    void addChild(Component* pComponent) override {
        this->cocuklar.push_back(pComponent);
        pComponent->setEbeveyn(this);
    }
    void removeChild(Component* pComponent) override {
        cocuklar.remove(pComponent);
        pComponent->setEbeveyn(nullptr);
    }
    bool IsComposite() const override {
        return true;
    }
    string displayOperation() const override {
```

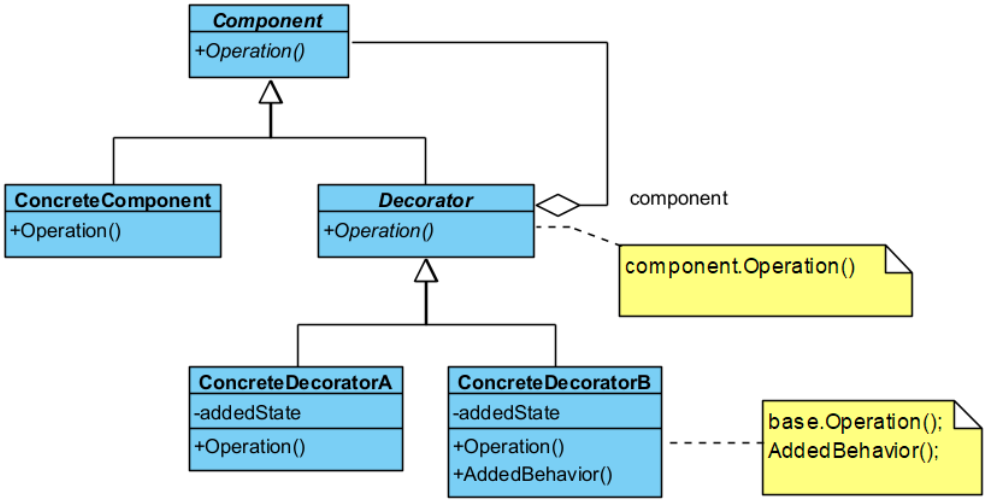
```
        string result;
        for (const Component *iterator : cocuklar) {
            if (iterator == cocuklar.back())
                result += iterator->displayOperation();
            else
            result += iterator->displayOperation() + "+";
        }
        return "Dal(" + result + ")";
    }
};
```

Şimdi istemci kodunu yazabiliriz;

```
int main() { // Client
    Component* yaprak = new Leaf;
    cout << "Sadece Yapraktan Oluşan Bileşik Nesne:" << endl;
    cout << yaprak->displayOperation();
    cout << endl;
    Component* kok = new Composite;
    Component* dal1 = new Composite;
    kok->addChild(dal1);
    Component *yaprak1 = new Leaf;
    dal1->addChild(yaprak1);
    Component *yaprak2 = new Leaf;
    dal1->addChild(yaprak2);
    cout << "Yapraktan ve Dallardan Oluşan Bileşik Nesne:" << endl;
    cout << kok->displayOperation();
    cout << endl;
    dal1->removeChild(yaprak2);
    cout << "Bir yaprağı Silinmiş ve Dallardan Oluşan Bileşik Nesne:" << endl;
    cout << kok->displayOperation();
    cout << endl;
    Component *dal2 = new Composite;
    kok->addChild(dal2);
    Component *yaprak3 = new Leaf;
    dal2->addChild(yaprak3);
    cout << "Bir dal daha eklenmiş ve Dallardan Oluşan Bileşik Nesne:" << endl;
    cout << kok->displayOperation();
    delete yaprak;
    delete kok;
    delete dal1;
    delete dal2;
    delete yaprak1;
    delete yaprak2;
    delete yaprak3;
}
/*Program Çıktısı:
Sadece Yapraktan Oluşan Bileşik Nesne:
Yaprak
Yapraktan ve Dallardan Oluşan Bileşik Nesne:
Dal(Dal(Yaprak+Yaprak))
Bir yaprağı Silinmiş ve Dallardan Oluşan Bileşik Nesne:
Dal(Dal(Yaprak))
Bir dal daha eklenmiş ve Dallardan Oluşan Bileşik Nesne:
Dal(Dal(Yaprak)+Dal(Yaprak))

...Program finished with exit code 0
*/
```

Dekoratör

Dekoratör deseninde (decorator pattern), bir nesneye dinamik olarak yeni durum ve davranışları eklemek mümkündür. Yani çalıştırma anında, bir nesnenin sahip olduğu yeteneklere, yeni yeteneklerin eklenmesini sağlar.



Şekil 42. Dekoratör Deseni UML Diyagramı

UML diyagramı incelendiğinde, `Component` nesnesi, `Decorator` nesnesi aracılığı ile çalıştırma anında (run time), `addedBehavior` yöntemine ve `addedState` durumuna (state) sahip olur.

İlk olarak sanal sınıf olan `Component` ve somut `ConcreteComponent` bileşenlerini kodlayalım;

```
#include <iostream>
using namespace std;
class Component {
protected:
    int state;
public:
    Component(int pState):state(pState) {
    }
    int getState() {
        return state;
    }
    void setState(int pState) {
        state=pState;
    }
    virtual void operation() =0;
};
class ConcreteComponent: public Component {
public:
    ConcreteComponent(int pState):Component(pState) {
    }
    void operation() override {
        cout << "Somut ürünün:" << endl
             << "state adında bir durumu var:" << state << endl;
    }
};
```

Şimdi de somut `Decorator` sınıfı ve alt sınıflarını kodlayalım;

```
class Decorator: public Component {
public:
    Decorator(int pState, Component* pComponent):Component(pState) {
        component=pComponent;
    }
```

```
    void operation() override {
        component->operation();
    }
protected:
    Component* component; // aggregation to Component
};
class ConcreteDecorator1: public Decorator {
public:
    ConcreteDecorator1(int pState, Component* pComponent):Decorator(pState, pComponent) {
        addedState=pState*100;
    }
    void operation() override {
       cout << "Decorator 1" << endl
            << "addedState adında yeni bir durumu var:" << addedState << endl;
    }
    int getAddedState() {
        return addedState;
    }
    void setAddedState(int pState) {
        addedState=pState;
    }
private:
    int addedState;
};
class ConcreteDecorator2: public Decorator {
public:
    ConcreteDecorator2(int pState, Component* pComponent):Decorator(pState, pComponent) {
        addedState=pState*200;
    }
    void operation() override {
        cout << "Decorator 2" << endl
             << "addedState adında yeni bir durumu var:" << addedState << endl;
        cout << "addedBehavior Davranışı Var:"
             << addedBehavior() << endl;
    }
    string addedBehavior() {
       return "Bu metin Yeni Davranışdan Geliyor...";
    }
    int getAddedState() {
        return addedState;
    }
    void setAddedState(int pState) {
        addedState=pState;
    }
private:
    int addedState;
};
```

Son olarak istemcimizi kodluyoruz;

```
int main() { // Client
    Component* component = new ConcreteComponent(1);
    component->operation();

    Component* decoratoratedComponent1 = new ConcreteDecorator1(2,component);
    decoratoratedComponent1->operation();

    Component* decoratoratedComponent2 = new ConcreteDecorator2(3,component);
    decoratoratedComponent2->operation();
    delete component,decoratoratedComponent1,decoratoratedComponent2;
}
/* Program Çıktısı:
```

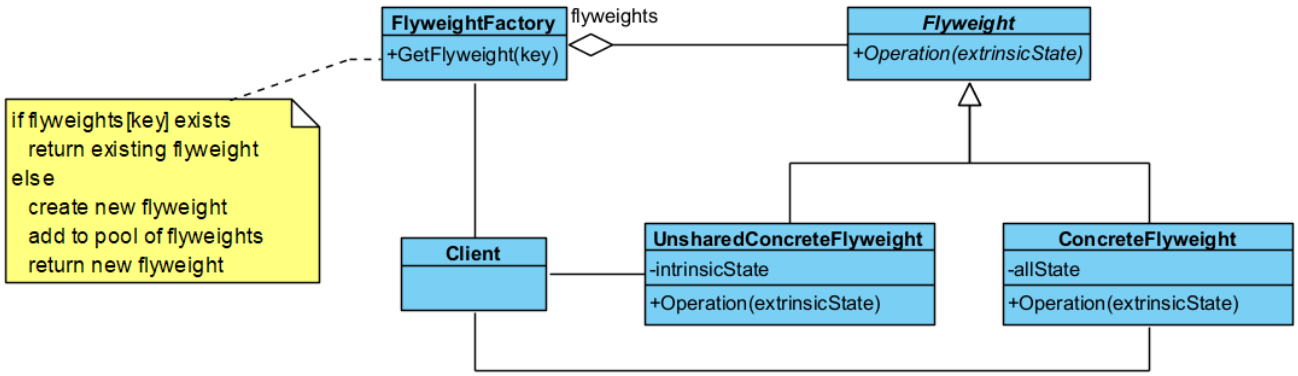
```
Somut ürünün:
state adında bir durumu var:1
Decorator 1
addedState adında yeni bir durumu var:200
Decorator 2
addedState adında yeni bir durumu var:600
addedBehavior Davranışı Var:Bu metin Yeni Davranışdan Geliyor...

...Program finished with exit code 0
*/
```

Sineksıklet

Sineksıklet deseninde (flyweight pattern), birçok küçük nesneden oluşan bir sistemi paylaşarak kullanma hedeflenmiştir. Temel olarak sineksıklet, paylaşılan bir nesnedir ve eş zamanlı olarak küçük nesneleri kullandırır. Yani sineksıklet nesnesi sözü geçen her bir küçük nesne gibi davranır.

Bu desen, bir metin editörü ile açıklanabilir. Metin editöründe yazılacak her bir harf, font bilgisine ve büyüklüğe sahiptir. İşte buradaki her bir harf ile kastedilen, sineksıklet deseninde konu olan küçük nesnelerdir. Bu nesneler bir araya gelerek satırları ve kolonları oluşturur. Sineksıklet nesnesi ise yerine göre bu harflerin her birinin yerine geçer.

Yukarıdaki UML dokümanı incelendiğinde; `FlyweightFactory` küçük nesnelerden büyük resmi oluşturan fabrikayı temsil etmektedir. Birçok uygulamada durumlar geçicidir (extrinsic state) ve bu geçici durumlar saklanmazlar. Geçici durumlarla yapılan işlemler sonunda, kalıcı durumlar (intrinsic state) olur ve nesnelerde saklanır. Geçici durumlarla şekillenen ortak bir nesne (shared object) tanımıyla küçük nesneler tanımlanır. Bu küçük nesnelerin her biri sineksıklet (flyweight) olarak adlandırılır. Ortak özellik taşımayan nesneler (unshared object) de büyük resim içinde yer alabilir. Bu nesneler de ortaklık dışı sineksıklet (unshared flyweight) olarak adlandırılır. Fabrika geçici durumlarla şekillenen sineksıklet nesneler üretir ve büyük resmi oluşturur.



Şekil 43. Sineksıklet Deseni UML Diyagramı

Sineksıklet nesneleri ortak kullanıldığından, uygulama nesnenin kimliğine bağımlı değildir. Bu desen çok sık kullanılmaz ancak aşağıdaki durumların çoğu varsa kullanılabilir;

- Bir uygulama geniş çaplı olup çok miktarda nesne içeriyorsa,
- Nesneleri durumlarıyla olduğu gibi saklamanın maliyeti yüksekse,
- Nesnelerin birçok durumu geçici ve saklanmasına gerek yoksa,
- Birçok nesneden oluşan bir nesne grubu, ortak kullanılan bir nesnedeki geçici değişime bağlı olarak değişebiliyorsa,

İlk önce soyut `Flyweight` sınıfı ve somut sınıflarını kodlayalım;

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
```

```
class Flyweight {
protected:
    string state;
public:
    Flyweight(string pState):state(pState) {
    }
    string getState() {
        return state;
    }
    void setState(int pState) {
        state=pState;
    }
    virtual void operation(string extrinsicState) =0;
};
class UnsharedConcreteFlyweight: public Flyweight {
protected:
    string intrinsicState;
public:
    UnsharedConcreteFlyweight(string pState): Flyweight(pState) {
        intrinsicState=pState+"+Ek özelliker";
    }
    void operation(string extrinsicState) {
        cout << "Somut Ortak Özellik TAŞIMAYAN Sinetsıklet:"
             << "Durumu:" << state << endl
             << "Geçici Durum:" << extrinsicState << endl
             << "Ortak Olmayan Durum:" << intrinsicState << endl;
    }
};
class ConcreteFlyweight: public Flyweight  {
public:
    ConcreteFlyweight(string pState): Flyweight(pState) {
    }
    void operation(string extrinsicState) {
        cout << "Somut Ortak Özellik Taşıyan Sinetsıklet:"
             << "Durumu:" << state << endl
             << "Geçici Durum:" << extrinsicState << endl;
    }
};
```

Şimdi de `FlyweightFactory` sınıfını kodlayalım;

```
class FlyweightFactory {
private:
    list<Flyweight*> flyweitghts; //Aggregation to Flyweight
public:
    FlyweightFactory(std::initializer_list<Flyweight*> list): flyweitghts(list) {
    }
    Flyweight* getFlyweitght(Flyweight* pFlyweight) {
        list<Flyweight*>::iterator iter = find(flyweitghts.begin(),
                                               flyweitghts.end(),
                                               pFlyweight);
        if (iter!=flyweitghts.end())
            return *iter;
        else {
            flyweitghts.push_back(pFlyweight);
            return pFlyweight;
        }
    }
    void showFlyweights() const
    {
        size_t count = flyweitghts.size();
        cout << "FlyweightFactory: " << count << " kadar flyweight sahibidir:" << endl;
```

```
        for (Flyweight* item : flyweitghts)
            item->operation("+Geçici Durumlar...");
    }
};
```

Son olarak de istemci sınıfı yazalım;

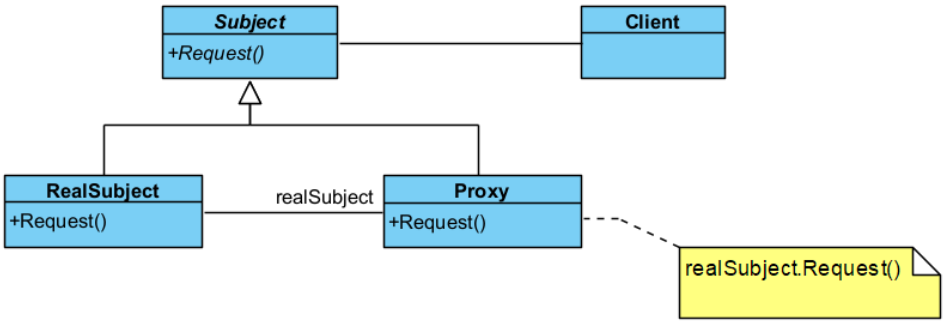
```
int main() { // Client
    Flyweight* ortakOlanSineksiklet1=new ConcreteFlyweight("Binek Araç 1");
    Flyweight* ortakOlanSineksiklet2=new ConcreteFlyweight("Binek Araç 2");
    Flyweight* ortakOlanSineksiklet3=new ConcreteFlyweight("Binek Araç 3");
    Flyweight* ortakOlmayanSineksiklet1=new UnsharedConcreteFlyweight("Kamyonet 1");
    Flyweight* ortakOlmayanSineksiklet2=new UnsharedConcreteFlyweight("Çekici 1");

    FlyweightFactory* fabrika1=new FlyweightFactory({ortakOlanSineksiklet1,
                                                     ortakOlanSineksiklet2,
                                                     ortakOlanSineksiklet3,
                                                     ortakOlmayanSineksiklet1});
    fabrika1->showFlyweights();

    FlyweightFactory* fabrika2=new FlyweightFactory({});
    fabrika2->getFlyweitght(ortakOlanSineksiklet1);
    fabrika2->getFlyweitght(ortakOlmayanSineksiklet2);
    fabrika2->getFlyweitght(ortakOlanSineksiklet1);
    fabrika2->showFlyweights();

    delete ortakOlanSineksiklet1,ortakOlanSineksiklet2,ortakOlanSineksiklet3;
    delete ortakOlmayanSineksiklet1,ortakOlmayanSineksiklet2;
    delete fabrika1,fabrika2;
}
/*Program Çalıştığında:
FlyweightFactory: 4 kadar flyweight sahibidir:
Somut Ortak Özellik Taşıyan Sinetsıklet:Durumu:Binek Araç 1
Geçici Durum:+Geçici Durumlar...
Somut Ortak Özellik Taşıyan Sinetsıklet:Durumu:Binek Araç 2
Geçici Durum:+Geçici Durumlar...
Somut Ortak Özellik Taşıyan Sinetsıklet:Durumu:Binek Araç 3
Geçici Durum:+Geçici Durumlar...
Somut Ortak Özellik TAŞIMAYAN Sinetsıklet:Durumu:Kamyonet 1
Geçici Durum:+Geçici Durumlar...
Ortak Olmayan Durum:Kamyonet 1+Ek özelliker
FlyweightFactory: 2 kadar flyweight sahibidir:
Somut Ortak Özellik Taşıyan Sinetsıklet:Durumu:Binek Araç 1
Geçici Durum:+Geçici Durumlar...
Somut Ortak Özellik TAŞIMAYAN Sinetsıklet:Durumu:Çekici 1
Geçici Durum:+Geçici Durumlar...
Ortak Olmayan Durum:Çekici 1+Ek özelliker

...Program finished with exit code 0
*/
```

Vekil

Vekil deseni (proxy pattern), kullanılacak ve erişilecek bir nesnenin yerine geçen bir başka nesneye ihtiyaç duyulduğunda kullanılır. İstemci (client), nesnenin doğrudan kendisine değil, vekil (proxy) nesneye ulaşır ve bu nesneye isteklerini iletir. Kısaca elinden her iş gelen veya diğer nesneler hakkında bilgi sahibi olan bir nesneye ihtiyaç duyulduğunda kullanılır;

– Bir nesne, bulunduğu yerden uzaktaki bir nesneyi kullanacaksa uzak vekil (remote proxy) kullanılır.

- Bir nesneye başka nesneler tarafından erişim kısıtlanacaksa koruma vekili (protection proxy) olarak adlandırılır.
- Bir nesneye erişim sırasında başka işlemler yapılacaksa vekil, akıllı referans (smart reference) olarak adlandırılır. Böyle durumda vekil, nesneye bir erişim yapıldığında, diğer nesnelerin erişmemesi için kullanılacak nesneye kilit koyabilir. Akıllı referans, akıllı gösterici (smart pointer) olarak da adlandırılır.



Şekil 44. Vekil Deseni UML Diyagramı

UML diyagramındaki `RealSubject`, bazı temel iş mantıklarını içerir. Genellikle, giriş verilerini düzeltme gibi hassas olabilen bazı yararlı işler yapma yeteneğine sahiptir. Bir `Proxy`, `RealSubject` kodunda herhangi bir değişiklik yapmadan bu sorunları çözebilir.

İstemci kodunun hem `RealSubject` hem de `Proxy` çalışabilmesi için tüm nesnelerle `Subject` ara yüzü üzerinden çalışması gerekir. Ancak gerçek hayatta, istemciler çoğunlukla `RealSubject` ile doğrudan çalışır. Bu durumda, deseni daha kolay uygulamak için `Proxy`, `RealSubject` sınıfından genişletebilirsiniz.

Şimdi `Subject` sanal sınıfı ve somut sınıflarını kodlayalım;

```
#include <iostream>
using namespace std;

class Subject {
public:
    virtual void request() const = 0;
};
class RealSubject : public Subject {
private:
    int state;
public:
    RealSubject(int pState):state(pState) {
    }
    void request() const override {
        cout << "RealSubject nesnesine gelen istek işleniyor..." << endl;
        cout << "RealSubject state: " << state << endl;
    }
};
class Proxy : public Subject {
private:
    bool CheckAccess() const {
        cout << "Proxy, realSubject nesnesinin erişimini kontrol ediyor..." << endl;
        return true;
    }
    void LogAccess() const {
        cout << "Proxy, gelen talebin iz kaydını tutuyor..." << endl;
    }
public:
    RealSubject *realSubject;
    Proxy()  {
        // RealSubject sınıfından bir örnek imal edip erişiyor.
        realSubject=new RealSubject(20);
```

```
    }
    Proxy(RealSubject* pRealSubject)  {
        // RealSubject sınıfından mevcut bir örneğin kopyası imal edip erişiyor.
        realSubject=new RealSubject(*pRealSubject);
    }
    ~Proxy() {
        delete realSubject;
    }
    void request() const override {
        if (this->CheckAccess()) {
            this->realSubject->request();
            this->LogAccess();
        }
    }
};
```

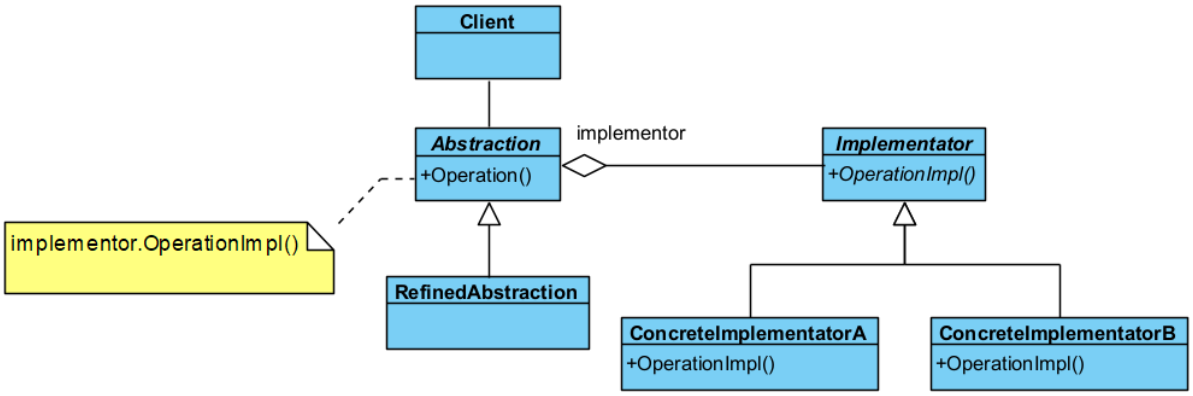
Son olarak istemci kodunu kodlayalım;

```
int main() { //Client-İstemci
    cout << "Client: gerçek özne nesnesine istek gönderiyor:" << endl;
    RealSubject *gercekOzne = new RealSubject(10);
    gercekOzne->request();
    cout << endl;

    cout << "Client: RealSubject sınıfından bir örnek imal ederek kullanıyor:" << endl;
    Proxy *vekil1 = new Proxy();
    vekil1->request();
    cout << endl;

    cout << "Client: gercekOzne nesnesinin kopyasını imal ederek kullanıyor:" << endl;
    Proxy *vekil2 = new Proxy(gercekOzne);
    vekil2->request();

    delete gercekOzne,vekil1,vekil2;
}
/* Program Çıktısı:
Client: gerçek özne nesnesine istek gönderiyor:
RealSubject nesnesine gelen istek işleniyor...
RealSubject state: 10

Client: RealSubject sınıfından bir örnek imal ederek kullanıyor:
Proxy, realSubject nesnesinin erişimini kontrol ediyor...
RealSubject nesnesine gelen istek işleniyor...
RealSubject state: 20
Proxy, gelen talebin iz kaydını tutuyor...

Client: gercekOzne nesnesinin kopyasını imal ederek kullanıyor:
Proxy, realSubject nesnesinin erişimini kontrol ediyor...
RealSubject nesnesine gelen istek işleniyor...
RealSubject state: 10
Proxy, gelen talebin iz kaydını tutuyor...

...Program finished with exit code 0
*/
```

Köprü

Köprü deseni (bridge pattern), işi yapan alt yüklenici veya taşeron (implementator) nesne ile işi yaptıran nesne (client) arasındaki bağımlılığı soyutlama ile ortadan kaldırır. Yani birbiriyle oldukça iç içe olan kodlama ile soyutlamayı ortak ara yüz ile birbirinden uzaklaştırır. Sistemin genişlemesine izin

verir. Bu desen, yazılımlarda oldukça çok kullanılan bir desendi. Çünkü taşeronlar değişse de yeni taşeron eklense de istemci tarafında bir şey değişmez. Köprü deseni aşağıdaki durumlarda kullanılır;



Şekil 45. Köprü Deseni UML Diyagramı

- Taşeron nesneye kalıcı bir bağımlılık istenmediği durumlar. Çalıştırma anında (run time) taşeronlar arasında geçiş yapmayı sağlar.
- Birden çok taşeron kullanarak sistemin genişlemesine ihtiyaç olduğu durumlar.
- Taşeronlar üzerindeki değişiklikler, istemciyi etkilemez. İstemci tarafında derleme olmadan yazılımın değiştirilmesi sağlanır.
- Projede aynı anda farklı birden çok taşeron tasarımı yapılabildiğinden, proje parçalara ayrılarak kendi içinde hızla sınıf tasarımı yapılabilir. Bu durum *Rumbaugh* tarafından iç içe genelleştirme (nested generalization) olarak adlandırılmıştır.

İlk önce taşeron olan soyut `Implementator` sınıfı ve somut alt sınıflarını tanımlayalım;

```
#include <iostream>
using namespace std;

class Implementator { // Taşeron arayüzü
public:
    virtual void operationImpl()=0;
};
class ConcreteImplementatorA: public Implementator{  // A Taşeronu
public:
    void operationImpl() override {
        cout << "A Taşeronu tarafından yapılan iş:..." << endl;
    };
};
class ConcreteImplementatorB: public Implementator{ // B Taşeronu
public:
    void operationImpl() override {
        cout << "B Taşeronu tarafından yapılan iş:..." << endl;
    };
};
```

Şimdi de yüklenici olan `Abstraction` sınıfını ve alt somut sınıfını kodlayalım;

```
class Abstraction { // soyut yüklenici
private:
    Implementator* implementor;
public:
    Abstraction(Implementator* pImplenmentor): implementor(pImplenmentor) {
    }
    void setImplementor(Implementator* pImplenmentor) {
        implementor=pImplenmentor;
    }
    Implementator* getImplementor() {
        return implementor;
```

```
    }
    void operation() {
        cout << "Yüklenici Taşerona iş yaptırıyor:" << endl;
        implementor->operationImpl();
    };
};
class RefinedAbstraction: public Abstraction { // somut yüklenici
public:
    RefinedAbstraction(Implementator* pImplenmentor): Abstraction(pImplenmentor) {
    }
};
```

Son olarak istemi tarafını kodlayalım;

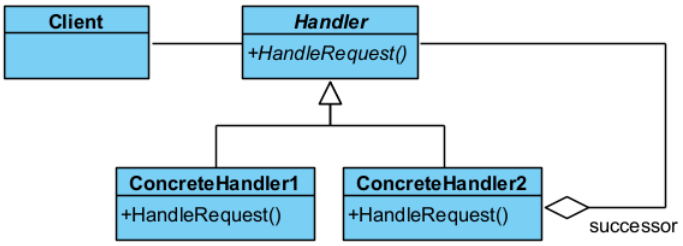
```
int main() { // İstemi-Client
    ConcreteImplementatorA* teseron1=new ConcreteImplementatorA();
    RefinedAbstraction* yuklenici=new RefinedAbstraction(teseron1);
    yuklenici->operation();

    ConcreteImplementatorB* teseron2=new ConcreteImplementatorB();
    yuklenici->setImplementor(teseron2); // çalıştırma anında taşeron değiştiriliyor.
    yuklenici->operation();

    delete teseron2, teseron1, yuklenici;
}
/* Program Çıktısı:
Yüklenici Taşerona iş yaptırıyor:
A Taşeronu tarafından yapılan iş:...
Yüklenici Taşerona iş yaptırıyor:
B Taşeronu tarafından yapılan iş:...

...Program finished with exit code 0
*/
```

Davranışla İlgili Desenler

Davranışla ilgili desenler (behavioral patterns) nesnelerin çeşitli olaylar karşısında gösterecekleri davranışlara çözüm getirmektedirler.

Sorumluluk Zinciri

Bazı durumlarda bir nesne, kendinin yapmayacağı bir işlemi halefine (successor) devredebilir. Ya da nesnenin kendi sorumluluğunu aşan durumlarda ilgili diğer nesneye işlem yaptırılır. İşte bu durumda sorumluluk zinciri deseni (chain of responsibility pattern) kullanılır.



Şekil 46. Sorumluluk Zinciri Deseni UML Diyagramı

Şimdi soyut `Handler` sınıfı ve somut sınıflarını kodlayalım;

```
#include <iostream>
using namespace std;
class Handler {
public:
    virtual void handleRequest(int pRequest)=0;
```

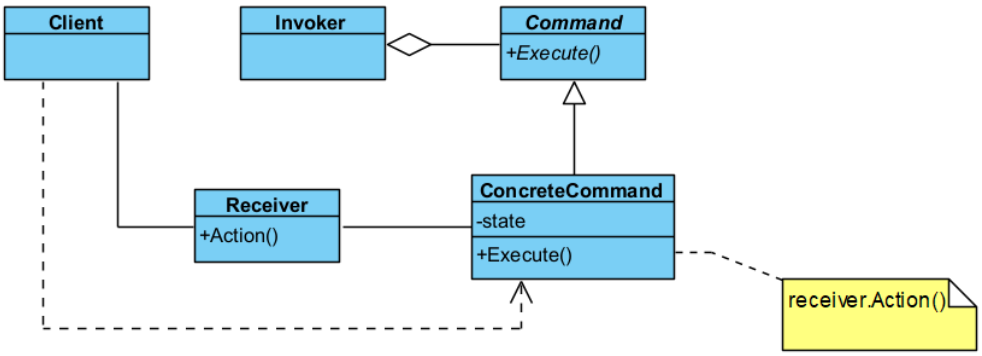
```
};
class ConcreteHandler1:public Handler {
public:
    void handleRequest(int pRequest) override {
        cout << "Yetki Devretmeyen ConcreteHandler1:" << endl;
        cout << pRequest << " ile gelen isteğin tamamını yerine getirdi." << endl;
    }
};
class ConcreteHandler2:public Handler {
private:
    Handler* successor;
public:
    //successor public hale getiriliyor: aggregatotion to Handler
    void setSuccessor(Handler* pSuccessor) {
        successor=pSuccessor;
    }
    Handler* getSuccessor() {
        return successor;
    }
    ConcreteHandler2(Handler* pSuccessor): successor(pSuccessor) {
    }
    void handleRequest(int pRequest) override {
        if (pRequest <100) {
            cout << "Yetki Devredebilen ConcreteHandler2:" << endl;
            cout << pRequest << " ile gelen isteğin tamamını yerine getirdi." << endl;
        } else  {
            cout << "Yetki Devredebilen ConcreteHandler2:" << endl;
            cout << pRequest << " ile gelen iyetkisini halefine devretti:" << endl;
            successor->handleRequest(pRequest);
        }
    }
};
```

Son olarak istemci kısmını kodlayalım;

```
int main() { //İstemci-Client
    Handler* yetkilimemur=new ConcreteHandler1();
    yetkilimemur->handleRequest(100);

    Handler* yetkisizmemur=new ConcreteHandler2(yetkilimemur);
    yetkisizmemur->handleRequest(50);
    yetkisizmemur->handleRequest(200);

    delete yetkilimemur,yetkisizmemur;
}
/* Program Çıktısı:
Yetki Devretmeyen ConcreteHandler1:
100 ile gelen isteğin tamamını yerine getirdi.
Yetki Devredebilen ConcreteHandler2:
50 ile gelen isteğin tamamını yerine getirdi.
Yetki Devredebilen ConcreteHandler2:
200 ile gelen iyetkisini halefine devretti:
Yetki Devretmeyen ConcreteHandler1:
200 ile gelen isteğin tamamını yerine getirdi.

...Program finished with exit code 0
*/
```

Komut

Bazı durumlarda bir nesne diğer bir nesneye ileti (message) verip bir işlem başlattığında, bu işlem bitmeden başka işlemler yapmak isteyebilir. İşte bu tür durumlarda komut deseni (command pattern) kullanılır.



Şekil 47. Komut Deseni UML Diyagramı

Bu desende istemcinin istekleri, `Invoker` nesnesi tarafından kuyruğa sokulur ve kuyruğa sokulan adımların her biri `Command` nesnesi tarafından asıl işi yapan `Receiver` nesnesine yaptırılır. Bu durum bize geri alma (undo) işlemi yapmamızı sağlar.

İlk önce asıl işi yapacak `Receiver` sınıfını kodluyoruz;

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
class Receiver {
public:
    void action(string pParam){
        cout << pParam << " komutu işlemleri yapılıyor..." <<endl;
    }
    void actionA(string pParam){
        cout << pParam << " komutu için A işlemleri yapılıyor"  <<endl;
    }
    void actionB(string pParam){
        cout << pParam << " komutu için B işlemleri yapılıyor"  <<endl;
    }
};
```

Sonrasında her bir komutu tutacak soyut `Command` sınıfı ve somut sınıflarını yazıyoruz;

```
class Command {
public:
    virtual void execute()=0;
};
class SimpleConcreteCommand: public Command {
private:
    string state;
    Receiver* receiver; // private association to Receiver
public:
    SimpleConcreteCommand(string pState):state(pState) {
        receiver=new Receiver();
    }
    ~SimpleConcreteCommand() {
        delete receiver;
    }
    void execute() override {
        receiver->action(state);
```

```
    };
};
class ComplexConcreteCommand: public Command {
private:
    string state;
    Receiver* receiver; // private association to Receiver
public:
    ComplexConcreteCommand(string pState):state(pState) {
        receiver=new Receiver();
    }
    ~ComplexConcreteCommand() {
        delete receiver;
    }
    void execute() override {
        receiver->actionA(state);
        receiver->actionB(state);
    };
};
```

Daha sonra ihtiyaç olursa birden fazla komut tutan ve çalıştıran `Invoker` sınıfını tanımlıyoruz;

```
class Invoker {
private:
    list<Command*> commands;
public:
    Invoker(std::initializer_list<Command*> pCommandList): commands(pCommandList) {
    }
    void addCommand(Command* pCommmand)  {
        this->commands.push_back(pCommmand);
    }
    void doCommands() {
        for (Command* command : commands)
            command->execute();
    }
};
```

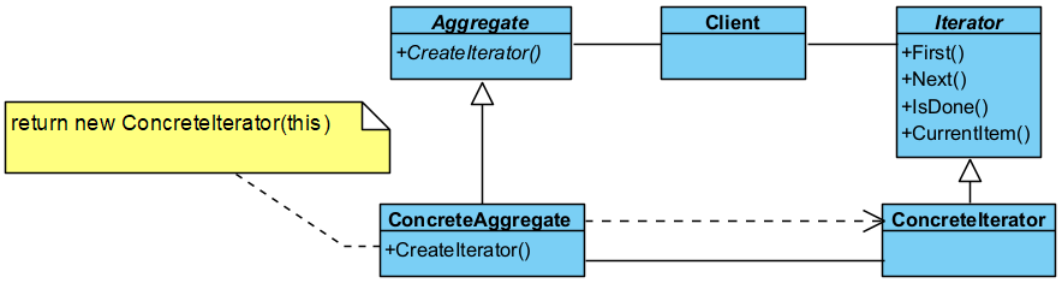
Son olarak istemci kodunu yazıyoruz;

```
int main() { // Client-İstemci
    Command* simpleCommand=new SimpleConcreteCommand("Yaz");
    simpleCommand->execute();

    Command* complexCommand=new ComplexConcreteCommand("Hem Dosyaya Hem Yazıcıya Yaz");
    Invoker* invoker=new Invoker({complexCommand,simpleCommand});
    invoker->doCommands();

    delete simpleCommand,complexCommand,invoker;
}
/* Program çalıştığında;
Yaz komutu işlemleri yapılıyor...
Hem Dosyaya Hem Yazıcıya Yaz komutu için A işlemleri yapılıyor
Hem Dosyaya Hem Yazıcıya Yaz komutu için B işlemleri yapılıyor
Yaz komutu işlemleri yapılıyor...

...Program finished with exit code 0
*/
```

Yineleyici

Yineleyici deseninde (iterator pattern), birden fazla elemandan oluşan küme (aggregate) içindeki her bir elemana sırayla erişim (iteration) sağlar. Erişim işleminde, istemcinin kümenin nasıl yapılandırıldığı konusunda bilgi sahibi olması gerekmez. Bu desende, istemci tarafından veri kümesi soyut (abstract)

olarak kullanılmış olur. Yani istemciye, veri yapısından (data structure) bağımsız bir şekilde verilere erişim sağlayan bir ara yüz (interface) sunulur. Aslında standart şablon kütüphanesindeki (Standart Template Library) yineleyiciler bu deseni kullanır.



Şekil 48. Yineleyici Deseni UML Diyagramı

Burada `Aggregate` ve `Iterator` sınıflarını ilk önce tanımlıyoruz;

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
class Iterator; // Böyle bir sınıf tanımlanacak!
class Aggregate {  // double veri tipinde değerlar tutan küme
public:
    virtual void addItem(double pItem)=0;
    virtual int size()=0;
    virtual double getItem(int index)=0;
    virtual Iterator* createIterator()=0;
};
class Iterator {
public:
    virtual double first()=0;
    virtual double next()=0;
    virtual bool isDone()=0;
    virtual double currentItem()=0;
};
class ConcreteIterator:public Iterator {
private:
    Aggregate* aggregate;
    int current;
public:
    ConcreteIterator(Aggregate* pAggregate): aggregate(pAggregate) {
        current=0;
    }
    double first() override {
        current=0;
        return aggregate->getItem(current);
    }
    double next() override {
        current++;
        return aggregate->getItem(current);
    }
    bool isDone() override {
        return (current< aggregate->size()) ? true : false;
    }
    double currentItem() override {
        return aggregate->getItem(current);
    }
};
class ConcreteAggregate : public Aggregate {
private:
```

```
    vector<double> items;
public:
    void addItem(double pItem) override {
        items.push_back(pItem);
    }
    int size() override {
        return items.size();
    }
    double getItem(int index) override {
        return items[index];
    }
    Iterator* createIterator() override {
        return new ConcreteIterator(this);
    }
};
```

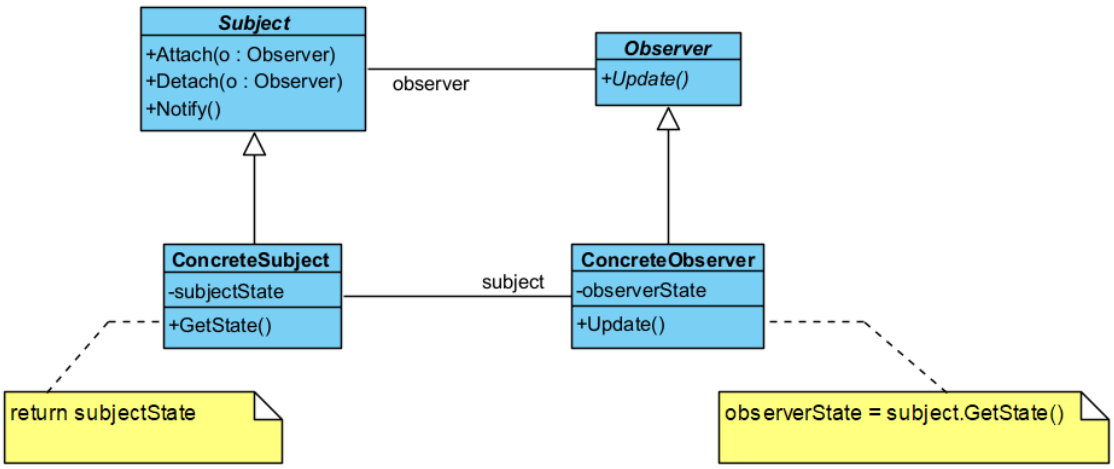
Şimdi de istemci kısmını kodluyoruz;

```
int main() { // İstemci-Client
    ConcreteAggregate* aggregate = new ConcreteAggregate();
    aggregate->addItem(10.0);
    aggregate->addItem(20.0);
    aggregate->addItem(30.0);

    Iterator* iterator = aggregate->createIterator();

    cout << "Aggregate nesneleri:" << endl;
    double item = iterator->first();
    if (iterator->isDone())
    while (iterator->isDone()){
        cout << item << endl;
        item = iterator->next();
    }

    delete aggregate, iterator;
}
/*
Aggregate nesneleri:
10
20
30

...Program finished with exit code 0
*/
```

Gözlemci

Gözlemci deseni (observer pattern), sistemdeki diğer nesnelerin durum değişiklikleri hakkında bir veya daha fazla nesnenin bilgilendirilmesini sağlar. Bağımlı olunan nesnenin durumu veya işin konusu (subject) değiştiğinde, bağımlı olan veya o işi izleyen gözlemcilerin (observer) kendileri de güncellenir. Bu desen aşağıdaki durumlarda kullanılır;

- Bir soyutlama (abstraction) sonucu ortaya çıkan bağımsız iki farklı yönün (aspect) ortaya çıktığı durumlarda, bu yönler birbirinden bağımsız olarak geliştirilebilir ve değiştirilebilir.
- Bir değişikliğin başka değişiklikleri gerektirdiği durumlar.
- Bir nesnenin, diğer nesnelerdeki değişiklikleri kendine vazife etmemesini sağlayan durumlar.



Şekil 49. Gözlemci Deseni UML Diyagramı

İlk önce sanal `Observer` sınıfını tanımlayalım;

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
class Observer {
public:
    virtual void update()=0;
};
```

Daha sonra sanal `Subject` ve somut `ConcreteSubject` sınıfını kodlayalım;

```
class Subject {
protected:
    list<Observer*> observers;
public:
    void attach(Observer* pObserver){
        observers.push_back(pObserver);
    }
    void detach(Observer* pObserver){
        observers.remove(pObserver);
    }
    void notify() {
        for (Observer* observer : observers)
            observer->update();
    }
};
class ConcreteSubject: public Subject {
protected:
    int subjectState;
public:
    ConcreteSubject() {
        subjectState=0;
    }
    int getState() {
        return subjectState;
    }
    void setState(int pSubjectState) {
        subjectState=pSubjectState;
    }
};
```

Daha sonra somut `ConcreteObserver` Sınıfını tanımlayalım;

```
class ConcreteObserver:public Observer {
private:
    string gozlemciAdi;
protected:
    ConcreteSubject* subject;
    int observerState;
public:
    ConcreteObserver(string pGozlemciAdi,ConcreteSubject* pConcreteSubject) {
        gozlemciAdi=pGozlemciAdi;
        subject=pConcreteSubject;
    }
    void update() override {
        observerState=subject->getState();
        cout << gozlemciAdi << " gözlemcisinin yeni durumu:" << observerState << endl;
    };
};
```

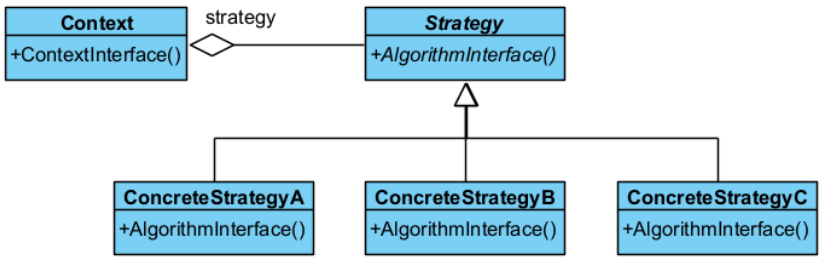
Son olarak istemci tarafını kodlayalım;

```
int main() { //İstemci
    ConcreteSubject* gozlemcileriDeğistirecek = new ConcreteSubject();
    Observer* observer1=new ConcreteObserver("A",gozlemcileriDeğistirecek);
    gozlemcileriDeğistirecek->attach(observer1);
    Observer* observer2=new ConcreteObserver("B",gozlemcileriDeğistirecek);
    gozlemcileriDeğistirecek->attach(observer2);
    Observer* observer3=new ConcreteObserver("C",gozlemcileriDeğistirecek);
    gozlemcileriDeğistirecek->attach(observer3);
    gozlemcileriDeğistirecek->setState(1);
    gozlemcileriDeğistirecek->notify();
    gozlemcileriDeğistirecek->setState(2);
    gozlemcileriDeğistirecek->notify();
    delete gozlemcileriDeğistirecek,observer1,observer2,observer3;
}
/*Program Çıktısı:
A gözlemcisinin yeni durumu:1
B gözlemcisinin yeni durumu:1
C gözlemcisinin yeni durumu:1
A gözlemcisinin yeni durumu:2
B gözlemcisinin yeni durumu:2
C gözlemcisinin yeni durumu:2

...Program finished with exit code 0
*/
```

Strateji

Strateji deseni (strategy pattern), belirli bir davranışı gerçekleştirmek için değiştirilebilen sarmalanmış (encapsulated) algoritmalar kümesini tanımlar. Bu desen aşağıdaki durumda kullanılır;

- Aralarında ilişki bulunan birçok sınıfın davranışlara bağlı olarak anlaşmazlığa düşmesini önlemek için,
- Birden çok algoritmanın olması durumunda,
- Algoritma, istemcinin bilmeyeceği bir veri yapısına sahip olduğu durumlarda,
- Bir sınıf kendi yöntemlerinde çok fazla sayıda duruma göre seçim (conditional choice) kullanıyorsa.



Şekil 50. Strateji Deseni UML Diyagramı

Bu desenin bağlam (Context) nesnesi, diğer tasarım desenlerinin biri ile iç içe kullanılması halinde, sıfır desen (null pattern) olarak adlandırılır. Sıfır desende, `Context` nesnesi akıllıdır ve her zaman ne yapacağını bilir.

İlk önce `Strategy` somut sınıfı ve alt sınıflarını kodlayalım;

```
#include <iostream>
using namespace std;
class Strategy{
public:
    virtual void algorithmInterface()=0;
};
class ConcreteStrategyA: public Strategy{
public:
    void algorithmInterface() override {
        cout << "A stratejisinin Algoritması Çalıştırılıyor.." << endl;
    }
};
class ConcreteStrategyB: public Strategy{
public:
    void algorithmInterface() override {
        cout << "B stratejisinin Algoritması Çalıştırılıyor.." << endl;
    }
};
class ConcreteStrategyC: public Strategy{
public:
    void algorithmInterface() override {
        cout << "C stratejisinin Algoritması Çalıştırılıyor.." << endl;
    }
};
```

Şimdi de `Context` sınıfını kodlayalım;

```
class Context {
private:
    Strategy* strategy;  // aggregation to Strategy
public:
    Context(Strategy* pStrategy):strategy(pStrategy) {

    }
    Strategy* getStrategy() {
        return strategy;
    }
    void setStrategy(Strategy* pStrategy) {
        strategy=pStrategy;
    }
    void contextInterface() {
        strategy->algorithmInterface();
    }
};
```

Son olarak istemciyi kodlayalım;

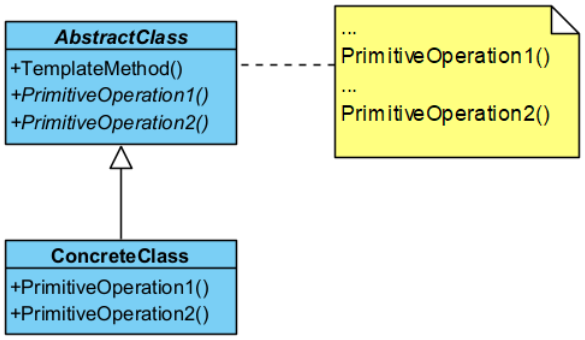
```
int main() { //istemci-client
    Strategy* strateji1=new ConcreteStrategyA();
    Context* baglam= new Context(strateji1);
    baglam->contextInterface();

    Strategy* strateji2=new ConcreteStrategyB();
    baglam->setStrategy(strateji2);
    baglam->contextInterface();

    Strategy* strateji3=new ConcreteStrategyC();
    baglam->setStrategy(strateji3);
    baglam->contextInterface();

    delete baglam,strateji1,strateji2,strateji3;
}
/*Program Çıktısı:
A stratejisinin Algoritması Çalıştırılıyor..
B stratejisinin Algoritması Çalıştırılıyor..
C stratejisinin Algoritması Çalıştırılıyor..

...Program finished with exit code 0
*/
```

Şablon Yöntem

Şablon yöntem deseninde (template method pattern), bir algoritmanın çerçevesini belirler ve uygulayıcı sınıfların gerçek davranışı tanımlamasına olanak tanır.

Kullanılacak algoritmalardan hangilerinin standart hangilerinin somut sınıflara özgü olacağı belirlenmelidir;

- İçinde "Bizi çağırmayın! Biz sizi çağıracağız." sloganını taşıyan yöntemler olan bir soyut bir taban sınıf tasarlanmalıdır.
- Taban sınıf içinde algoritmanın kabuğunu oluşturacak standart adımları içeren şablon yöntem tanımlanır.
- Somut sınıflarda detaylandırılacak yöntemlere ilişkin sanal yöntemler taban sınıfta tanımlanır.
- Şablon yöntem içinde algoritmayı oluşturan yöntemler çağrılır.
- Şablon yöntem içinde yer almayan tüm detay kodlamalar somut sınıflar için de yapılır.



Şekil 51. Şablon Yöntem Deseni UML Diyagramı

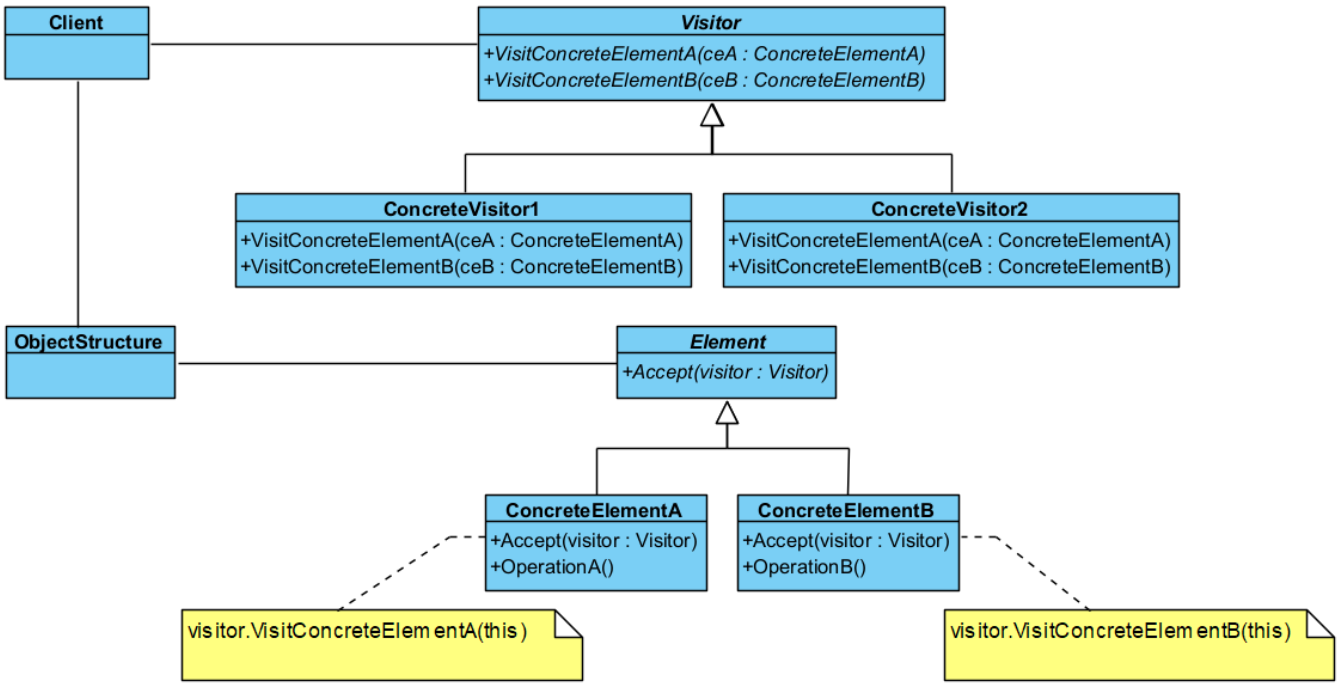
Şimdi Programı kodlayalım;

```
#include <iostream>
using namespace std;

class AbstractClass {
public:
    void templateMethod(){
        baseOperation1();
```

```
        primitiveOperation1();
        requiredOperations1();
        baseOperation2();
        primitiveOperation2();
        requiredOperations2();
    }
    virtual void primitiveOperation1()=0;
    virtual void primitiveOperation2()=0;
protected:
    void baseOperation1() const {
        std::cout << "Algotitmanın 1. TEMEL Adımı yapılıyor...\n";
    }
    void baseOperation2() const {
        std::cout << "Algotitmanın 2. TEMEL Adımı yapılıyor...\n";
    }
    virtual void requiredOperations1()=0;
    virtual void requiredOperations2()=0;
};
class ConcreteClass: public AbstractClass {
public:
    void primitiveOperation1(){
        cout << "Algoritmanın 1. adımı yapılıyor..." << endl;
    }
    void primitiveOperation2(){
        cout << "Algoritmanın 2. adımı yapılıyor..." << endl;
    }
protected:
    void requiredOperations1() override {
        cout << "Algoritmanın 1. GEREKLİ adımı yapılıyor..." << endl;
    }
    void requiredOperations2() override {
        cout << "Algoritmanın 2. GEREKLİ adımı yapılıyor..." << endl;
    }
};
int main() {
    AbstractClass* algoritma= new ConcreteClass();
    algoritma->templateMethod();

    delete algoritma;
}
/*Program Çıktısı:
Algotitmanın 1. TEMEL Adımı yapılıyor...
Algoritmanın 1. adımı yapılıyor...
Algoritmanın 1. GEREKLİ adımı yapılıyor...
Algotitmanın 2. TEMEL Adımı yapılıyor...
Algoritmanın 2. adımı yapılıyor...
Algoritmanın 2. GEREKLİ adımı yapılıyor...

...Program finished with exit code 0
*/
```

Ziyaretçi

Ziyaretçi deseni (visitor pattern), bir sınıfa ilişkin verileri kullanarak işlem yapan bir başka yeni sınıf tanımlanmak istendiğinde kullanılır. Yani çalışma zamanında bir veya daha fazla işlemin bir nesne kümesine uygulanmasına izin verir ve işlemleri nesne yapısından ayırır.



Şekil 52. Ziyaretçi Deseni UML Diyagramı

Başka amaçla tanımlanan yeni sınıfın (`Visitor`), mevcut sınıfın (`Element`) verilerine ulaşmak istemesi halinde bu desen kullanılır. Bu desen aşağıdaki üstünlükleri sağlar;

- Yeni işlemleri (operation) sisteme eklemeyi kolaylaştırır.
- Birbiriyle ilişkili olan işlemleri ilişkili olmayandan ayırmaya yardımcı olur.
- Yeni `ConcreteElement` nesnelerinin eklenmesini zorlaştırır.
- Bütün nesne hiyerarşisi baştanbaşa ziyaret edilebilir.
- Sınıflardaki tüm durumlar (state) biriktirilebilir.
- Bazı durumlarda sarma (encapsulation) işlemini engeller. Yani nesnenin iç durumlarından hareketle herkese açık işlemler tanımlanabilir.

Bu desen aşağıdaki durumlarda kullanılır;

- Nesneler birden fazla olup birçok ara yüze sahip olduğunda, bu nesnelerin somut sınıfları ile bir işlem yapmak gerektiğinde,
- Sınıflar üzerinde yapılacak işlemlerin, sınıflar üzerinde bir iz bırakmamasını sağlamak amacıyla,
- Nadir değişen sınıf yapılarına, sınıf yapısını değiştirmeden, yeni işlemler eklemek gerektiğinde.

İlk önce `Visitor` sınıflarını kodlayalım;

```
#include <iostream>
using namespace std;

class ConcreteElementA; // Böyle bir sınıf tanımlanacak!
class ConcreteElementB; // Böyle bir sınıf tanımlanacak!
class Visitor {
public:
    virtual void visitConcreteElementA(ConcreteElementA* pConcreteElementA)=0;
    virtual void visitConcreteElementB(ConcreteElementB* pConcreteElementB)=0;
};
class ConcreteVisitor1 : public Visitor{
public:
    void visitConcreteElementA (ConcreteElementA* pConcreteElementA) override {
        cout << "ConcreteElementA, ConcreteVisitor1 tarafından ziyaret edildi" << endl;
    }
    void visitConcreteElementB (ConcreteElementB* pConcreteElementB) override {
        cout << "ConcreteElementB, ConcreteVisitor1 tarafından ziyaret edildi" << endl;
    }
```

```
};
class ConcreteVisitor2 : public Visitor{
    void visitConcreteElementA (ConcreteElementA* pConcreteElementA) override {
        cout << "ConcreteElementA, ConcreteVisitor2 tarafından ziyaret edildi" << endl;
    }
    void visitConcreteElementB (ConcreteElementB* pConcreteElementB) override {
        cout << "ConcreteElementB, ConcreteVisitor2 tarafından ziyaret edildi" << endl;
    }
};
```

Daha sonra `Element` sınıflarını kodlayalım;

```
class Element{
public:
    virtual void accept(Visitor* visitor) =0;
};
class ConcreteElementA : public Element{
public:
    void accept(Visitor* visitor) override {
        visitor->visitConcreteElementA(this);
        operationA();
    }
    void operationA() {
        cout << "ConcreteElementA, operationA işlemini yürütüyor..." << endl;
    }
};
class ConcreteElementB : public Element{
public:
    void accept(Visitor* visitor) override {
        visitor->visitConcreteElementB(this);
        operationB();
    }
    void operationB() {
        cout << "ConcreteElementB, operationB işlemini yürütüyor..." << endl;
    }
};
```

Bir de `ObjectStructure` sınıfını kodlayalım;

```
class ObjectStructure{
private:
    list<Element*> elements;
public:
    void attach(Element* pElement){
        elements.push_back(pElement);
    }
    void detach(Element* pElement){
        elements.remove(pElement);
    }
    void accept(Visitor* pVisitor){
        for (Element* element : elements)
            element->accept(pVisitor);
    }
};
```

Son olarak istemci kodumuzu kodlayalım;

```
int main() { //İstemci-Client
    ObjectStructure* nesneler = new ObjectStructure();
    nesneler->attach(new ConcreteElementA());
    nesneler->attach(new ConcreteElementB());
```

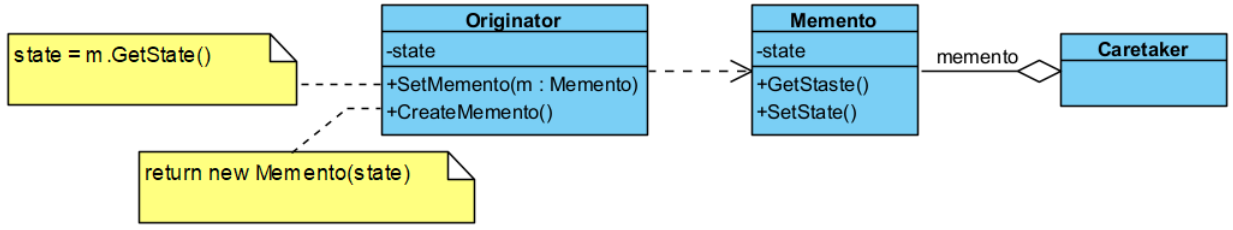
```
    Visitor* ziyaretci1 = new ConcreteVisitor1();
    Visitor* ziyaretci2 = new ConcreteVisitor2();
    nesneler->accept(ziyaretci1);
    nesneler->accept(ziyaretci2);
}
/* Programın Çıktısı:
ConcreteElementA, ConcreteVisitor1 tarafından ziyaret edildi
ConcreteElementA, operationA işlemini yürütüyor...
ConcreteElementB, ConcreteVisitor1 tarafından ziyaret edildi
ConcreteElementB, operationB işlemini yürütüyor...
ConcreteElementA, ConcreteVisitor2 tarafından ziyaret edildi
ConcreteElementA, operationA işlemini yürütüyor...
ConcreteElementB, ConcreteVisitor2 tarafından ziyaret edildi
ConcreteElementB, operationB işlemini yürütüyor...

...Program finished with exit code 0
*/
```

Ziyaretçi deseni uygulanırken iki kodlama (implementation) sorunu irdelenmelidir;

- Çifte sevkiyat (double dispatch), Bir sınıfı hiçbir şekilde değiştirmeden, söz konusu sınıfa etkin biçimde yöntem eklenebilmesi işlemidir. Bu çok bilinen bir tekniktir ve Common Lisp Object System (CLOS) gibi programlama dilleri tarafından desteklenir ve nesne üzerinde sarma (encapsulation) yapılmasına izin verilmez.
- C++, C#, Java gibi birçok nesne yönelimli programlama dili tek sevkiyat (single dispatch) yöntemi ile çalışır. Bu teknik ziyaretçi deseni için anahtardır. `Accept` yöntemi çifte sevkiyat sağlayan yöntemdir. Yöntemler, statik olarak yalnızca `Element` sınıfına bağlanmaz. `Element` sınıfı yapılacak işlemleri `Visitor` üzerinde birleştirir ve `Accept` yöntemi ile çalıştırma anında icra eder. İcra edilen yöntemler, `Visitor` ve `Element` veri tipi olmak üzere iki veri tipine bağlıdır.
- İşlemler dışarıda tutularak oluşturulan nesne yapısının sorumlusu kimdir? `Visitor`, oluşturulan nesne yapısının her bir elemanını tek tek ziyaret etmek zorundadır. Bunun nasıl ve hangi sırada yapılacağı belirgin değildir. Birçok durumda nesne yapısının kendisi bundan sorumlu tutulur. Nesne yapısı basit bir küme olarak tanımlanır ve `Accept` yöntemi her biri için çağrılır. Bazı durumlarda da yineleyici deseni (iterator pattern) kullanılır.

Hatıra

Hatıra deseni (memento pattern), nesnelerin bir andaki durumlarını (state) saklayıp bir başka zaman da bu duruma geri dönmek gerektiğinde kullanılır. Bir nesnenin iç durumunun yakalanmasına ve dışarda saklanmasına olanak tanır, böylece daha sonra geri yüklenebilir fakat tüm bunlar sarmalamayı (encapsulation) ihlal etmeden yapılır.



Şekil 53. Hatıra Deseni UML Diyagramı

Yukarıdaki diyagram incelendiğinde; `Originator` nesnesi, geçmişteki duruma dönecek nesneyi, `Memento` ise `Originator` nesnesinin bir anlık resmini çeken nesneyi, `Caretaker` ise `Memento` nesnelerini saklayan yardımcı nesneyi ifade etmektedir.

İlk olarak `Memento` sınıfını kodlayalım;

```
#include <iostream>
using namespace std;
class Memento {
```

```
private:
    int state;
public:
    Memento(int pState):state(pState) {
    }
    int getState() {
        return state;
    }
    void setState(int pState) {
        state=pState;
    }
};
```

Daha sonra `Caretaker` sınıfını kodlayalım;

```
class CareTaker{
private:
    Memento* memento;
public:
    CareTaker(Memento* pMemento):memento(pMemento) {
    }
    Memento* getMemento() {
        return memento;
    }
    void setMemento(Memento* pMemento) {
        memento=pMemento;
    }
};
```

Bir de `Originator` sınıfını kodlayalım;

```
class Originator {
private:
    int state;
public:
    void setState(int pState) {
        state=pState;
        cout << "State Değiştirildi:" << state << endl;
    }
    int getState() {
        return state;
    }
    void setMemento(Memento* pMemento) {
        state=pMemento->getState();
        cout << "State geri alındı:" << state << endl;
    }
    Memento* createMemento() {
        cout << "State saklandı:" << state << endl;
        return new Memento(state);
    }
};
```

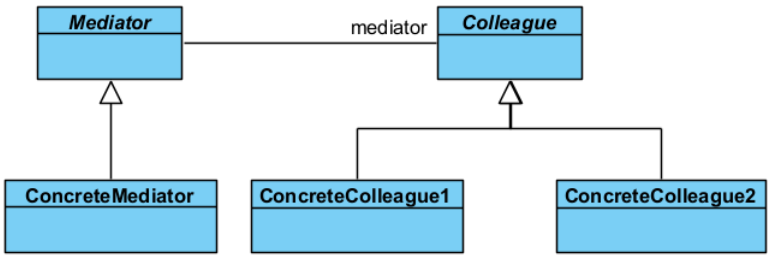
Son olarak istemci kodunu yazalım;

```
int main() { //istemci-client
    Originator* durumDegistiren= new Originator();
    durumDegistiren->setState(1);
    //Burada durumDegistiren nesnesinin durumu saklanıyor:
    CareTaker* durumuSaklayan = new CareTaker(durumDegistiren->createMemento());
    durumDegistiren->setState(-1);
    //Burada durumDegistiren nesnesinin durumu geri alınıyor:
    durumDegistiren->setState(durumuSaklayan->getMemento()->getState());
```

```
}
/*Program Çıktısı:
State Değiştirildi:1
State saklandı:1
State Değiştirildi:-1
State Değiştirildi:1

...Program finished with exit code 0
*/
```

Arabulucu

Arabulucu deseni (mediator pattern), sınıfların arasındaki iletişimi basitleştirir. Bir iletişim nesnesi tanımlanır, diğer nesneler bu nesne üzerinden birbirleriyle haberleşir. Diğer nesnelerin birbiriyle doğrudan iletişim kurmasına izin verilmez. Amaç, iletişimin kontrol altına alınmasıdır. Bu desende arabulucunun (mediator haberi olmadan hiçbir kişi (colleauge) birbiriyle haberleşmez. Ya da bir başka deyişle arabulucunun haberi olmadan hiç kimse birbiriyle görüşmez.



Şekil 54. Arabulucu Deseni UML Diyagramı

Bu desende, sistemde yalnızca bir adet arabulucu olacaksa soyut `Mediator` nesnesi tanımlanmaz. Ayrıca arabulucu ile kişiler arasındaki iletişim iki türlü yapılır;

- Birincisinde bu iletişim gözlemci deseni (observer pattern) ile sağlanır. `Colleague` sınıfı gözlemci desenindeki `Subject` sınıf gibi davranır.
- İkincisinde ise `Mediator` nesnesine, kişiler (colleauge) görüşecekleri kişileri belirterek doğrudan ileti gönderirler.

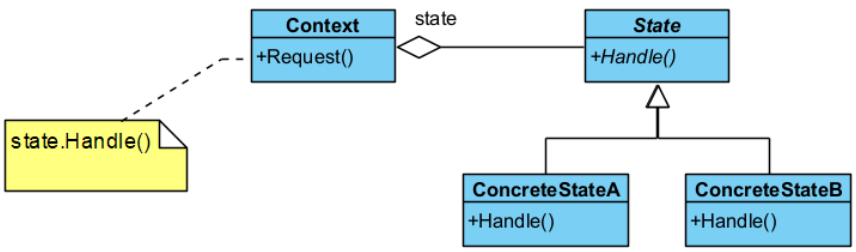
Aşağıda tek bir `Mediator` ve `ConcreteColleague` sınıf olan bir uygulama örneği verilmiştir;

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
class Colleague;
class Mediator {
protected:
    list<Colleague*> colleauges;
public:
    virtual void attach(Colleague* pColleague)=0;
    virtual void send(string pMessage, Colleague* pColleauge)=0;
};
class Colleague {
protected:
    string name;
    Mediator* mediator;
public:
    Colleague(string pName,Mediator* pMediator):name(pName),mediator(pMediator) {
        mediator->attach(this);
    }
    string getName() {
        return name;
    }
```

```
    virtual void sendMessage(string pMessage) {
        mediator->send(pMessage, this);
    }
    virtual void getMessage(string pMessage) {
        cout << name <<":mesaj aldım:" << pMessage << endl;
    }
};
class ConcreteMediator: public Mediator {
public:
    virtual void attach(Colleague* pColleague) {
        colleauges.push_back(pColleague);
    }
    void send(string pMessage, Colleague* pColleauge){
        cout << "LOG:" << pColleauge->getName() << " mesaj gönderdi:" << pMessage << endl;
        for (Colleague* college:colleauges)
            college->getMessage(pMessage);
    }
};
class ConcreteColleague:public Colleague {
public:
    ConcreteColleague(string pName,Mediator* pMediator):Colleague(pName,pMediator) {
    }
};
int main() { //istemci-client
    Mediator* arabulucu=new ConcreteMediator();
    Colleague* ilhan=new ConcreteColleague("ilhan",arabulucu);
    Colleague* mehmet=new ConcreteColleague("mehmet",arabulucu);

    ilhan->sendMessage("Selam");
    mehmet->sendMessage("Merhabalar...");
    ilhan->sendMessage("Nasılsınız?");
}
/* Program Çıktısı:
LOG:ilhan mesaj gönderdi:Selam
ilhan:mesaj aldım:Selam
mehmet:mesaj aldım:Selam
LOG:mehmet mesaj gönderdi:Merhabalar...
ilhan:mesaj aldım:Merhabalar...
mehmet:mesaj aldım:Merhabalar...
LOG:ilhan mesaj gönderdi:Nasılsınız?
ilhan:mesaj aldım:Nasılsınız?
mehmet:mesaj aldım:Nasılsınız?

...Program finished with exit code 0
*/
```

Durum

Durum deseni (state pattern), nesnenin durumunu davranışına bağlar, nesnenin içsel durumuna göre farklı şekillerde davranmasına olanak tanır.



Şekil 55. Durum Deseni UML Diyagramı

Bu desenin bağlam (`Context`) nesnesinin, diğer tasarım desenleri ile iç içe kullanılması halinde, sıfır desen (null pattern) olarak adlandırılır. Bu desenle ilgili olarak iki önemli şeyden bahsetmek gerekir:

- Sahip olunan duruma göre icra edilecek davranış, tanımlanan `State` sınıfına bağlıdır. Yani ilgili davranış somut sınıflardan biri tarafından icra edilir. Bu nedenle somut `State` sınıfları içinde tanımlanan yöntemler durumu kullanan bağlam (`Context`) sınıf içinde tanımlanmak zorundadır.
- Durumu kullanan bağlam (`Context`) sınıf içinde hangi duruma geçildiğinin anlaşılması için ilgili duruma ilişkin tanımlanan özellik (property) içinde set ifadesi kullanılmak zorundadır.

Bu desen netice olarak, nesnenin duruma özgü davranış göstermesini sağlar veya farklı durumlara göre davranışları birbirinden ayırır, durum geçişlerini (state transition) açığa çıkarır. Bu deseni kullanırken aşağıda verilen durumlar özellikle irdelenmelidir;

- Durum geçişleri kim tarafından tanımlanmalıdır? Bu desen, hangi şartlarda durumların ayrıştırılacağını belirtmez. Eğer şartlar belirgin ise bu işlem `Context` nesnesi tarafından yapılır. Bu şartlar belirgin değil ve karmaşık ise bu işlem `State` nesnesi ve halefi (successor) tarafından yapılır.
- Tablo tabanlı alternatif: Bu durumda durum geçişleri bir tablo aracılığı ile yapılır. Tablo üzerine sağlanacak her bir durum yazılır. Bu durumda, durum geçişlerini sağlayacak değişiklikler yeniden kodlama gerektirmez. Ancak bu yöntemde hız düşer. Durum geçişleri tekdüze (uniform) tanımlanmak zorundadır.
- `State` nesnelerini yaratma ve yok etme: Bu nesnenler çalıştırma anında yaratılıp yok edildiklerinden, bu süre zarfında yapılacak işlemler için bir çözüm bulunmalıdır. İki yaklaşım vardır. Birincisi, `State` nesnesini ihtiyaç olduğunda yaratmak kullanmak ve öldürmektir. İkincisi ise bu nesneleri baştan yaratıp hiç öldürmemektir. Genellikle birinci yöntem tercih edilir.
- Dinamik kalıtım (dynamic inheritance) kullanma: Bazı durumlarda nesne, ait olduğu sınıfı çalıştırma anında değiştirir. Bu durumda talep edilen isteğin yerine getirilmesi tam olarak başarılamaz. Ancak birçok nesne yönelimli programlama dili bunu desteklemez.

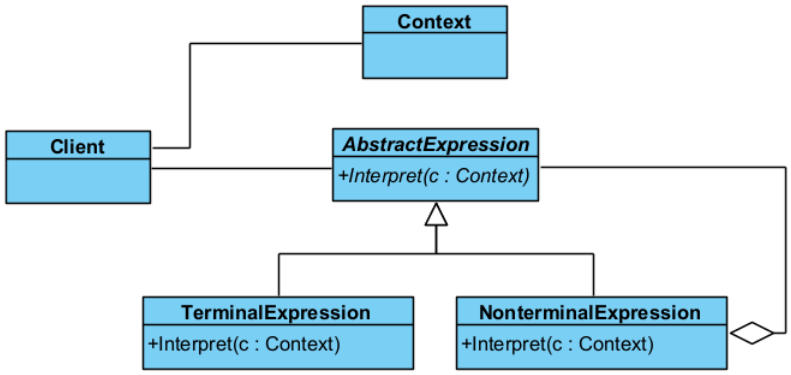
Aşağıda durum nesnelerini değiştiren basit bir önek verilmiştir;

```
#include <iostream>
using namespace std;

class State {
public:
    virtual void handle() = 0;
};
class Context {
private:
    State* state;
    State* state1;
    State* state2;
public:
    Context(State *pState1,State* pState2) : state1(pState1),state2(pState2) {
        state=pState1;
    }
    void durumDegistir() {
        if (state==state1)
            state=state2;
        else
            state=state1;
        cout << "Context: Durum nesnesi değişti!" << endl;
    }
    void request() {
        this->state->handle();
        durumDegistir();
    }
};
class ConcreteStateA : public State {
public:
```

```
    void handle() override {
        cout << "ConcreteStateA  request talebini yerine getiriyor..." << endl;
    };
};
class ConcreteStateB : public State {
public:
    void handle() override {
        cout << "ConcreteStateB  request talebini yerine getiriyor..." << endl;
    }
};
int main() {
    State* durumA=new ConcreteStateA();
    State* durumB=new ConcreteStateB();
    Context *context = new Context(durumA,durumB);
    context->request();
    context->request();
    context->request();
    delete durumA,durumB,context;
}
/* Program Çıktısı:
ConcreteStateA  request talebini yerine getiriyor...
Context: Durum nesnesi değişti!
ConcreteStateB  request talebini yerine getiriyor...
Context: Durum nesnesi değişti!
ConcreteStateA  request talebini yerine getiriyor...
Context: Durum nesnesi değişti!

...Program finished with exit code 0
*/
```

Yorumlayıcı

Yorumlayıcı deseni (interpreter pattern), programlama dili yorumlama özelliğini uygulamalarımıza katmanın yolunu gösterir. Yani bir dilbilgisi için bir temsili ve aynı zamanda dilbilgisini anlayıp ona göre hareket etmeyi sağlayacak bir mekanizmayı tanımlar.



Şekil 56. Yorumlayıcı deseni UML Diyagramı

Bu desen, yorumlanacak dilin dilbilgisi (grammar) kurallarının basit olduğu durumlarda kullanılır. Karmaşık dilbilgisi kurallarına sahip diller için yaratılacak sınıfların yönetimi oldukça zorlaşır. Burada, bağlam (`Context`) nesnesi dilbilgisi kurallarına sahip yorumlanacak nesneyi gösterir. `TerminalExpression`, en ilkel dil öğesini yorumlayacak nesnedir. `NonterminalExpression` ise ilkel dil öğelerinden oluşacak karmaşık yapıyı yorumlayacak nesneyi belirtir.

Bu deseni uygularken aşağıda verilen durumlar irdelenmelidir;

- Soyut sözdizimin oluşturulması: Bu desen yorumlanacak dile ilişkin soyut bir sözdizimin nasıl olması gerektiğini açıklamaz. Tabloya dayalı, ağaç yapısı şeklinde el becerisi ile veya doğrudan istemci tarafından yapılabilir.
- Yorumlayacak işlemin tanımlanması: Eğer işlemler ortak bir yapıdaysa yeni bir `Expression` nesnesi tanımlanarak yapılabilir. Daha karmaşık durumlarda ziyaretçi deseni kullanılır. Programlama dillerinin çoğu soyut bir sözdizimine sahiptir ve yorumlanacak öğeler ayrı bir `Visitor` nesnesi tarafından yorumlanır.
- Cümle sonlarını bitiren ortak semboller için sineksıklet deseni kullanılabilir: Cümle sonlarındaki semboller genellikle bilgi saklamazlar. Yorumlamaya gerek duyulmazlar. Bu nedenle sineksıklet deseni (flyweight pattern) yorumlanacak yerler için bir eleme yapar.

Aşağıda kavramsal olarak bu desenin kolanmış örneği bulunmaktadır;

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

class Context {
private:
    string context;
public:
    Context(string pContext): context(pContext) {
    }
};

class AbstractExpression {
public:
    virtual void interpret(Context* context)=0;
};

class TerminalExpression : public AbstractExpression {
public:
    void interpret(Context* context) override {
        cout << "Terminal Expression İşlemi Yorumluyor..." << endl;
    }
};

class NonterminalExpression : public AbstractExpression {
private:
    AbstractExpression* abstractexpression;
public:
    void interpret(Context* context) override {
        cout << "Non Terminal Expression İşlemi Yorumluyor..." << endl;
        abstractexpression = new TerminalExpression();
        abstractexpression->interpret(context);
        delete abstractexpression;
    }
};

int main(){ //İstemci-Client
    list<AbstractExpression*> expressions;
    AbstractExpression* nte=new NonterminalExpression();
    AbstractExpression* te=new TerminalExpression();
    expressions.push_back(te);
    expressions.push_back(nte);
    Context* context = new Context("987698");
    for (AbstractExpression* ae:expressions)
        ae->interpret(context);
}
```

```
/*Program Çıktısı:
Terminal Expression İşlemi Yorumluyor...
Non Terminal Expression İşlemi Yorumluyor...
Terminal Expression İşlemi Yorumluyor...

...Program finished with exit code 0
*/
```

ÇOK DEĞİŞKENLİ FONKSİYONLAR

Değişken Sayıda Argüman Alabilen Fonksiyonlar

Değişken sayıda argüman alabilen bir fonksiyona çok değişkenli fonksiyon (variadic function) denir. C dilindeki güçlü ancak çok nadiren kullanılan özelliklerden biridir. C++ dili de bu fonksiyonları özellik olarak devralmıştır.

Değişken sayıda bağımsız değişkeni olan bir fonksiyon; en az bir sabit bağımsız değişkene sahip olacak şekilde tanımlanır ve ardından derleyicinin değişken sayıda bağımsız değişkeni ayrıştırmasını sağlayan bir üç nokta simgesi (elipsis) eklenir.

```
dönüş-tipi fonksiyonkimliği(veri-tipi birinciargüman, ...);
```

Değişken argümanları işlemek için kodunuza `stdarg.h` başlık dosyasını kodumuza dahil etmeniz gerekir;

Fonksiyon	Açıklama
`va_start(va_list ap, arg)`	Bu fonksiyon, üç nokta ile verilen argümanları `va_list` değişkenine aktarır.
`va_arg(va_list ap, type)`	Her seferinde, üç nokta ile temsil edilen değişken listesindeki bir sonraki argümanı `va_list` üzerinden işler ve listenin sonuna ulaşana kadar onu `type` ile verilen veri tipine dönüştürür.
`va_copy(va_list dest, va_list src)`	`va_list`'teki argümanların bir kopyasını oluşturur.
`va_end(va_list ap)`	Bu, `va_list` değişkenlerine erişimi sonlandırır.

Tablo 25. CStdarg Fonksiyonları

Aşağıda ilk parametre ile belirlenmiş argüman sayısı kadar argüman alan iki fonksiyon tanımlanmıştır.

```
#include <iostream>
#include <cstdarg>
using namespace std;

void sayilariYaz(int count, ...) {
    va_list args;
    va_start(args, count);

    for (int i = 0; i < count; ++i) {
        cout << va_arg(args, int) << " ";
    }
    cout << endl;
    va_end(args);
}
int argumanlarinHepsiniTopla(int kacAdet, ...) {
   va_list argumanlar;
   int sayac, toplam = 0;
   va_start(argumanlar, kacAdet);
   for (sayac = 0; sayac < kacAdet; sayac++)
      toplam += va_arg(argumanlar, int);
   va_end(argumanlar);
   return toplam;
}
int main(){
    sayilariYaz(4, 1.0, 2.0, 30, 40);
    cout << "3 Argüman Toplamı =" << argumanlarinHepsiniTopla(3, 10, 20, 30)
         << endl;
    cout << "5 Argüman Toplamı =" <<
            argumanlarinHepsiniTopla(5, 10, 20, 30, 40, 50) << endl;
}
```

```
/*Program Çıktısı:
30 40 2140265872 0
3 Argüman Toplamı =60
5 Argüman Toplamı =150

...Program finished with exit code 0
*/
```

Ana Fonksiyonun Parametreleri

C++ dilinde ana fonksiyon (main function) programın icra edilmeye başladığı fonksiyondur. Ana fonksiyon;

- Aşırı yüklenemez (overload)!
- Satır içi (inline) fonksiyon olarak bildirilemez!
- Statik (static) fonksiyon olarak bildirilemez!
- Adresi alınamaz!
- Programın başka hiçbir yerinden çağrılamaz (call)!

Yazdığımız programlar da çalıştırılırken konsoldan argüman alabilir. Şu ana kadar argümansız ana fonksiyonu gördük. Argüman alan ana fonksiyon (main function) aşağıdaki iki şekilde gibi tanımlanır;

```
int main(int argc,
         char* argv[]) {
    // Ana fonksiyon Gövdesi
}
//YADA
int main(int argc,
         char* argv[],
         char ** envp) {
    // Ana fonksiyon Gövdesi
}
```

Aşağıda konsoldan çalıştırılırken alınan argümanları gösteren bir program örneği verilmiştir;

```
#include <iostream>
using namespace std;
int main( int argc, char *argv[] )
{
    for ( int i = 0; argv[i] != NULL; ++i )
        cout << i << ": "<< argv[i] << "\n";
}
/* "C:\Users\ILHANOZKAN>main.exe 10 20 -v" Şeklinde :alıştırıldığıda:
0: main.exe
1: 10
2: 20
3: -v
*/
```

Aşağıda konsoldan çalıştırılırken girilen parametreler ile işletim siteminin ortam değişkenleri (environment variable) konsola yazan bir program örneği verilmiştir;

```
#include <iostream>
using namespace std;
int main( int argc, char *argv[], char *envp[] )
{
    for ( int i = 0; argv[i] != NULL; ++i )
        cout << i << ": "<< argv[i] << "\n";

    for ( int i = 0; envp[i] != NULL; ++i )
        cout << i << ": "<< envp[i] << "\n";
}
```

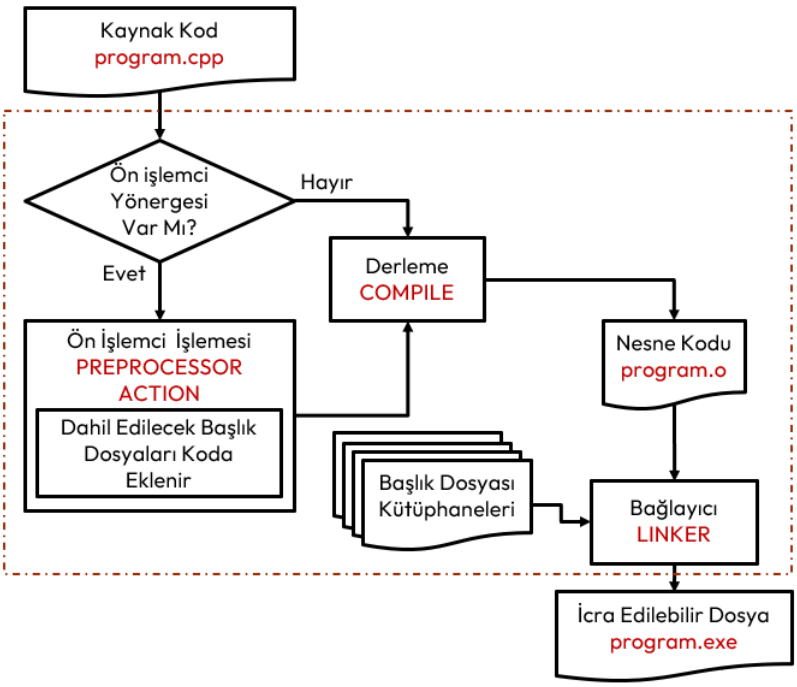
```
/* “C:\Users\ILHANOZKAN>main.exe 10 20 -v” Şeklinde çalıştırıldığıda:
0: main.exe
1: 10
2: 20
3: -v
0: ALLUSERSPROFILE=C:\ProgramData
1: APPDATA=C:\Users\ILHANOZKAN\AppData\Roaming
2: CommonProgramFiles=C:\Program Files\Common Files
3: CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
4: CommonProgramW6432=C:\Program Files\Common Files
5: COMPUTERNAME=ILHANOZKAN
6: ComSpec=C:\Windows\system32\cmd.exe
7: DriverData=C:\Windows\System32\Drivers\DriverData
8: EFC_25916=0
9: FPS_BROWSER_APP_PROFILE_STRING=Internet Explorer
10: FPS_BROWSER_USER_PROFILE_STRING=Default
11: HOMEDRIVE=C:
12: HOMEPATH=\Users\ILHANOZKAN
13: LOCALAPPDATA=C:\Users\ILHANOZKAN\AppData\Local
14: LOGONSERVER=\\ILHANOZKAN
15: NUMBER_OF_PROCESSORS=20
16: OneDrive=C:\Users\ILHANOZKAN\OneDrive
17: OS=Windows_NT
18: Path=C:\Windows\system32;C:\Windows;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Program
Files\dotnet\;C:\Users\ILHANOZKAN\AppData\Local\Microsoft\WindowsApps;D:\msys64\ucrt64\bin;C:\
Users\ILHANOZKAN\.dotnet\tools
19: PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
20: PROCESSOR_ARCHITECTURE=AMD64
21: PROCESSOR_IDENTIFIER=Intel64 Family 6 Model 186 Stepping 2, GenuineIntel
22: PROCESSOR_LEVEL=6
23: PROCESSOR_REVISION=ba02
24: ProgramData=C:\ProgramData
25: ProgramFiles=C:\Program Files
26: ProgramFiles(x86)=C:\Program Files (x86)
27: ProgramW6432=C:\Program Files
28: PROMPT=$P$G
29: PSModulePath=C:\Program
Files\WindowsPowerShell\Modules;C:\Windows\system32\WindowsPowerShell\v1.0\Modules
30: PUBLIC=C:\Users\Public
31: SESSIONNAME=Console
32: SystemDrive=C:
33: SystemRoot=C:\Windows
34: TEMP=C:\Users\ILHANO~1\AppData\Local\Temp
35: TMP=C:\Users\ILHANO~1\AppData\Local\Temp
36: UATDATA=C:\Windows\CCM\UATData\D9F8C395-CAB8-491d-B8AC-179A1FE1BE77
37: USERDOMAIN=ILHANOZKAN
38: USERDOMAIN_ROAMINGPROFILE=ILHANOZKAN
39: USERNAME=ILHANOZKAN
40: USERPROFILE=C:\Users\ILHANOZKAN
41: windir=C:\Windows
42: ZES_ENABLE_SYSMAN=1
*/
```

ÖN İŞLEMCİ YÖNERGELERİ

Ön İşlemci Yönergesi Nasıl Çalışır

C ve C++ dilinde ön işlemci (preprocessor), derleyicinin bir parçası değildir, ancak derleme sürecinde ayrı bir adımdır. Ön işlemci, yalnızca bir metin değiştirme aracıdır ve derleyiciye gerçek derlemeden önce gerekli ön işlemeyi yapmasını söyler. Yani derleme önce bul-değiştir mantığı ile kodda değişiklikler yaptırır.

- Ön işleme bir C++ kodunun derlenmesindeki ilk adımdır.
- Kodu sembollere ayırma (tokenization) adımından önce gerçekleşir.
- Ön işlemcinin önemli işlevlerinden biri de programda kullanılan kütüphane işlevlerini içeren başlık dosyalarını koda dahil etmek için kullanılmasıdır.
- Ön işlemci ayrıca sabitleri tanımlar ve makro kullanımını sağlar.



Şekil 57. Derleme Süreci

C++ dilindeki ön işlemci ifadelerine yönergeler (directive) denir. Programda ön işlemci bölümü her zaman C kodunun en üstünde görünür. Her ön işlemci ifadesi, karma (hash) sembolüyle başlar. Aşağıda en çok kullanılan yönergeler bulunmaktadır.

Yönerge	Açıklama
`#define`	Ön işlemci makrosunu değiştirmek ya da ya da ilk defa tanımlamak için kullanılır.
`#include`	Bir başlık dosyasını diğer ile dahil etmek için kullanılır.
`#undef`	Tanımlanmış bir makroyu tanımsız hale getirmek için kullanılır.
`#ifdef`	Bir makro tanımlı ise doğru/true değerini döndürür.
`#ifndef`	Bir makro tanımlı değilse doğru/true değerini döndürür.
`#if`	Derleme zamanında bir durumun kontrolü için kullanılır.
`#else`	`#if` Yönergesinde alternatif durumu ifade eder.
`#elif`	`#else` ve `#if` yönergelerini tek bir şekilde ifade etmek için kullanılır.
`#endif`	Şart ön işlemcilerini (`#if`, `#else`, `#elif`) bitirir.
`#error`	`stderr` standart dosyasına hata mesajını yazar.
`#pragma`	Derleyiciye özel komutlar vermek için kullanılır.

Tablo 26. Ön İşlemci Yönergeleri

Kullanıcı Tanımlı Başlık Dosyası

`#include` yönergesi ile; hazır olan başlık dosyalarını `< >` karakterleri arasında yazarak koda dahil ederiz. Hazır olmayan ve programcı tarafından hazırlanan başlık dosyalarını çift tırnak `" "` karakterleri arasında yazarak koda dahil ederiz.

`#define` yönergesi ile yeni bir makro tanımlanabileceği gibi tanımlanmış bir makro `#undef` ile ortadan kaldırabilir veya yenisini tanımlayabiliriz.

Aşağıdaki örnekte `PI` adlı bir makro tanımlanmış ve `3.1415` reel sayısını vermektedir. Daha önce `PI` adlı bir makro tanımlanmış ise derleyici hata verir.

```
#include <iostream>
#include "baslik.h"

#define PI 3.1415

#ifndef ENFAZLAOGRECISAYISI
   #define ENFAZLAOGRECISAYISI 100
#endif

#ifdef ENFAZLAOGRECISAYISI
   #undef ENFAZLAOGRECISAYISI
   #define ENFAZLAOGRECISAYISI 20
#endif
```

`DEBUG` hazır tanımlanmış bir makrodur ve hata ayıklama modunda kod derlenmesi halinde doğru/true değerini döndürür. Bunun için derleyiciye `-DDEBUG` argümanı verilir. `DEBUG` hazır tanımlanmış bir makro olup hata ayıklama modunda doğru/true olduğundan hata ayıklama modunda daha çok durum ve değer konsola yazılır. Bu durumda hatayı bulmak daha da kolaylaşır.

```
#ifdef DEBUG
  /*
   Hata ayıklama modunda
   derleme yapıldığında
   çalışacak kod buraya yazılır. */
#endif
```

Aşağıda kullanıcı tanımlı bir başlık dosyası örneği verilmiştir; Baslık dosyasının bir koda birden fazla dahil (include) edilmesini önlemek için `_BASLIK_H_` sabiti şartlı (conditional) olarak tanımlanmıştır. Bu başlık dosyası bir kod projesinde birden fazla dahil olması halinde, `_BASLIK_H_` tanımlanmaz ise, başlık içindeki değişken ve fonksiyonlar birden fazla aynı kimlikle tanımlanacağından derleme yapılamayacaktı.

```
#ifndef _BASLIK_H_
#define _BASLIK_H_

#include <iostream>
using namespace std;

#define PI 3.1415
#define ENFAZLAOGRECISAYISI 100

float* ogrenciNotlari() {
  static float notlar[ENFAZLAOGRECISAYISI];
  return notlar;
}

float ogrenciNotlarOrtalamasi(float* pNotlar, int pOgranciSayisi) {
  int sayac;
  float ortalama=0;
  for (sayac=0;sayac<pOgranciSayisi;sayac++) {
```

```
    ortalama+=pNotlar[sayac];
    #ifdef DEBUG
        cout << sayac << ". Ogrencide Ortalama: " << ortalama endl;
        // Buradaki kod hata ayıklama modunda derlendiğinde çalışır.
        // Bunun için derleyiciye -DDEBUG argümanı verilir
    #endif
  }
  return ortalama/sayac;
}
#endif
```

Bütün bu `#ifndef`, `#define` ve `#endif` yönergeleri ile yapılmak isteneni `#pragma once` yönergesi ile aşağıdaki gibi yapabiliriz.

```
#pragma once

#include <iostream>
using namespace std;

#define PI 3.1415
#define ENFAZLAOGRECISAYISI 100

float* ogrenciNotlari() {
  static float notlar[ENFAZLAOGRECISAYISI];
  return notlar;
}

float ogrenciNotlarOrtalamasi(float* pNotlar, int pOgranciSayisi) {
  int sayac;
  float ortalama=0;
  for (sayac=0;sayac<pOgranciSayisi;sayac++) {
    ortalama+=pNotlar[sayac];
    #ifdef DEBUG
        cout << sayac << ". Ogrencide Ortalama: " << ortalama endl;
        // Buradaki kod hata ayıklama modunda derlendiğinde çalışır.
        // Bunun için derleyiciye -DDEBUG argümanı verilir
    #endif
  }
  return ortalama/sayac;
}
```

Kendi hazırlamış olduğumuz başlık dosyasını (`baslik.h`) kodumuza dahil ettiğimizde artık başlık içindeki fonksiyon ve değişkenleri kullanabilir hale geliriz. Aynı klasörde/dizinde bulunacak şekilde bu başlık dosyasını kullanan örnek program aşağıda verilmiştir.

```
#include <iostream>
#include "baslik.h"

using namespace std;

int main() {
    printf("PI=%f\n",PI);
    printf("En fazla öğrenci Sayısı=%d\n",ENFAZLAOGRECISAYISI);
    float* notlar=ogrenciNotlari();
    notlar[0]=100;
    notlar[1]=80;
    notlar[2]=60;
    float ortalama=ogrenciNotlarOrtalamasi(notlar,3);
    cout << "3. öğrenci için Ortalama: " << ortalama << endl;
}
```

Ön Tanımlı Ön İşlemci Makroları

DEBUG gibi her c derleyicisi için standart olarak tanımlı makrolar bulunmaktadır;

Makro	*Açıklama*
`__DATE__`	Mevcut tarihi "MMM DD YYYY" biçiminde dizgi (string) olarak tanımlıdır.
`__TIME__`	Mevcut saat "HH:MM:SS" biçiminde dizgi (string) olarak tanımlıdır.
`__FILE__`	Dosya adı biçiminde dizgi (string) olarak tanımlıdır.
`__LINE__`	Dosyadaki satır numarası tamsayı (int) olarak tanımlıdır.
`__STDC__`	ANSI standardında derleme yapılıyorsa 1 olarak tanımlıdır.

Tablo 27. Standart Tanımlı Ön İşlemci Makroları

Bu makroları kullanan örnek program aşağıda verilmiştir;

```
#include <iostream>
using namespace std;

int main() {
    cout << "File: " << __FILE__ << endl;
    cout << "Date: " << __DATE__ << endl;
    cout << "Time: " << __TIME__ << endl;
    cout << "Line: " << __LINE__ << endl;
    cout << "ANSI: " << __STDC__ << endl;
}
```

Parametreli Ön İşlemci Makroları

Ön işlemci yönergeleriyle (preprocessor directive) tanımlanan makrolar daha derleme yapılmadan işlem görür. Dolayısıyla bu aşamada bazen parametrelerle işlem yapılması gerekebilir.

```
#define kare(x) ((x) * (x))
#define kup(x) ((x) * (x) *(x))
#define buyugu(x,y) ((x) > (y) ? (x) : (y))
```

Burada kullanılacak parametreler veri tipi tanımlanmaz. Dolayısıyla kullanılacağı yerlere buna dikkat edilerek parametreli makro tanımlanmalıdır. Eğer makro birkaç satırdan oluşacak ise ters bölü (back slash) `\` karakteri kullanılır.

```
#include <iostream>
using namespace std;

#define MAKRO(num, str) { \
            Cout << num  \
                 <<" is") \
                 << str << " number" \
                 << endl; \
           }
```

Derleme Sürecinin Detayı dilinde derleme (compile) sürecinin iki aşamadan oluştuğu daha önce anlatılmıştı. Aşağıda tüm süreç verilmiştir;

1. Birinci aşama:
   a. Kaynak koddaki tüm açıklamalar silinir.
   b. #define, #if, #include, gibi tüm yönergeler ile verilen işlemler yapılır. Bunlar arasında başlık dosyalarının koda dahil edilmesi de vardır.
   c. Bu aşamaya gelen kod, `g++ -E main.cpp` komutu yazılarak görülebilir.
2. İkinci aşama:
   a. Kaynak kod montaj koda (assembly code) çevrilir.
   b. Bu duruma gelen kod, `g++ -S main.cpp` komutu dosya haline getirilebilir.
   c. Assembly kod derlenerek makine diline çevrilir ve amaç dosya (object file) oluşur.
   d. Bu duruma gelen kod, `g++ -c main.cpp` komutu ile dosya haline getirilebilir.

e. Amaç dosya kodda belirtilen başlık dosyalarına uygun kütüphaneler (library) ile birbirine bağlanır ve icra edilebilir dosya (executable file) elde edilir. Burada bahsedilen kütüphaneler her işletim sistemi için ayrı olarak oluşturulur ve derleyici kurulumunda bir klasörde tutulur.
f. Şimdiye kadar yapılan tüm derlemelerde yukarıdaki adımlar atlanarak `g++ main.cpp -o main.exe` komutuyla icra edilebilir dosya oluşturulmaktadır.

ŞEKİL LİSTESİ

TABLO LİSTESİ

DİZİN