

Temel C Programlama

Gökhan Dökmetaş

www.lojikprob.com

©Bu elektronik kitabın tüm hakları Gökhan DÖKMETAS' a aittir. Kitap Ücretsizdir. PARA İLE SATILAMAZ. Ashna sadık kalınmak şartıyla ücretsiz olarak indirilebilir, çoğaltılabilir ve paylaşılabılır. Yasal iktibaslarda kaynak göstermek zorunludur.

Benimle iletişime geçmek için aşağıdaki e-posta adresini kullanabilirsiniz,
gokhandokmetas0@gmail.com

Ayrıca Facebook grubumuza üye olarak bütün çalışmalarımızdan haberdar olabilirsiniz.

<https://www.facebook.com/groups/lojikprob/>

Söz Başı

Günümüzde C programlama dili hakkında pek çok yazılı ve görsel içerik hazırlanmasına karşın öğrencilerin bundan tatmin olmadığını ve kaynak arayışında olduğunu görmekteydim. Öğrenciler özellikle kendilerine C programlama dilinin amacını ve işleyişini kavratacak bir kitap arıyordu. Pek çok içerik hazırlayıcısı gibi popüler konular hakkında pastadan pay kapma kabilinden içerik hazırlamayan biri olarak bu yönde büyük ihtiyaca istinaden bu dersleri yazma kararı aldım. Bir kitap yazar gibi yazdığım için toplanıp kitap haline getirilmesinde bir beis görmedim. Bu kitabın içeriği www.lojikprob.com sitesinde yer alan “Temel C Programlama” derslerinin derlenmesinden meydana gelmiştir.

Gökhan DÖKMETAS

-1- Giriş

C programlama dili derslerine kış tatilinde başlamayı düşünsem de bu arada dijital elektronik, bilgisayar mimarisi ve STM32 çalışmaları yaptığımız için C programlama derslerimiz gecikti. Bir sebebi bu olsa da asıl sebebi bir programlama dilini anlatabilmek için o dili gerçek anlamıyla anlamak gerektiğidir. O yüzden belki yıllar sonra anlatacak olsam da gelen talep doğrultusunda C ve Gömülü C programlama derslerini biraz aceleye getirmek zorundayım. Çünkü Elektrik-Elektronik ve Bilgisayar mühendislikleri dahil çoğu bölümü okuyan öğrencilerde C programlama dilini anlama ve kullanma sıkıntısı gözlemlemekteyim. Bu yönde bir eğitim verilse de kaynaklar da olsa da ya bu kaynaklar gömülü sistemlerde çalışmak isteyenlere yönelik değil ya da eğitim ezberci olduğundan öğrenci bu dili anlayıp kullanamıyor. İşin ilginç yanı olarak ise bunca yıl bilişim sektörünü takip eden biri olarak şunu gözlemleme şansını buldum. Bizim ülkede ezberci eğitimle Visual Basic, Visual C# gibi hazır kütüphane ve uygulama geliştirme arayüzüne sahip platformlar öğretiliyor ve öğrencinin kullanma şansı oluyor. İş ne zaman kalabalıktan kurtulup C, C++ gibi temel ve sade dillere gelirse o zaman öğrenci bunu öğrenemiyor. Bu aynı kelimelerden cümle kurabilme ya da hazır cümleleri ezberleyip kullanma gibi bir durum ortaya çıkarıyor. Ezbere eğitimle hazır kodları ve kütüphaneleri kullanabilse de kendisi temelden bir şey ortaya koyamıyor. C dili ise günümüzün en temel dili olduğu için ezberlemekten çok anlamayı gerektiriyor. O yüzden bunca zaman en zor kısım olarak C dilini anlatabilmeyi gördüm. Yazdığım Arduino Eğitim Kitabı'nda C programlama ve kütüphane kullanımına dair bilgilere yer versem de daha ileri bir seviye

kaynağa işte ilerledikçe ihtiyaç duyuluyor. Bu eksikliği ortadan kaldırma adına C programlama derslerini en kapsamlı şekilde yazmaya gayret edeceğiz.

Ders içeriği sadece gömülü C veya sırf C programlamaya yönelik olmayacak. Bu sefer farklı bir yaklaşım yapıp hem masaüstünde C programlama dilini hem de gömülü C programlamayı beraber anlatacağız. Kısacası bize lazım olan her konuyu anlatma gayretinde olacağız. Sadece gömülü sistemler üzerinde programlama yapmak günümüzde yeterli olmamaktadır. Pek çok elektronik projenin bilgisayar ile iletişime geçmesi gerekli ve bilgisayarda bir arayüz programının ihtiyacı duyulmaktadır. Bu yüzden C programlamayı öğrenmek hem gömülü sistemlerde program yazabilmeyi hem de masaüstü programlarını yazmayı sağlamalıdır. Öğrendiğimiz C dilinden sonra C++ diline geçiş yapabilmemiz gereklidir. Böylelikle Visual C++ gibi yazılım ortamlarında uygulama yazabilmeliyiz. Aynı zamanda C dilini alt seviye kullanabilmeli ve mikrodenetleyici ve mikroişlemciler üzerinde yazmaç tabanlı uygulama yazabilmeliyiz. Bu geniş bir yelpaze olsa da ülkemizde gömülü sistemler sektörü oldukça dar olduğu için çok yönlülüğün fazla olması gereklidir. Donanımcı-Yazılımcı diye bir ayırımdan kurtulup bir projenin her aşamasında söz sahibi olmamız bizim yararımıza olacaktır. Çünkü bazı projeler donanım yönünden iyi olsa da iyi yazılımcı olmadığı için ortaya çıkamayabilir. Bazı projelerin de donanım yönünden eksikliği yazılımı ne kadar iyi olursa olsun telafi edilemez.

Planladığımız ders içeriğini üç ana maddeye ayırmamız mümkündür,

- Temel C
- Gömülü C
- Masaüstü C/C++

Bunlardan Temel C kısmı en kolay kısım olup giriş konularından meydana gelecektir. Pek çok uygulamayı Eclipse üzerinden konsol ekranından gerçekleştireceğiz. Burada konsol ekranında uygulama yapmamız şart gibidir. Çünkü C dilini bilmeyen birine donanımı veya diğer yazılım ortamlarıyla beraber C dilini anlatmak iyice kafa karıştıracaktır. C dilini sadece C dili olarak kullanabildiğimiz konsol ortamları bu konuda kafa karışıklığına sebep olmayacaktır. O yüzden Gömülü C'ye geçmeden önce temel C programlamayı bu şekilde öğrenmek gereklidir. Bu temel C kısmında okuyucuyu gömülü C programlamaya hazırlar nitelikte bilgileri vermekten kaçınmayacağız.

Gömülü C kısmında ise donanım ile beraber dersleri yürüteceğiz. Çünkü donanım bilmeden gömülü sistemler üzerinde de gömülü C üzerinde de çalışmak mümkün değildir. Gömülü C'nin farkı zaten buradadır. O yüzden donanım ile yazılımı beraber götüreceğiz. Yalnız burada dersler AVR Programlamada olduğu gibi donanım ağırlıklı değil yazılım ağırlıklı geçecek. Önceden alışkın olduğumuz AVR ve STM32 üzerinden dersleri anlatmaya devam edeceğiz. Aynı zamanda ileri seviye Arduino konularına da değineceğiz. Örneğin kütüphaneleri inceleyeceğiz, kaynak kodundan örnekler vereceğiz, kütüphane yazacağız.

Masaüstü C/C++ kısmında ise artık C konularını bitirmiş olacağımızdan yavaştan C++ ile karışık C programları yazmaya başlayacağız. Visual C++ başta olmak üzere C ve C++ için masaüstü geliştirme ortamlarına göz atacağız. Klasik kitapların konsol ekranından çıkamaması gibi bir duruma düşmeyeceğiz. Hem teoriyi öğreneceğiz hem pratikte bunu gerçekleştireceğiz. Fakat uygulama yazma noktasında pratik daha ön planda olacak. Bilgisayar programı yazarak mikrodenetleyici kartları ile iletişimde bulunacağız ve arayüz

ortamı geliştireceğiz. Bu noktadan sonra tam kapsamlı bir proje geliştirmek için eksiklerimiz kalmayacak.

Lojikprob.com'a ilk yazmaya başladığım hakkında yıllardır Türkçe kaynak bulamadığım "AVR Programlama" konusundan itibaren pek çok başarıya imza attık ve artık internet sayfamız gömülü sistemler alanında bir numaralı kaynaklardan biri haline geldi. Bu projeyi tamamlamak amacıyla artık yazdığımıza "boş zaman uğraşı" gözüyle bakmıyor daha ciddiye alarak yazılarımızı yazmaya devam ediyoruz.

Daha önceki yazıda bahsettiğimiz üzere C programlama eğitimini üç ana başlıkta sizlere anlatacağız. Bu eğitimlerden ilki giriş seviyesi olup hiç bilmeyen birinin C dilini anlaması ve rahatça kullanmasına yönelik temel C dersleri olacak. Bu temel C derslerinin sadece fonksiyonları ve döngüler gibi basit konulardan ibaret olacağını düşünmemek gerekir. Piyasadaki C kitapları ve C eğitimi her zaman dilin temelini ve dili anlatmaktadır. Biz de dili anlattığımız ve diğer kitaplarda olduğu gibi öğretici konsol uygulamaları yaptığımız eğitime "Temel C" adını veriyoruz. Kısacası bu seriyi C dilini öğreneceğiniz bir kitap yerine koyabilirsiniz. Bu seride ileri seviye kavramlardan çok fazla söz ederek başlangıçta sizi sıkmak istemiyoruz. İleri seviye kavramları öğrenmeniz gerekse de bunu daha ileri konularda anlatacağız. Bu seri "Bilgisayar okur-yazarlığı" edinmiş herkese yönelik olacaktır. Bilgisayar okur yazarlığını programlamaya ve mühendislik uygulamalarına başlamadan önce edinmeniz gereklidir. Çoğu zaman mühendislik öğrencilerinin bilgisayar okur yazarlığı edinmeden 32-bit mikrodenetleyicileri bile programlamaya başladığını görebiliriz. Fakat bilgisayar okur yazarlığı edinmeyen birisi programlama aşamasında oldukça zorlanacaktır. O yüzden bilgisayar okur yazarlığı bu alanda çalışmak isteyenlerin ilk edinmesi gereken deneyimdir. Biz de bilgisayar okur

yazarlarına göre bu yazıları yazacağız. Yani basacağınız her düğmeyi, kuracağınız her programı “next tuşuna basın, ok deyin” diye anlatarak kaliteyi düşürmeyeceğiz.

Müfredat konusunda genellikle İngilizce eğitim kitaplarını ve referansları kaynak olarak alsak da bazen konunun dışına çıkarak ilginç noktaları sizinle paylaşacağız. Herhangi bir Türkçe kaynaktan istifade etmeyeceğiz ve ayrıca internetteki yabancı tutorial içerikleri gibi değil kitap kalitesinde bir seri ortaya koyacağız. Yazacağımız programlar konsol tabanlı olacak ve bilgisayar üzerinde çalışacak. Bu noktada öğrenciyi uğraştırmama ve kafasını karıştırmama adına doğrudan dile odaklanma gereği duyduk ve bunu da böyle sağlayacağız. Pek çok öğrencinin temel C bilgilerinden yoksun olduğunu görüyoruz bunları gömülü C ile başlatmakla zorluk çektirmek istemiyoruz.

Temel C derslerimizin prensibini ve sebebini anlattığımıza göre şimdi müfredattan kısaca bahsedelim. Bu müfredat dersler devam ederken zamanla ilave ve çıkarmalara maruz kalacak olsa da fikir edinmeniz açısından bir liste oluşturalım.

Temel Bilgisayar Konuları

Burada bilgisayar bilginizi bir adım ileriye götürmek adına bilgisayar mimarisinden ve bilişim dünyasından önemli gördüğümüz başlıkları sizlere anlatacağız. Bilgisayar ile 1998’de tanıştığım için biraz da nostalji yapacağız.

C Diline ve Programlamaya Giriş

Burada algoritma mantığını, program akışını ve bilgisayarın programı nasıl anladığını sizlere açıklayacağız. Programlama dilleri seviyeleri ve bu dillerin

kullanım alanlarından bahsedeceğiz. C dilinin tarihini ve günümüzde kullanım alanlarını inceleyeceğiz ve C dilinin temel mantığını sizlere anlatacağız.

Temel Program Bileşenleri

Burada örnek programları inceleyerek temel program mantığını sizlere anlatacağız. Program akışını, operatörleri ve çalışma mantığını örnek program üzerinde göstereceğiz ve yapısal programlamaya giriş yapacağız.

Program Akışı Denetimi (Döngüler ve Kararlar)

C programlamada büyük bir adım olan döngü ve kararları size bütün ayrıntısıyla açıklayacağız. Gömülü sistemlerde program yazarken en çok kullandığımız yapı olmasının yanında bilgisayar programlamanın ve hatta yapay zekanın temelini de döngü ve kararlar oluşturmaktadır. O yüzden bu konu üzerinde uzun süre duracağız.

C Dilinde Veri Tipleri ve Giriş/Çıkış

Veri tipleri, tipler arası dönüşüm, veri formatları ve bunları temel çıkış fonksiyonları ile nasıl kullanacağınızı anlatacağız. Gömülü C kullanırken printf fonksiyonunu pek kullanmasak da aynı mantığı sprintf’de yine kullanmak gereklidir. Burada anlatılan bütün derslerin gömülü C’de rahatça kullanılabilir olmasına dikkat edeceğiz.

Fonksiyonlar

Fonksiyonlar ileri seviye programlamaya atılan ilk adım olarak değerlendirilebilir. C dilinin kuvvetini de fonksiyon kullanmakla anlayabiliriz. Kütüphane fonksiyonlarını kullanmak ve fonksiyonların mantığını kavramak

çok önemlidir. Fonksiyonların mantığını kavrayan biri kendi kütüphanesini yazarken sıkıntı çekmez ve başka kütüphane fonksiyonlarını da rahatça kavrayabilir.

Diziler

Dizi değişkenleri ileri seviye veri işlemenin ilk adımıdır. C dilinde karakter dizisi olarak “String” yer almadığı için string işlemleri dizi olarak yerine getirilir. Bu yönden dizi konusunda karakter dizilerini de yoğun olarak anlatacağız.

Karakter Dizisi Fonksiyonları ve Metin İşleme

Bu alanda programcılık yeteneğimizi geliştirme adına uygulamalar yapacağız ve C dilinin standart kütüphanesinde yer alan karakter dizisi fonksiyonlarını kullanmayı anlatacağız.

İşaretçiler

İşaretçileri anlamak için donanım bilgisinin gerektiğinin farkına vardım. Çünkü donanım bilmeden önce işaretçileri tam anlamıyla anlayamıyordum. Ne zaman bellek adreslerini, bilgisayar mimarisini yani gömülü sistemleri öğrenmeye başladıysam işaretçilerin ne kadar kolay olduğunun farkına vardım. O yüzden işaretçileri farklı bir açıdan sizlere aktarmaya çalışacağım.

Yapılar, Birlikler ve Operatörler

Operatör konusunda bitwise operatörleri mümkün olduğu kadar ayrıntılı anlatacağız. Çünkü Gömülü C’nin en önemli konularından biri bitwise operatörlerdir. Bütün yazmaç ve bit işlemleri bu operatörler vasıtasıyla yapılır ve bunlar olmadan gömülü sistem üzerinde çalışma imkansız gibidir. Yapı

değişkenleri basit seviyede karşımıza çıkmaya da STM32'nin HAL kütüphanesi gibi gelişmiş yazılımlarda karşımıza çıkmaktadır. Burada basit olarak temeli öğrenmemiz Gömülü C'de işimizi kolaylaştıracaktır.

Derleyici Direktifleri ve Kütüphane Oluşturma

Gömülü C programlamada sürekli karşımıza çıkan bu konuyu eksik bırakmamaya çalışacağız. Aynı zamanda basit kütüphanelerin nasıl oluşturulduğunu anlatarak uygulama aşamasında işinizi kolaylaştıracacağız.

Diğer C konuları

Önemli gördüğüm diğer C konularını anlatarak derslere devam edeceğim. Ayrıca açık kaynak olarak bulduğum örnekleri AVR derslerinde olduğu gibi imkan bulursam satır satır inceleyeceğim.

C programlama hakkında pek çok kaynak olduğu için bu alanda farklı, orjinal ve en iyi kaynağı ortaya koymaya çalışıyorum. Ayrıca zamanım kısıtlı olduğu için kısa sürede yeteri kadar ayrıntıya giremiyorum. O yüzden bu eğitimi uzun bir sürece yaymak gerekiyor. Bu konuda okuyucularımızın gecikmeleri anlayışla karşılamasını umuyoruz.

-2- Bilgisayar Mimarisi Temelleri

Bilgisayar temellerini C programlamaya geçmeden önce sizlere kısaca anlatmak istiyoruz. Biz bilgisayarları kullanıcı olarak içini ve ayrıntısını pek bilmeden karşımıza gelen arayüzü kullanarak az veya çok kullanabiliyoruz. Fakat bir geliştiricinin bilgisayara bakış açısı bir kullanıcının bakış açısından çok daha geniş ve kapsamlı olmalıdır. Bir kullanıcı bakış açısıyla bilgisayara baktığımız sürece asla bir geliştirici olamayız. Bilgisayar sadece onu

kullanabildiğimiz değil aynı zamanda ona hükmettiğimiz bir eşya olmalıdır. Bu aynı araba kullanıcısı ile otomobil tamircisi arasındaki fark gibidir. Kimse otomobil üretecek kadar bilgiye sahip olmasını beklemeyiz fakat bir uzmanın tüm ayrıntısını bilip ona söz geçirmesini bekleriz. Bizim geliştirici olarak belki bilgisayar üretecek kadar bilgiye sahip olmamız gerekme de bilgisayarı kullanmak değil kontrol etmek noktasında bir bilgiye sahip olmamız gerekir. Hazır olan şeyi kullanana kullanıcı adı verilir, geliştirici değil. Bizim de bu noktadan itibaren bilgisayarı en iyi kullanıcıdan daha iyi kullanabilir olmamız gereklidir. Bazen mühendislik öğrencilerinin bile sürücüleri yükleyemediğini, basit hatalara çözüm bulamadığını görüyoruz. Bunlar bilgisayar kullanmayı iyi bilmemekten kaynaklanmaktadır. Biz de dersleri iyi bilgisayar kullananların bilgilerine biraz bilgi katmak amacıyla teknik bilgilere yer vererek devam edeceğiz.

Kullandığımız kişisel bilgisayarlar daha önce mikrodenetleyici mimarisini anlatırken size anlattığımız bilgisayar sistemlerinden çok farklı bir yapıda değildir. Yine merkezi bir işlem birimi olarak CPU, veri hafızası olarak RAM ve program hafızası olarak da Harddisk veya ROM türevi bellekler (SSD) kullanır. Yalnız gömülü sistemler ile kişisel bilgisayarların büyük bir farkı vardır. Bu fark da işletim sistemi (OS) kullanmalarıdır.

Gömülü sistemler de kendi içlerinde işletim sistemi kullanabilir. Üstelik RTOS ve gömülü Linux'ün kullanma oranı da son zamanlarda kapasitenin artmasıyla iyice artmıştır. Fakat gömülü sistemlere genel olarak baktığımızda tek bir program yazılır ve tek bir uygulamayı çalıştırır nitelikte sistemler olduğunu görürüz. Bunlar kişisel bilgisayarlar tarafından yazılan ve derlenen programları çalıştırmaları. Kendi kendilerine programı kendi içlerindeki derleyici ile yazmayız. Bilgisayarda ise bilgisayar uygulamaları “native” yani yerli uygulama özelliği taşımaktadır. Program yazacağımız program için yine kişisel bilgisayardaki

programını kullanırız. Derleyici programların da uygulama programlarının da birlikte çalışmasını ve düzenlenmesini sağlayan “Ana program” ise işletim sistemidir.

Bilgisayar sisteminin merkezi işlem birimine bağlı ROM ve RAM bellekten meydana geldiğini size söylemiştik. Kişisel bilgisayarlarda bu ROM bellek birincil hafıza olarak yine kullanılmaktadır. Harddisk, SSD dediğimiz aygıtlar ikincil hafıza olup giriş ve çıkış birimleri vasıtasıyla sonradan okunmaktadır. Bu giriş ve çıkış birimlerini denetleyecek bir programa başlangıçta ihtiyaç duyulmaktadır. Bilgisayarın asıl işletim sistemi, ilk kullandığı program “BIOS” yani “Basic Input/Output Operating System” adıyla adlandırılır. Temel giriş ve çıkış işletim sistemi anlamına gelen “BIOS” bir rom bellekte anakartın bir köşesinde yer almaktadır. Bu BIOS önceleri bir defa programlanabilir PROM belleklerden olsa da artık Flash BIOS’lar ile yazılım vasıtasıyla BIOS yazılımını güncelleyebiliriz.

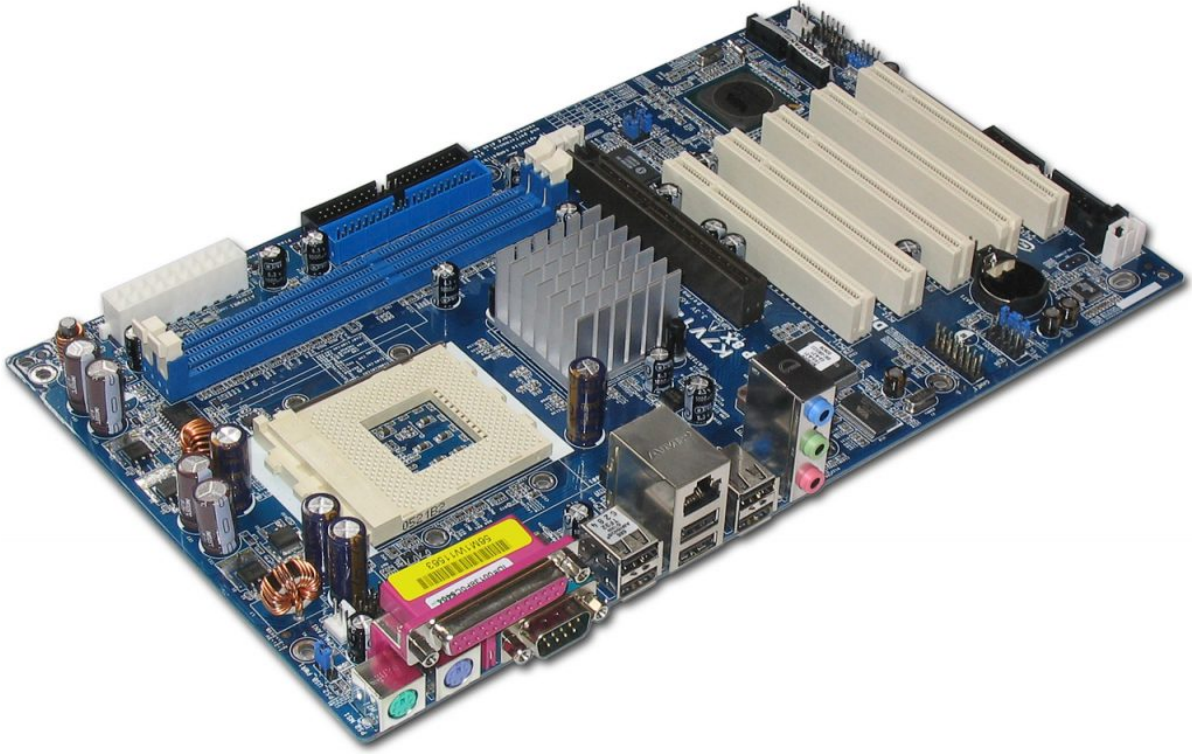


Resimde gördüğümüz BIOS çipinin bir ROM türü bellek olup bilgisayarın ilk açılış programını içinde barındırdığını söylemiştik. Bu açılış programı aynı mikrodenetleyicilerdeki program hafızasına benzer. Bu program hafızası bilgisayarların ayarlarını yapar, giriş ve çıkış birimlerini çalıştırır ve harddiskten işletim sistemi dosyalarını okuyup RAM belleğe aktarır ve bizim ana programımız olan işletim sistemi (Windows, DOS, Linux gibi) çalışır. İşletim sistemleri gömülü sistemlerde yazdığımız programlardan farklı olarak olay ve görev tabanlı çalışmaktadır. Yani pek çok küçük program bu ana program üzerinde çalışır ve görev olarak adlandırılır. Ayrıca bilgisayara giren girişler ve bilgisayardan alınan çıkışlar kesmelere benzer bir yapıda olan “olaylar” tarafından değerlendirilir. Yani bilgisayar bizim fare’nin sol tuşuna tıklamamızı sürekli kontrol etmez fakat fare sol tuşuna tıkladığımızda sinyal bilgisayara gider ve işletim sistemi ilgili görevi yerine getirir. İşletim sistemleri doğrudan donanım ile muhatap olan yazılımlar olduğu için en alt seviye programlama dillerinde yazılan programlar üzerine bina edilmiştir. İşletim sistemlerini farklı platformlarda çalıştırmak istediğimizde mimari engeli ile karşılaşmamız bu yüzdendir. Bilgisayar üzerinde işletim sistemi yazmak için x86 mimarisine ve x86 Assembly diline hakim olmak gerekir. Aynı zamanda en karmaşık sistemlerde Assembly dili kullanıldığı için x86 Assembly programcıları dünya üzerinde oldukça sayılıdır.

Bir donanım üzerinde C dilinde programlama yapabilmek için de önce Assembler adı verilen çevirici program tarafından yazılan C derleyicisine ihtiyaç duyulur. C derleyicisi en basit haliyle C dilinde sembolleştirilen komutları donanıma uygun Assembly diline çevirir ve sonrasında çevirici program bunu makine diline çevirmektedir. Biz C dilinde program yazarken işletim sistemi için yazılan grafik, giriş ve çıkış ve fonksiyonları içeren kütüphaneleri de kullanmamız gerekecektir. Donanıma müdahale için tek başına C dilini değil işletim sisteminin dosyalarını da C diliyle beraber

kullanmamız gerekir. Yine gömülü sistemler üzerinde çalışırken yazmaçlar ve belli bellek adreslerine müdahale ile donanıma erişim sağlarız. Bu C dilinde operatörleri kullanarak adreslere değer atamakla da olmaktadır. Gömülü sistemler ile kişisel bilgisayarlar arasında işletim sistemi farkı olmasından dolayı gömülü sistemlerde donanıma müdahaleyi genelde adresler vasıtasıyla yaparken kişisel bilgisayarlarda işletim sistemi dosyaları vasıtasıyla yaparız. Örneğin bir grafik programı için DirectX veya OpenGL kütüphane dosyalarını kullanırız. Pencere ve temel arayüz yazılımı için windows.h kütüphane dosyasını kullanırız. C dilinde her sistem için standartlaşan stdio kütüphanesi ile temel giriş ve çıkış işlemlerini gerçekleştiririz.

Bilgisayar anakartına baktığımızda yukarıda bahsettiğimiz BIOS (ROM), RAM ve CPU'dan başka bileşenlerin olduğunu da görürüz. Bu bileşenler gömülü sistemlerde bilgisayar sistemleri olmasına rağmen karşımıza çıkmamaları sistemlerin karmaşıklığından dolayı olmaktadır. Şimdi orta seviye ve biraz da eski diyebileceğimiz bir anakartın resmine bakalım ve bileşenlerini inceleyelim.



Bir bilgisayar anakartı bilgisayar sisteminin en kalabalık ve temel parçası olsa da anakartın üzerindeki parçaların görevlerini anlamak çok zor değildir. Öncelikle merkezi işlem birimini tutan bir soket, ram belleklerin takıldığı slotlar ve güç girişi bilgisayar sisteminin asgari elemanlarıdır. Burada I/O portu olarak gördüğümüz PCI, IDE ve SATA portları ikincil cihazların bağlanmasını sağlayan konnektörlerdir. Bu I/O portları I/O denetimcisi olan kuzey ve güney köprüleri ile irtibat halindedir. Kuzey ve güney köprüleri gelişmiş giriş ve çıkış çipleri olup bilgisayarın işlemcisi ile bu elemanlar arasındaki bağlantıyı sağlar. Bundan başka görüntü elde etmek için görüntü işlemcisi ve belleğini bulunduran ekran kartı bu anakartın üzerine takılmaktadır. Çoğu ağ ve ses denetleyicileri tek entegre olarak kart üzerinde lehimli halde gelir. Bu entegreler de giriş ve çıkış denetimcileri ile irtibat halinde olup donanım sürücülerini vasıtasıyla kullanılabilir hale gelir. BIOS sistemini anakart üreticilerinin yazıp karta entegre etmesinin sebebi ise bütün bu giriş ve çıkış denetiminin sorumluluğunun kendi üzerlerinde olmasıdır. İşlemci üreticisi

sadece işlemciyi üretse de bu işlemcinin düzgün çalışabilmesi için anakart üreticilerinin düzgün bir anakart tasarlaması gereklidir.

Bu yazıdan bilgisayar mimarisini ve işletim sisteminin prensibini çok özet bir şekilde öğrendik. Bilgisayar mimarisini ve donanımını anlatacağımız ayrı bir yazı dizisi düşünüyoruz fakat şimdilik programlamaya başlayacakların fikir edinmesi ve ufkunun açılması için bundan kısaca bahsettik. Bir sonraki yazıda işin yazılım tarafıyla daha çok ilgileneceğiz ve ardından programlama mantığını anlatmaya başlayacağız.

-3- Bilgisayar Yazılımı Temelleri

Alt seviye programlama yaparken bilgisayar yazılımından çok mikroişlemci yazılımı tabirini kullanmamız gerekir. Çünkü biz bilgisayar kasasına veya anakarta yönelik değil doğrudan doğruya mikroişlemcinin çalıştıracağı komutları yazmaktayız. Donanım soyutlamasının tam manasıyla görüldüğü yüksek seviye diller ve geliştirme ortamlarında o platforma göre yazdığımız yazılım bile temelde bir alt seviye programın ürettiği yazılım ile aynı özelliği taşır. Eninde sonunda mikroişlemci komut kümesini işleten ve 1 ve 0'lardan oluşan makine kodu bilgisayar yazılımını çalıştırmaktadır.

Gömülü sistemlerde programın çalışması için .hex uzantılı ve makine dilindeki kodu mikrodenetleyicinin program hafızasına yükleriz. Yani programın program olarak çalışması için öncelikle metin editöründe metin halinde bulunan ve programlama dilinde yazılmış olan programın makine koduna çevrilmesi gerekir. BASIC, Python gibi dillerde çevirmen yardımıyla (interpreter) derlenmeden de program çalıştırılabilmektedir. Ama bizim alışkın

olduğumuz C, C++ gibi dillerde programın muhakkak makine diline çevrilmesi gereklidir. Yani C dilinde mikrodenetleyici için program yazıp bunu makine dilinde çalıştırmakla bilgisayar için C dilinde uygulama yazıp bunu çalıştırmak arasında çok fark yoktur. .hex uzantılı program burada yerini .exe uzantılı programa bırakmaktadır. İşletim sistemi ise bu makine kodunu çalıştıran programı kısıtlar, denetler ve düzenler. Örneğin bu program her makine kodunu istediği gibi çalıştırıp her adrese erişim sağlayamaz. Diğer programların ve ana programın (işletim sistemi) sağlıklı çalışması için bu uygulamaya bir kısıtlama getirilmesi gereklidir. Arada güvenlik açığı olur da bu programlar kritik noktalara ulaşırsa bunlara “bilgisayar virüsü” adı verilir. Bilgisayar virüslerinin normal bir programdan hiçbir farkı yoktur fakat çalışma mekanizması farklıdır.

Bilgisayarlarda işletim sisteminin olmasının en büyük sebeplerinden biri de bu güvenliği sağlayabilmesidir. Güven vermeyen işletim sistemleri işletim sistemi olabilmeyi tam anlamıyla yerine getiremez. O yüzden Linux tabanlı işletim sistemleri virüssüz ve güvenli olmalarından dolayı işletim sisteminde olması gereken özellikleri daha iyi karşılamaktadır. İşletim sistemlerine baktığımızda ise pek çok çeşitte işletim sistemlerini görsek de üç ana işletim sisteminin günümüz bilgisayarlarında etkin olarak kullanıldığını görürüz. Bunlardan biri başlarda DOS tabanlı olarak piyasaya sürülmüş olan Windows diğer ikisi ise Unix tabanlı olan Linux ve MacOS işletim sistemleridir. Windows şu an DOS tabanlı olmasa da Windows 98 kullanırken DOS programlarına çift tıklayarak programları açmamız mümkündü. Windows ve MacOS işletim sistemleri kapalı kaynak olduğundan perde arkasında ne olduğunu bize söylememekteler. Yalnız bunların kapalı kaynak olması onları kullanarak program yazamayacağımız anlamına gelmez. Windows’un kütüphanelerini, derleyicilerini ve öğelerini kullanarak yine programımızı yazabiliriz. Yazdığımız program işletim sistemi platformunda olduğu için işletim sistemi altında

çalışacaktır. O yüzden Linux programları Windows'da, Windows programları da Linux'de çalışamaz. Aynı mimari, aynı platform ve aynı donanım olmasına karşın makine dilinde yazılan programlar işletim sistemine ait parçaları kullandığı ve dosya sistemleri farklı olduğu için birbiriyle uyumlu olmamaktadır.

Biz Linux işletim sistemi üzerinde program yazmak istediğimizde ilk olarak GNU kavramıyla karşılaşırız. GNU, FSF yani özgür yazılım vakfının en büyük projelerinden biridir ve bir açık kaynak derleyici arşividir. Bu derleyicileri herhangi bir şirkete veya platforma bağlı kalmadan özgürce ve herkes kullanabilir. Windows'da program yazmak istediğimizde ise karşımıza genelde Visual Studio ve .NET Framework çıkmaktadır. Bu Microsoft'un kendi ürünlerine program yazmak isteyen geliştiricilerine verdiği araç ve kütüphanelerdir. Visual Studio'yu geliştirerek ve kütüphaneleri artırarak geliştiricileri elinde tutsa da Windows işletim sisteminde farklı bir derleyici ve geliştirme stüdyosu ile program yazan bir derleyici de yine eninde sonunda Windows öğelerini ve kütüphaneleri kullanmak zorunda kalacaktır. Çünkü donanımı denetleyen işletim sistemi olmaktadır.

Visual Basic, Visual C++, Visual C# Windows'da masaüstü program yazmakta tercih edilen yöntemlerden olmuştur. Visual Basic ve C# hakkında 1000 sayfaya yakın pek çok Türkçe kitap yazılmıştır. Fakat bugün uzun yıllardır kitap yazan bir yazarın Visual Basic kitabını incelediğimde F1 yardım menüsünün tercümesinden ibaret olduğunu gördüm. Pek çok hazır fonksiyonu, kütüphaneyi ve özelliği içeren bu platformlarda programlayıcı programlama dilini çok bilip anlamasa da hazır olan parçaları birbirine birleştirerek program yazabilmektedir. O yüzden bilişim dünyasında "buton altı programcılık" denen bir tabir ortaya çıkmıştır. Bu tabir Visual Studio gibi ortamlarda pencere üzerine birkaç düğmeyi sürükleyip düğme fonksiyonlarına

birkaç satır ezbere ya da kopyala yapıştır kod yazma işini bize anlatmaktadır. Böyle bir yazılımcılık ve geliştiriciliği asla tasvip etmiyoruz. Siz C ve C++ dillerini öğrendiğinizde asla tek bir platforma veya firmaya bağlı kalmayacaksınız. İstedığınız derleyicide, kütüphanede, platformda özgürce program yazabileceksiniz. Çünkü C ve C++ dilleri herhangi bir firmanın ticari ürünü değildir. O yönüyle kendileri günümüzde kullanılan çoğu dilden çok daha ayrıcalıklı bir noktadır.

Yazılıma dair önemli gördüğümüz noktaları bu başlık altında sizlerle paylaştık. Bir sonraki başlıkta programlama dilleri hakkında bir karşılaştırma yapacağız ve yazılım çeşitlerinden bahsedeceğiz.
























-4- Programlama Dilleri

Programlamaya giriş yapmak istediğimizde hangi programlama dilini öğrenmemiz gerektiği konusu kafamızı oldukça kurcalar. Ben programlamaya 2007-2008 yıllarında meraklanmış ve Visual Basic üzerinden basit programlar yapmaya başlamıştım. Tabi Visual Basic basit programlar yapmaya yarıyordu ve internette Türkçe Visual Basic kaynakları da hesap makinesi yaptırmaktan öte gidemiyordu. Ben de hesap makinesi yaptıktan sonra sıkılıp bırakmıştım. İşin önemli yanı Visual Basic bize gerçekten program yazmayı ve algoritma kurmayı öğretmiyordu. Düğmelerin, metin kutularının arkasına yazılan fonksiyonlar olay tabanlı çalışıyor ve program akışı ve algoritmadan bizi uzak tutuyordu. Daha sonrasında ilk öğrenmeye çalıştığım programlama dili C++ oldu. Burada yine C++ hakkında Türkçe kaynak olmadığı için tercüme bir kitap almıştım. Halen de C++ hakkında kapsamlı bir kitap yazılmış değildir. O dönem bir veya iki Youtube kanalından video izleyerek bunu ilerletiyordum. Şimdi video izleyerek öğrenmeyi tasvip etmesem de o dönem ulaşabileceğimiz tek kaynak olarak bulunmaz nimetti. Lisede Web programcılığına yönelik eğitim görüp HTML, ASP.NET ve PHP üzerinden programlamaya devam etsem de benim asıl öğrenmem gereken dilin en başta C olması gerektiğini daha sonra öğrenebildim. C++'nın ileri seviye konuları hariç C kısmıyla aynı olan yerlerini zaten bildiğim için C öğrenmem zor olmadı. Gömülü sistemler üzerinde çalışmaya başladıktan sonra ise bu alan için C dilinin tek başına bile yeterli bir dil olduğunun farkına vardım.

C dili öğrenmem uzun zaman içerisinde yayılan bir süreç halindeydi. Çünkü bir dili öğrenmek farklı o dili sindirmek farklı bir durumdur. Diğer dillerde olduğu gibi C dilini öğrenmek kolay olsa da anlamak için zamana ihtiyaç vardır. Bu anlama süreci ezbere kod yazmaya devam edildikçe oldukça gecikmektedir. Belki hiç anlamadığı halde o dilde yıllar boyu program yazan programcılığı

görebiliriz. Gömülü sistemlerde ise anlamadan ve ezbere kod yazma imkanı yoktur. Çünkü yazdığınız kodun en değerli olduğu yer gömülü sistemlerdir. Örneğin Web tasarımında HTML ve CSS kodları yüzlerce satırdan oluşabilir ama ortaya çıkan ürünün değeri az olur. O yüzden çoğu zaman da oturup sıfırdan kod yazılmaz belli başlı şablonlar ve hazır kodlar kullanılır. Gömülü sistemlerde ise yapılan ürün değerli olduğu gibi yazılan kod da değerlidir. 5-10 bin liralık bir ürünün programı 200-300 satır koddan oluşabilir. Ayrıca Web tasarımında olduğu gibi kalitesiz koda ve hataya asla yer yoktur. Yazdığınız program kodu doğru olmasının yanında kaliteli de olmak zorundadır. Gömülü sistemler üzerinde çalışırken ben her gün oturup yüzlerce satır kod yazmıyorum. Şu an bir yılda 5-6 düzgün program yazmam işimde yeterli oluyor.

Gömülü sistemlerde çalışmak isteyenlerin öğrenmesi gereken en önemli dil olan C dilini teorik olarak öğrenmek kolay olsa da bunu uygulamaya dökmek zor olabilir. Bilgisayar uygulamasında C++ ağırlıkta, gömülü sistemlerde ise C ağırlıktadır. Yani gömülü sistemler için C kullanmak istesek bile C eğitimleri bilgisayar üzerinden ve bilgisayar programı yazarak gerçekleşmektedir. C dilinde ise bilgisayar alanında pek kullanım olmadığı için ortada kalacağımız düşüncesi olabilir. Biz masaüstü programlarda neden C/C++ tabirini kullandığımızın cevabı da buradadır. Bilgisayar ortamlarında iki dili beraber kullanmamız şart gibidir.

Language Rank	Types	Spectrum Ranking
1. Python	 	100.0
2. C	  	100.0
3. Java	  	99.4
4. C++	  	96.9
5. C#	  	88.6
6. R		88.1
7. JavaScript	 	85.3
8. Go	 	75.7
9. Swift	 	74.3
10. Ruby	 	72.0

Burada en popüler programlama dillerinin bir listesini görmekteyiz. Bu popülerlik her siteye veya platforma göre değişmektedir. Github'daki popülerlik çok daha farklı olabilir. Bu popülerliğin sizi aldatmaması gereklidir. Çünkü en popüler olan en iyi veya en değerli anlamı taşımamaktadır. Bazı programlama dilleri bir anda parlar ve sonrasında modası geçer. Tarihte bir döneme damgasını vurmuş pascal gibi programlama dillerini artık günümüzde görememekteyiz. Modaya bağlı olmayan, bir şirketin arkasında olmayan ve piyasada her zaman kendinden söz ettiren iki dili bu liste arasında sayabiliriz. Bunlardan biri C diğeri ise C'nin nesne tabanlı ve gelişmiş sürümü diyebileceğimiz C++ dilidir. Bu iki dilin diğer diller arasında seçkin bir yeri olmasının bir sebebi vardır. Günümüzde Python, PHP gibi diller bile C/C++ ile yazılmıştır. Programlama dili yazdığımız bir programlama dilinin modasının geçmesi bu durumda mümkün değildir. C/C++ dilleri ile oyun motoru, internet tarayıcı, photoshop, CAD programları, işletim sistemi, bilimsel programlar ve dahası yazılmaktadır. C ve C++ zor, derin bilgi gerektiren ve diğerleri gibi

kullanıcı dostu olmayan bir yapıda olsa da bu dilleri etkin olarak kullananlar diğer programcılar arasında ayrı bir yere sahiptir.

Yazının ikinci kısmında ise programlama dillerini dil yapısı yönünden kısaca inceleyelim. Programlama dillerinin meydana gelmesinde iki önemli dil unsuru vardır. Bunlardan biri makine dili öteki ise İngilizce'dir. Programlama dillerini incelediğinizde alt seviye dillerin makine dili mantığına yakın olduğunu üst seviye dillerin ise İngiliz dili mantığına yakın olduğunu görürsünüz. Bu durumda İngiliz dili mantığını iyi bilen birisi üst seviye programlama dillerini anlamada sıkıntı çekmez. Örneğin "if" deyiminin en başta yer alması gibi İngilizce cümlelerde de "if" deyimi başta yer almaktadır. Bizde ise -sA eki ile bu durum sağlanmaktadır. Bunun gibi bütün üst seviye programlama biçimleri Türkçe'nin söz dizimine uyum sağlamaz. Fakat İngilizce'deki cümle söz dizimlerine benzer yapıdadır. O yüzden özellikle ana dili İngilizce olan birisi adeta konuşur gibi program yazabilir. İngiliz dili mantığını bilmeyen biri içinse program yazmak makine kodu yazmaktan farklı gelmez. O yüzden program yazarken İngilizce bilmenin önemi sadece İngilizce kaynaklara erişimde değil programlama dilini anlamada da etkili olmaktadır.

Makine diline geldiğimizde ise bu dilin mantığı insan dili mantığından uzak bir noktadadır. Daha çok matematik ve mantık işlemlerine yakın olan bu makine dilini öğrenmenin yolu ise makinenin çalışma mekanizmasını anlamaktan geçer. Makine dilini öğrenmek derken Assembly dilinden bahsettiğimizi bilmeniz gerekir. Çünkü Assembly dili temelde makine dilinin alfanümerik kodlanmasından öte bir şey değildir. Bu kodlanma yine İngilizce'ye çevirmekle olmaktadır. Örneğin makinenin toplama işlemi yapması için gerekli kod "0011 0110" ise bunu Assembly'e çevirdiğimizde "ADD" yani İngilizce "Ekle" olarak çeviririz. Bu durumda ADD komutu benzeri komutların kelime anlamı dışında bir yabancı dil unsuru Assembly dilinde görülmemektedir. Çok yüksek seviye

diller iyice İngilizce mantığına yaklaşmış ve donanımdan soyutlanmış dillerdir. O yüzden yüksek seviye dilde programlama yaparken bilgisayar mantığını anlamamız zor olacaktır.

Alt seviye programlama program akışında çok fazla dil unsuru gerekmesi de bunlar hakkında bilgiyi elde etmek için yine yabancı kaynakları okuyabilecek kadar yabancı dil bilmek gereklidir. İşin ilginç yanı buradadır. Örneğin Excel programında yüksek seviye programcılık yapılsa da Türkçe fonksiyonlar olduğu için oldukça işimiz kolaylaşmaktadır. Fakat diğer yüksek seviye dillerde Türkçe kaynak olsa da fonksiyonlar yabancı dilde olduğu için bunları kavramak yabancı dil bilmeyenler için güç olmaktadır. C dilinde ise yabancı dil bilmekten çok algoritma ve program mantığını bilmek gereklidir. Bu noktada C dilini Türkçe kaynaklardan öğrenip bir noktaya gelenler bile bu dil üzerinde çalışabilmek için yeterli İngilizceye sahip olması gereklidir. Çünkü kullanılacak kütüphaneler ve platformlar hakkında Türkçe kaynak yok gibidir. Biz sadece C dilini öğrenmekle kalmayıp bunu kullanmayı da anlatacağız.

-5- Makine Dili, Assembly Dili ve Yüksek Seviye Diller

Programlama dilleri arasında C dili özel ve önemli bir yere sahiptir. Bunu anlamak için öncelikle dil seviyelerini bilmek gereklidir. Dilleri makine dili, assembly dili ve yüksek seviye diller olarak ayırabiliriz. Makine dili tamamen sayısal biçimde kodları işler ve herhangi bir insan diline benzerliği yoktur. Örneğin 0x2525, 0x5542 gibi komutların yan yana sıralanmasıyla makine programı meydana gelir. Burada 0x onaltılık değer örnekidir. Bu değerler onaltılık sayı olarak da ikilik sayı olarak da hatta onluk sayı olarak da temsil edilebilir. Fakat işin temelinde 1 ve 0'lardan oluşan ikilik sayılar vardır.

Makine Dili

Makine dilinde program yazmak çoğu programcıya göre imkânsız görünse de biz gömülü sistem geliştiricileri için imkânsız değildir. Fakat etkili bir çözüm de değildir. Biz makine dilini mikroişlemcinin komut seti kılavuzundaki opcode açıklamalarından öğrenebiliriz. Aynı zamanda Assembly dilinde yazdığımız programı makine koduyla karşılaştırmaya çalıştığımız yazılımlar mevcuttur. Bütün bunlara rağmen alt seviye programlamada çalışmak oldukça zordur ve yüksek seviye donanım bilgisi gerektirmektedir. İlk zamanlarda bilgisayar programları makine dili ile yazılmaktaydı.

```
:10010000214601360121470136007EFE09D2190140  
:100110002146017E17C20001FF5F16002148011928  
:10012000194E79234623965778239EDA3F01B2CAA7  
:100130003F0156702B5E712B722B732146013421C7
```

Yukarıda makine dilini görmekteyiz. Yüksek seviye programcılar için anlamsız harf ve sayı topluluğu gibi görünse de işinde ilerlemiş bir alt seviye programcı için anlaşılmaz değildir.

Assembly Dili

Assembly makine dilinin alfanümerik olarak temsil edilmesine denir. O yüzden tek başına bir dil değildir. Ne kadar makine ve donanım varsa o kadar fazla Assembly dili bulunmaktadır. O yüzden PIC Assembly, AVR Assembly, 6502 Assembly diyerek farklı farklı Assembly dillerinden bahsederiz. Tek bir Assembly dili öğrenip bütün makineleri Assembly'de programlamak imkânsızdır. Şu vardır ki makinelerin mimarisi prensipte birbirine benzemektedir. Bunu daha önce mikroişlemci mimarisini anlattığımız makalelerde ayrıntısıyla sizlere açıkladık. Assembly dili makine dilinin harf ve rakamlarla temsil edilmesi olmasına karşın bu seçilen harfler insan dilinde anlamlı kelimeler veya onların kısaltmaları olmaktadır. Örneğin makine dilinde toplama işleminin komutu 0x25 ise bu Assembly dilinde İngilizce ekle anlamına gelen ADD komutu olur. Böyle böyle makine dilindeki komutlar İngilizce anlamlarına göre harf olarak temsil edilmektedir. Aynı zamanda girilen sayısal değerler ve yazmaçlar yine insanın anlayacağı şekilde düzenlenmektedir. Örneğin 0x0002 adresindeki yazmaç R2 yani Register 2 olarak adlandırılmaktadır. Opcode içerisinde dağılmış ve sıkışmış sayı değerlerini anlamak için çoğu zaman 1 ve 0 ları saymak lazımdır. Assembly dilinde ise 0x55 diye bu sayı değerini yazma imkanımız olur. Yine de donanım bilmeyen yüksek seviye programcı için Assembly dili oldukça anlaşılmaz olmaktadır. Alt seviye programcılar için de zor olsa da alıştıktan sonra fazla sıkıntı olmamaktadır. Şimdi bir Assembly örneği verelim ve bunu inceleyelim.

```
ADD      R16, R17
DEC      R17
MOV      R18, R16
END:     JMP      END
```

Burada komutların ve operandların satır satır yazıldığını görüyoruz. Bu BASIC dili ile benzerlik göstermektedir. Örneğin toplama işlemi yapacaksak ADD komutunu kullanmalıyız. Fakat toplama yani ADD komutu neyi toplamalıdır?, sorusunun cevabını operandları yani işlem yapılacak parçaları yazarak veririz. ADD R16, R17 komutunun anlamı R17 yazmacındaki değeri R16 yazmacına EKLE anlamı taşımaktadır. Burada insan dilinden farklı olarak soldan sağa değil sağdan sola bir okuma vardır. İngiliz dil mantığında bile R16'yı R17'ye eklemesi gerekirken burada bu böyle gerçekleşmemektedir. Önceki yazıda bahsettiğimiz gibi programlama dillerinin İngilizce'ye benzerliği burada R16(R Register'in kısaltması), ADD (Ekle) gibi çeviriden ibarettir. Üzerinde çalıştığımız dil ise AVR Assembly dilidir. Başka bir donanım üzerinde yazılmış Assembly diline baksaydık benzer kelimeleri görsek bile bambaşka komutlarla karşı karşıya kalacaktık. DEC (Decrement) yani azalt anlamına gelmektedir. DEC R17 komutunun işleyişini buradan tahmin edebiliriz. R17 yazmacındaki değeri BİR AZALT anlamı taşımaktadır. Burada yine sağdan sola okunduğuna dikkat edin. MOV ise İngilizce çoğumuzun bildiği "Move" yani taşı anlamına gelmektedir. Burada da "Move" kelimesinin anlamında bir işi gerçekleştirmektedir. MOV R18, R16 komutunun anlamı R16 yazmacındaki değeri R18 yazmacına TAŞI anlamına gelir. JMP ise "Jump" yani ATLA komutudur. Program akışını belirlenen etikete götürmek için kullanılır.

Şimdi Assembly diline baktığımızda aslında çok zor olmadığını görmekteyiz. Zor olan bir diğer nokta makine dilinin işleyişinde yatmaktadır. Makine bizim yazdığımız matematik, mantık ve fonksiyonları bizim gibi işlememektedir. Yani toplama işlemi yapmak istediğimizde $A + B = \text{Sonuç}$ yazma imkânımız yoktur veya mantık işlemi yapmak istediğimizde boolean operatörlerini kullanamayız. Bu durumda bizim matematik, mantık ve dile dair bilgilerimiz burada işimize pek yaramamaktadır. Yüksek seviye dillerin insan diline yakınlığı sadece İngilizce tabirlerden ve söz diziminden ibaret değildir. Görüldüğü gibi toplama işlemi için ADD A, B yazmak bile $A + B$ yazmaktan bize daha yabancı gelmektedir.

Yüksek Seviye Diller

Yüksek seviye diller Assembly dilinden başka donanım soyutlamasını artırmış hemen her dil için kullanılmaktadır. Burada bütün dillere yüksek seviye dil demek aslında pek doğru olmamaktadır. Orta seviye, yüksek seviye ve çok yüksek seviye programlama dillerinden bahsedebiliriz. Burası tartışılan bir konu olsa da bu seviye adlandırma konusunda değişik kaynaklar değişik bilgiler vermektedir. Örneğin Python, Visual Basic, Perl, Ruby gibi diller çok yüksek seviye programlama dilleri arasındadır. Bu dillerde programlama yapmak için bilgisayarın inceliklerinden değil uygulamanın inceliklerinden haberdar olmak yeterlidir. Daha önce “buton altı programcılık” için kullanılan Visual Basic’den bahsettiğim gibi düğmenin altına bir kaç fonksiyon yazmanız program yazmak için yeterli oluyordu. Bu fonksiyonları ise o yazılımı öğrenmekle öğrenebiliyordunuz. Hatta ezberlediğiniz komutları ve fonksiyonları anlamadan da olsa kullanmanız mümkündü. Çünkü çok yüksek seviye diller az bilgi çok iş gücü esasına göre çalışmaktadır. Bazen bizim alt seviyede 1 saatte yaptığımız aynı işi yapan kodu yüksek seviyede 1 dakikada yazmak mümkündür. Gömülü sistemlerde Arduino’dan ileri PARSIC, Mblok

gibi sadece kutucukları lego gibi birleřtirip program yazdıđımız ortamlar vardır. Bunların ok yksek seviye programlama sunması donanımın zelliklerini tam anlamıyla kullanamama gibi bir dezavantaj dođurmaktadır. Yazılımın bize verdiđi 10-20 fonksiyonla avunup kendi fonksiyonlarımızı, ktphanelerimizi yazamama gibi bir durumda kalmamak iin iřin derinini đrenmek řarttır. Genelde ok yksek seviye diller kendilerine ait programlama arayzleri ile beraber gelmektedir. Python gibi diller ise bu konuda olduka iyi bir konuma gelmiřtir nk genel maksatlı ve her alanda kullanıma uygun bir noktadadır. Hatta programlamayı đrenmek iin Python ile bařlamak iyi bir seenek olabilir. Belki biz de ileride bu dili inceleriz.

Yksek seviye diller arasında ise gnmzde C#, Java, C++ gibi diller bulunmaktadır. C dilini ierdiđi alt seviye zelliklerden ve gml sistemlerde kullanılmasından dolayı orta seviye olarak da nitelendirmek mmkndr. Bazı kaynaklarda C dili orta seviye bir dil olarak nitelendirilmektedir. nk C dili C# veya Java gibi yksek seviye dillerden olduka farklı bir yapıya sahiptir ve C dilinden bir ařađısı Assembly dili olmaktadır. C dili Assembly dili ile aramızda olan bir kpr olduđu iin bunu orta seviye dil olarak deđerlendirmek C dilinin programlama dilleri arasındaki sekin konumuna vurgu yapmak demektir. Ama literatre baktıđımızda low-level, Assembly ve high-level olmak zere  ayrı terim mevcuttur. Yani Visual Basic ile C dili aynı kefeye konup yksek seviye dil olarak tanımlanmaktadır.

-6- C Diline Giriş

Temel C Programlama derslerimizin altıncısında C diline giriş yaparak devam ediyoruz. C diline güzel bir giriş yaparsak C dilinde ilerlemek daha kolay olacaktır. Çünkü C dilini anahtar kelimeler, kurallar ve fonksiyonlar bütünü olarak telakki etmek programlama dillerini anlamakta bize fayda sağlamayacaktır. Daha önceki yazılarımız da C dilini doğru biçimde anlamanız için yardımcı niteliktedir. Şimdi C dilinin anahatlarından bahsederek dile geçiş yapalım.

C dili genel maksatlı bir programlama dilidir. Genel maksatlı bir dil olmasının önemli bir yanı vardır. Bazı diller belli amaçlara yönelik üretilmiş dillerdir. Örneğin FORTRAN dili matematik ve bilimsel hesaplamalara yönelik çıkarılmış bir programlama dilidir. Bazı programlama dilleri genel maksatlı olsa bile bazı platformlar ve sektörler ile sınırlandırılmıştır. Örneğin Ada dili havacılıkta, Objektif-C dili ise Apple ürünlerine program yazmada kullanılır. C dili ise gerçek manada genel maksatlı bir programlama dilidir. Ne bir platforma ne de bir sektöre bağlıdır. C dili ile hemen her alanda her programı yazma imkânımız vardır. Hatta C dili ile programlama dilleri bile yazılmaktadır. Bunun tersi ise amaca yönelik (domain specific) programlama dilidir.

C dili emirli (imperative) bir dildir. Yani C dilinde her yazdığımız komut satır satır emirlerden oluşmaktadır. Makine veya Assembly dilinde olduğu gibi C dilinde de belli bir program akışına göre bu sırasıyla bu emirler işletilir. Dikkat edilmesi gereken nokta bu emirlerin işin bütün ayrıntılarını içermesidir. Emirli dillerin karşısında tanımlamalı diller bulunur. Bu dillerde program yazarken işin iç yüzünden çok belli amaca göre kodlanmış program parçaları kullanılır. Örneğin emirli bir programlama dilinde şöyle bir komut işletilir. [1]

- Kapıyı 80 cm ileri doğru ittir

-

Tanımlamal dilde ise şu şekilde olur,

- Kapıyı Aç

-

C dili yapısal bir programlama dilidir. Bunun karşısında nesne tabanlı programlama dillerini sayabiliriz. Yapısal programlama dili olması bizi donanımına daha yaklaştırmaktadır. Çünkü makine dilinde komutlar sırayla işletilir, belli mantıksal işlemler yapılır, belli bloklar olur ve bu bloklar arasında alt yordam (subroutine) adı verilen program akışı gerçekleşir. Yapısal programlamada da nesne ve sınıflar gibi alt seviye dillerden ve program mantığından uzak parçalar yoktur. O yüzden C dilinde program yazan birisi çok ileri programlama yöntemleriyle uğraşmaz. Basit ve etkili program yazar. Bu bakımdan C dilini öğrenmek C++, C# gibi nesne tabanlı dilleri öğrenmekten daha kolaydır. Yapısal programlama dilleri üç elementten oluşmaktadır. Bunlar kontrol yapıları, alt yordamlar ve bloklardır. Yapısal programlama dillerini algoritma akış diyagramlarına uyarlamak çok kolaydır. Yalnız program kalabalıklaştıkça (karmaşıklılaştıkça demiyorum) yapısal programlama dillerinde program yazmak zorlaşmaktadır. O yüzden nesne tabanlı programlama dilleri ortaya çıkmıştır.

C dili prosedürel bir programlama dilidir. Prosedürel programlama yapısal programlamadan türetilmiştir. Prodesürel programın temelleri alt yordamlar, yordamlar (routine) ya da fonksiyonlardır. Bu fonksiyonlar ifadelerin (statement) bir araya getirilmesinden meydana gelen program bloklarıdır ve bu program blokları arasında bir akış ve atlama ile program çalışmaktadır. Bu programlama yönteminde modülerlik ve kapsam önemli bir yerdedir. Modülerlik büyük programı küçük parçalara ayırıp kullanmak anlamı taşıırken

kapsam ise bu prosedürleri modül halinde tutmaya yardımcı olmaktadır. Bunları daha ilerleyen konularda uygulamalı olarak göstereceğiz.

C dili alt seviye ve üst seviye programlama arasında bir köprüdür.

Donanım seviyesi bakımından bu kadar esnekliği sağlaması C dili ile hem performanslı hem de karmaşık uygulamaları yapma imkanı vermektedir. Pek çok oyun motoru ve donanımın sınırlarını zorlayan program C/C++ dilinde yazılmaktadır. C dili hafızaya adres adres erişebildiği gibi bit operatörleri ile bitlerine müdahalede bulunmamıza olanak sağlamaktadır. O yüzden çoğu zaman Assembly dilini aratmamaktadır çünkü donanım ne yapabiliyorsa biz bunu C dili ile yapabiliriz. Eksik komut olması, kütüphane zorunluluğu gibi meseleler burada yoktur. Fakat performans konusunda ve hassas noktalarda yine Assembly dilini kullanmamız gerekebilir. Buna da büyük oranda ihtiyaç duymayız.

C dili kullanım alanı en geniş dillerden biridir. Tarih boyunca C dilinde onlarca farklı işletim sisteminde, yüzlerce farklı donanımda program yazılmıştır. Basit bir ev bilgisayarından süperbilgisayarlara kadar. Basit programlardan bilimsel hesaplamalara kadar her alanda C dilinin kullanıldığını görebiliriz. Örneğin C# dili Windows platformunda çalışmakta, Objective-C dili ile Apple cihazlarına program yazılmaktadır. C dilinde ise böyle bir sınırlama yoktur. C dilinde oyun, metin işleme programı, otomasyon programı işletim sistemi, kontrol programı gibi onlarca kullanım alanı ve sektöre yönelik program tipleri yazılabilir. Yalnız bazı alanlarda C dilinin pratiği yoktur. Örneğin Web programcılığında C değil de PHP dili kullanılır. Fakat PHP dilinin C dili ile yazılmış olduğunu hatırlatalım.

C dilinin tarihine baktığımızda Dennis Ritchie tarafından 1972-1973 yıllarında Bell Laboratuvarlarında geliştirildiğini öğreniyoruz. Bu programlama dilinin

geliştirilmesi Unix işletim sistemi yazılırken olmuştur. Çünkü Assembly dili ile tam kapsamlı bir işletim sistemini yazmak aşırı derecede zordur ve işin içinden çıkılmaz noktaya varılır. C dili Unix işletim sisteminin yazılmasında ve bu işletim sistemine yönelik araçların geliştirilmesinde yardımcı olmuştur. C dili ilk geliştirildiği zamanlar daha bir standarta sahip olmadığı için popülerlik kazanması 1990'ların başına kadar uzamıştır. Bunun bir sebebi de C dilinde program geliştirmek yerine Assembly dilinde program geliştirmenin zorunluluğudur. Bu zorunluluğun ortaya çıkmasında donanımların yeteri kadar güçlü olmaması ve C gibi performanslı dilin bile bu donanımlar için optimal programı ortaya çıkaramamasıdır. Hatta 1993 yılında çıkan ve çıktığı zamana göre grafik olarak hem yeni bir teknoloji hem de büyük ilerleme getiren DOOM oyunu sadece C dili ile yazılmış değildir. Oyunun kaynak kodlarını ID software günümüzde paylaşmıştır. Oyunun kaynak kodlarını incelediğimizde C ile Assembly dilinin beraber ve etkin bir kullanımını görürüz.

C dilinin performans ve esnekliği sayesinde ileri seviye programlar yazılabilir. Bunlar C#, Python gibi dilleri kullanan geliştiricilerin ilgisini çekmese de alt seviye programcılarının ilgisini bir hayli çekmektedir. Örneğin işletim sistemleri, gömülü sistemler, gerçek zamanlı sistemler ve iletişim sistemleri gibi hassas ve kritik uygulamaları alt seviye programcılar yapmaktadır. Bunları kütüphane fonksiyonlarını ezberleyip kullanan programcılarının yapmasının imkanı yoktur. O yüzden C programlama dili ile ezbere ve kopyala-yapıştır programcılık yapmanın imkanı yoktur.

C dili günümüzde kullanılan dillere öncü olmuştur. Şu an kullanılan Objective-C, C#, Java ve C++ dillerinin atası C programlama dilidir. O yüzden C programlama dilini öğrenen birisi bu dilleri öğrenmekte zorluk çekmez. Yani hem gömülü sistemler üzerinde çalışma hem de bu bilgiyle daha yüksek

seviye ortamlarda çalışma imkanı buluruz. Yüksek seviyenin her zaman en iyi olmadığını şimdiye kadar anlamış olmanız gerekir.

Bu noktada sizlere C dilini anahatlarıyla tanıtmış olduk. Bir sonraki makalede C diline ait diğer özellikleri ve geliştirme ortamlarını sizlere anlatacağız.

Kaynaklar,

[1} Emirli Programlama (Imperative Programming), Şadi Evren ŞEKER,
<http://bilgisayarkavramlari.sadievrenseker.com/2009/11/16/emirli-programlama-imperative-programming/>, Erişim Tarihi 28.02.2019

-7- C Geliştirme Aşamaları ve Derleyici Kurulumu

C dilinde program yazmaya başlamadan önce bu yazdığımız programın hangi süreçlerden geçerek program haline geldiğini öğrenmemiz gerekir. Biz alt seviye programcı olduğumuz için işin derinini öğrenmemiz her zaman şarttır. O yüzden temel seviyede bile işi basitleştirme taraftarı değiliz. O yüzden okurun anlayışla karşılaşmasını bekliyoruz.

C dilinde yazılan programın program haline gelme süreci dört ana başlıkta incelenebilir. Bunlar önışleme, derleyici, bağlayıcı ve yükleyici işlemleridir.

Program Yazma Sürecinde, biz metin editörü vasıtasıyla boş bir deftere program kodlarını program bilğimiz vasıtasıyla yazarız. İstenirse bu metin editörü Notepad gibi basit bir program da olabilir. Burada programı klavye ile yazarız ve metin editöründe .c uzantılı olarak kaydederiz.

Önişleme Sürecinde, program yazma sürecinde yaptığımız ön işlemci direktifleri işlenir ve kod metin bazında bu şekilde hazırlanır. Örneğin #define ile yaptığımız bir tanımlama aslında C diline ait bir komut değildir. Bu önişleme komutu olarak tanımladığımız değeri bulup yerine yapıştırmadan öte değildir. Bu durumda ön işlemci bizim yazdığımız program metni üzerinde değişiklikler yapmaktadır. Önişlemci yönergeleri # işareti ile belirtilmektedir. Örneğin #include "lcd.h" dediğimizde bunu önişlemci işlemektedir. Önişlemci yönergelerini biraz ileri seviye bir konu olduğu için ileride daha ayrıntılı olarak sizlere anlatacağız.

Derleme Sürecinde, önişleme sürecinden geçmiş program kodu assembly diline çevrilir. Burada bu kodu assembly diline çevirecek programa derleyici adı verilir. C ile program yazmanın en önemli aşamasını gerçekleştiren program derleyicidir. Derleyici olmadan C dilinde yazdığımız programların bir anlamı olmayacaktır. Örneğin yeni üretilmiş bir işlemci için C programı yazmamız gerektiğinde öncelikle bunun için birinin derleyici yazması gereklidir. C dili herhangi bir program halinde değildir. Kitapta yer alan dilin özelliklerini ve komutlarını donanımın anlayacağı dile çevirecek bir program yapmak elbette en zor işlerden biridir. Üstelik bu derleyici programın donanım için hızlı ve verimli bir kod üretmesi gereklidir. C dili Assembly diline çevrilirken belli kural ve ilkeler dışında her komutun assembly karşılığı kullanılır ve kodlar okundukça bu assembly komutları dosyaya eklenir. Örneğin “+” operatörünü gördüğümüz zaman “ADD” komutunu ekler veya “++” komutunu gördüğümüz zaman “INC” komutunu ekler. Bu görülen operatörlerin makine dilinde bir komut olarak karşılığı bulunmaktadır ve bu her makine için farklıdır. Fakat C dili büyük ölçüde taşınabilir bir dil olduğu için farklı derleyicilerde aynı C kodunu derletip farklı donanımlar için program yazabiliriz.

Bağlama süreci, ise derleme işleminden sonra derlenmiş halde duran kütüphanelerle program arasında bağ kurma sürecidir. Kullandığımız fonksiyonların kodu başka bir yerde yer aldığı anda, başka bir kütüphane kullandığımızda bu parçalar alınır ve bizim programımıza eklenir.

Yükleme süreci, programın çalışabilir olmasından sonra belleğe yüklenme işlemidir.

Bu aşamaların hepsini program yapsa da biz bazen bu üretilen dosyaları inceleme ihtiyacı duyarız. Gömülü sistemler üzerinde çalışıyorsak bu ihtiyaç daha da artmaktadır. Örneğin C dilinde program yazsak da bunun Assembly

haline çevrilmiş versiyonunu görme ihtiyacı duyabiliriz. Çünkü derleyiciye her zaman en iyi kodu üretme konusunda güvenemeyiz.

Derleyici Kurulumu

Öncelikle derleyici ile IDE arasındaki farkı bilmemiz gereklidir. IDE metin editörü, hata ayıklayıcı, programlayıcı gibi araçları içerisinde bulunduran bir geliştirme ortamına verilen isimdir. Visual Studio ve Eclipse IDE olarak en meşhurlarıdır. Derleyiciler ise platforma veya üreticiye göre değişiklik gösterebilir. Örneğin Visual Studio yüklediğinizde Microsoft'un C/C++ derleyicisini kullanırsınız. Eğer Eclipse kullanmak istiyorsanız GCC derleyicilerin Windows sürümlerini kullanırsınız. Biz masaüstünde C programlama için Eclipse IDE ve MinGW derleyicisini kullanacağız.

Şimdi Eclipse'nin nasıl kurulacağından size kısaca bahsedelim.

Eclipse'nin sitesine giriyoruz ve Eclipse C/C++ sürümünü indiriyoruz. Aşağıdaki bağlantıdan bunu indirip kurabilirsiniz.

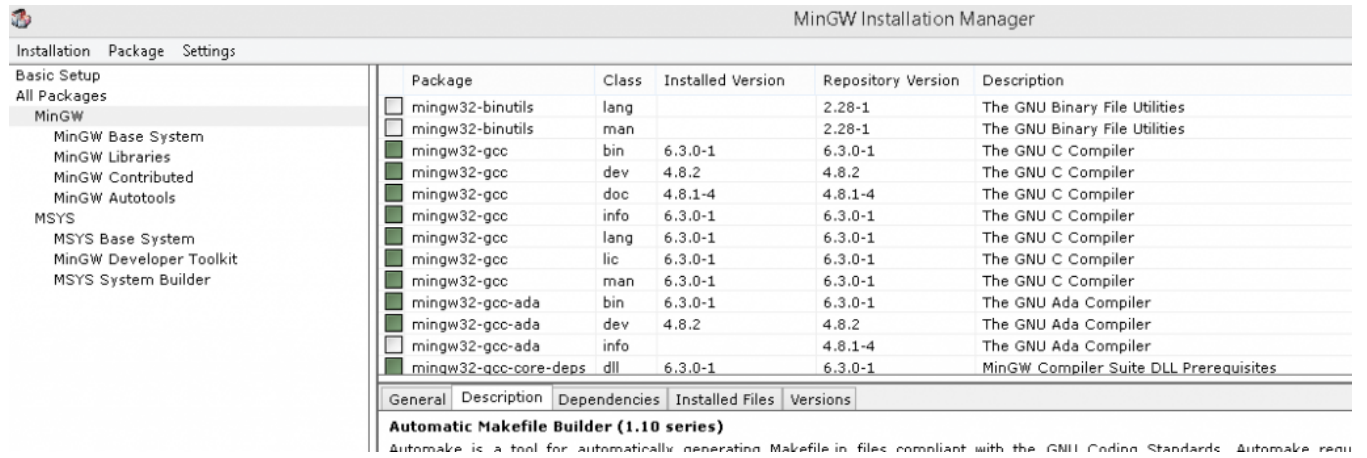
<https://www.eclipse.org/downloads/packages/release/2018-12/r/eclipse-ide-cc-developers>

Biz Eclipse'yi yüklediğimizde açıp hemen kod yazmaya başlayamayız. Eclipse içerisinde derleyici program ile beraber gelmemektedir. O yüzden derleyiciyi ayrıca indirip kurmamız gereklidir.

Windows için GNU derleyici koleksiyonu arasında MinGW ve Cygwin ikilisini görmekteyiz. Biz daha basit bir yapıda olan MinGW'yi indirip kuracağız. Bunun için aşağıdaki bağlantıya tıklıyoruz.

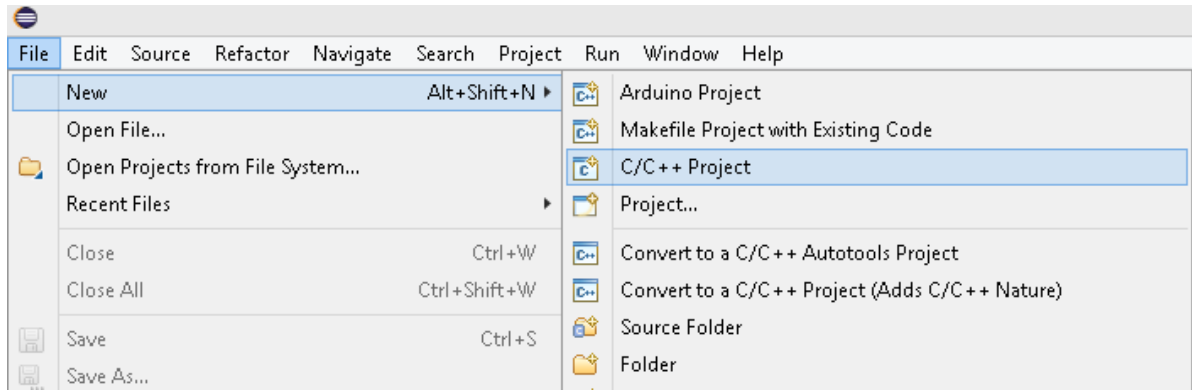
<https://osdn.net/projects/mingw/downloads/68260/mingw-get-setup.exe/>

Burada MinGW kurulum programını yüklemiş oluyoruz. Bir sonraki adım ise açılan MinGW Installation Manager'den C ile ilgili olan derleyicileri seçip kurmaktır.

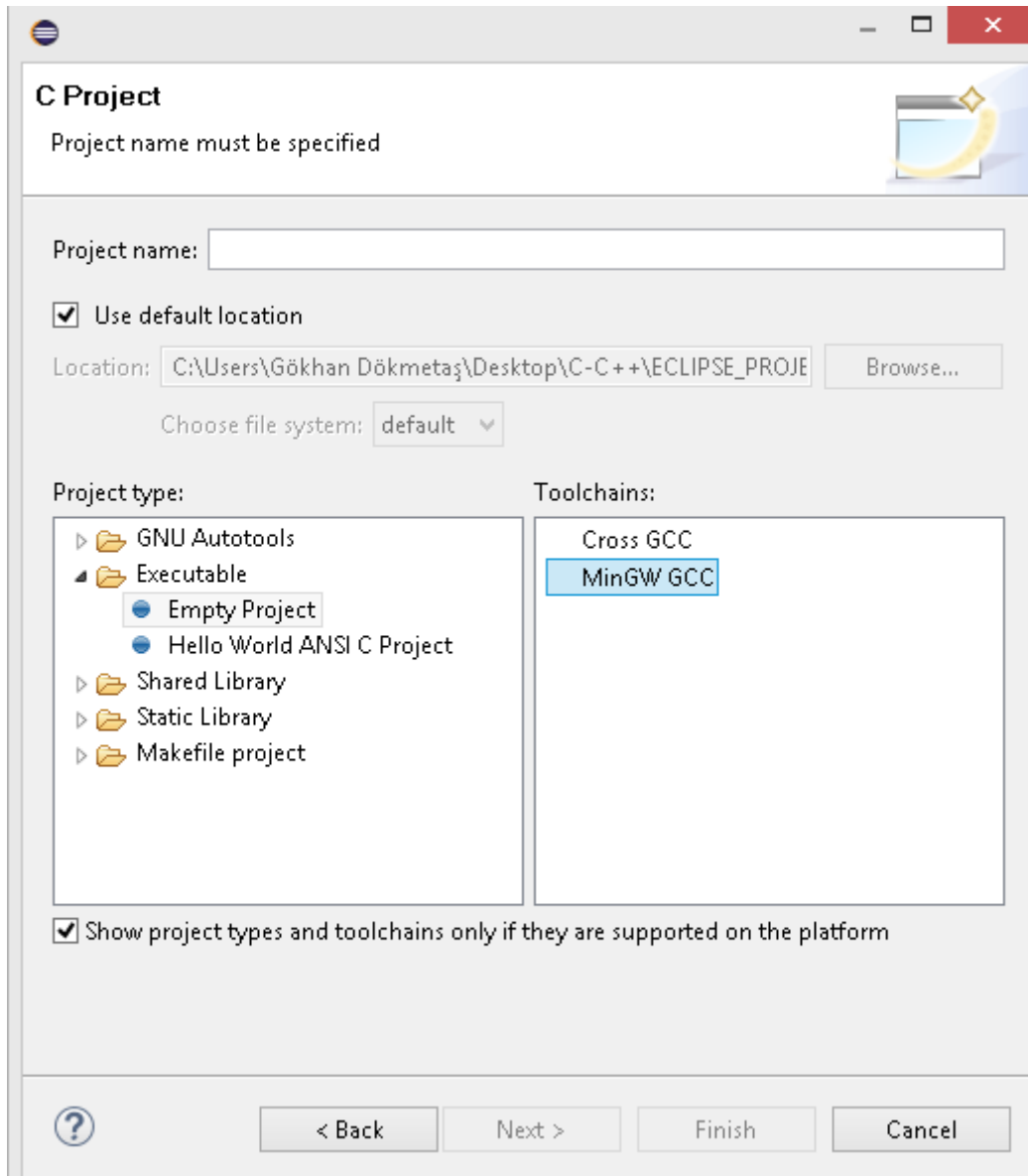


Buradaki C Compiler kısımlarının hepsinin işaretli olduğuna emin olmak lazımdır. Çok uğraşmak istemiyorsanız Visual Studio indirebilirsiniz orada otomatik olarak derleyiciler mevcuttur. Eclipse geliştirme ortamı açık kaynaklı olduğu için pek çok firma Eclipse'yi temel alan geliştirme ortamları hazırlamaktadır. STM32'nin TrueStudio gibi geliştirme ortamlarına hazırlık için Eclipse'ye alışmamız iyi olur. Visual Studio tabanlı Atmel Studio için de Visual Studio'yu öğrenmemiz iyi olur. Biz yaygınlığı daha fazla olan Eclipse'yi kullanma taraftarıyız.

Şimdi Eclipse'de yeni bir proje oluşturmayı anlatalım. Eclipse'yi açtığımızda üst taraftaki menüden File/New/C/C++ Project sekmesini seçiyoruz.



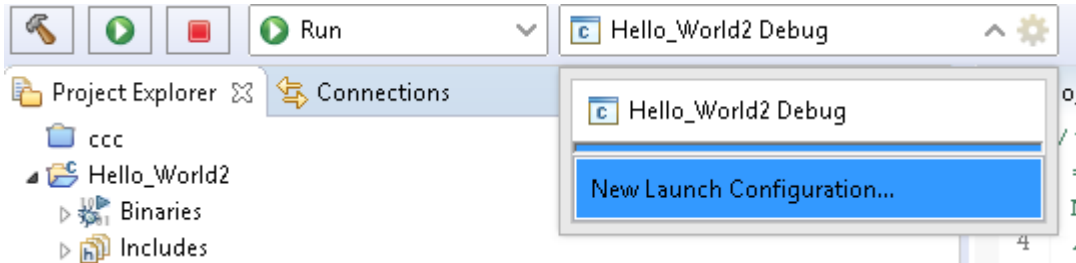
Burada C projesini seçtikten sonra “Toolchain” kısmından MinGW GCC derleyicisini seçiyoruz. MinGW yüklemeden burada bu gözükmeyecektir.



Burada öncelikle bizim için hazır bir proje ortaya çıkması için “Hello World ANSI C Project” sekmesini seçmemiz gerekli. Sonra sağ taraftan MinGW GCC derleyicisini seçiyoruz. Projeyi oluşturduktan sonra hazır bir “Merhaba Dünya” kodu karşımıza geliyor. Bunu çalıştırmak için yukarıda yer alan üç düğmeyi kullanacağız.



Burada yer alan birinci tuş “Build” yani derleme tuşudur. Yazdığımız programı önce derlememiz gerekir. İkinci tuş ise çalıştırma tuşudur. Programda hata ayıklama ve çalıştırma işlemini burada yaparız. Üçüncüsü ise çalıştırma veya hata ayıklamayı durdurma tuşudur. Programı çalıştırmak için öncelikle bir çalıştırma ayarı yapmamız gereklidir. Onun için sağ taraftaki açılır kutuya tıklıyoruz ve “New Launch Configuration...” seçeneğini seçiyoruz.



Burada “Run” seçeneğini seçip dosyayı belirttikten sonra çalıştırılabilir .exe dosyasını program debug klasöründe üretiyor. Projenin hangi klasörde olduğunu ise başta projeyi oluştururken belirliyoruz. Eclipse’nin güzel bir yanı olarak dahili konsol yer almakta. Yani konsolda gördüğümüz çıktıları aşağıda yer alan konsol ekranından görebiliriz. Eclipse’yi beğenmeyenler Visual Studio yükleyip kullanabilir. Hatta çoğu eğitimde olduğu gibi Dev-C++ da kullanabilirsiniz. Bunların çok farkı yoktur. Fakat ben Dev-C++’yı pek

beğenmediğim için daha profesyonel işlere yönelik olan Eclipse'yi tercih ediyorum.

Yazının devamında derleyici veya geliştirme ortamından bağımsız olarak konumuza devam edeceğiz.

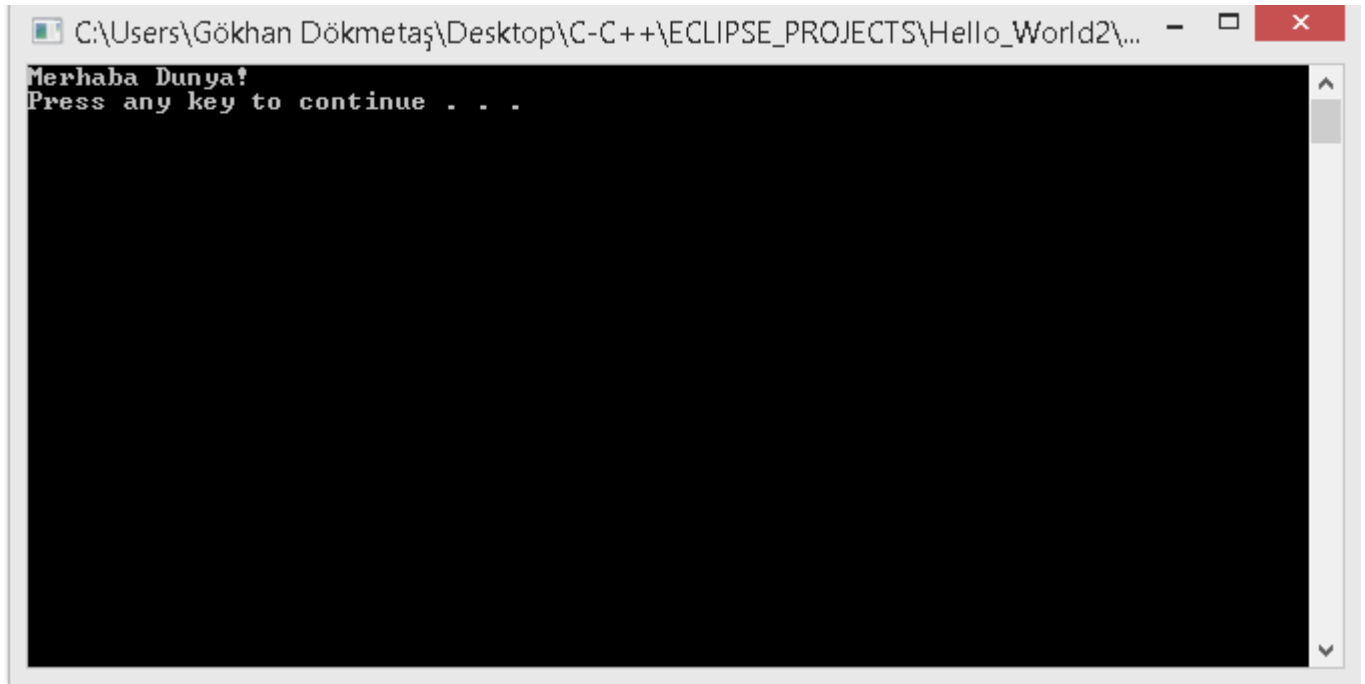
Temel C Programlama -8- İlk Program

C dilinde en büyük adımı ilk programı yazmakla ve ikinci büyük adımı ise ilk programı anlamakla atarız. Bu noktadan sonra anlamadan ilerlememek gereklidir. Eğer anlamadığınız nokta olursa farklı kaynaklardan yararlanarak anlayana kadar o konu üzerinde durmanız gereklidir. Biz ilk program olarak diğer bütün eğitimlerde olduğu gibi “Merhaba Dünya!” programını yazacağız. Bu program basit bir görevi yerine getirirse de pek çok bilgiyi bünyesinde barındırmaktadır. Bu bilgileri öğrendikten sonra programa tekrar bakınca programı anlayabiliriz. Fakat şimdilik programı deneyip nasıl çalıştığını görelim.

```
// C ile Merhaba Dünya Programı
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf( "Merhaba Dünya!\n" );
    system("PAUSE");
} // Main Sonu
```

Bu programı yüklediğimizde konsol ekranında şöyle bir çıkış verecektir



```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...  
Merhaba Dünya!  
Press any key to continue . . .
```

Görüldüğü gibi program “Merhaba Dünya!” yazısını yazdırdıktan sonra devam etmek için bir tuşa basın uyarısını verip kendini beklemeye alıyor. İngilizce işletim sistemi kullandığım için “Press any key to continue . . .” yazsa da Türkçe Windows kullanıyorsanız size “Devam etmek için bir tuşa basın . . .” ifadesini yazdıracaktır. Şimdi bütün program komutlarını tek tek inceleyelim.

Yorumlar

C dilinde programlama yaparken program komutlarından farklı olarak not almak istediğimiz veya açıklamada bulunduğumuz metinleri yorum işaretini (//) kullanarak programa dahil ederiz. Bu işareten sonraki kısmı derleyici program görmezden geldiği için istediğimizi o satıra yazabiliriz. Yorum işaretleri çok satırlı olacaksa farklı bir yol olarak /* ve */ işaretlerini kullanırız. Örneğin çok satırlı bir yorum yazacaksak şu şekilde yorum işaretlerini kullanabiliriz.

```
/*  
    Birinci Satır  
    İkinci Satır  
    Üçüncü Satır  
*/  
  
/* Tek Satırda Yorum Da Yapılabilir */
```

Programda yorumların olması programı daha anlaşılır kılacaktır. Sadece başkalarının okuyacağı program değil kendi yazdığımız programda da yaptığımız işlemi unutmamak ve önemli noktaları gözden kaçırmamak için bu yorumları kullanmamız gereklidir. Bildiğimiz ve her yazdığımız komutu yorumlamak elbette gerekli değildir fakat önemli kısımlarda kullanmamız bizim yararımızdır.

Boş Satır (White Space)

4 numaralı satırda göreceğiniz üzere boş bıraktığımız bir satır programda yer almaktadır. Bu boş satırlar derleyici tarafından görmezden gelinmektedir ve programa bir etkisi olmamaktadır. Bu boşlukları bırakma nedenimiz komutları birbirinden ayırarak daha düzenli bir hale getirmek istememizdir. Siz de program yazarken bu boşlukları programın uygun yerlerine eklerseniz daha kolay okunabilir bir kod meydana getirebilirsiniz.

#include Yönergesi

Önceki yazılarımızda ön işlemci yönergelerinden bahsetmiştik. Bu yönergeler her zaman # işareti ile başlamaktadır. Bunlar programa dahil olmasa da derleyici programa belli bir iş yapması konusunda talimat vermektedir. #include'nin anlamı “dahil et” demektir. Yani bir dosyayı yazdığımız program

dosyasına dahil etmek istiyorsak `#include` yönergesini kullanmamız gereklidir. `#include <stdio.h>` komutu ile Standart giriş/çıkış başlık dosyasını programımıza dahil ettik. `<` ve `>` işaretleri arasına dosya adını yazmamız dosyanın bilinen bir konumda olduğu durumlar için geçerlidir. Eğer kendi yazdığımız veya sonradan eklediğimiz bir kütüphane dosyası varsa bu işi `"` ve `"` işaretleri arasına dosyanın adını yazmakla gerçekleştiririz. C dili herhangi bir kütüphane dosyası eklemediğimiz zaman dilden ibaret olmaktadır. Yani ne fonksiyon ne de farklı bir özellik kullanmaya müsaade etmektedir. O yüzden hangi fonksiyonu kullanacaksak ilgili kütüphanenin dosyalarını buna eklememiz gereklidir. En temel giriş ve çıkış işlemleri için bu geçerlidir. Örneğin BASIC ve Python gibi dillerde bu böyle değildir. PRINT komutu dilin içindedir ve ayrı bir kütüphane eklemeye gerek yoktur. Bu dili kalabalıklaştırmaktadır. C dili ise mümkün olduğu kadar sade yapıdadır ve beraberinde olmazsa olmaz standart kütüphanelerle gelmektedir.

main Fonksiyonu

Kodlardan oluşan program blokuna fonksiyon adı verilir ve C dilinde yazılmış programlar fonksiyonlardan oluşmaktadır. Hiçbir fonksiyon içermeyen basit bir program bile bir fonksiyon içerisinde yer almaktadır. Program başlayınca ilk çalıştırılacak komutların olduğu fonksiyon main fonksiyonu olarak ayarlanmıştır ve program yazarken öncelikle bir main adında fonksiyon oluşturmamız gereklidir. Kod bloku süslü parantez yani `{` ve `}` işaretleri arasında sınırlandırılmıştır. Bunların dışına yazdığımız program çalıştırılmaz ve derleyici hatası ile karşılaşırız. main fonksiyonu içerisine ilk çalıştırılacak komutlar yazılır. Önışlemci direktifleri ve global değişkenler gibi tanımlamalar program dosyasının başına yazılabilir. Şimdi bir fonksiyon yapısının nasıl olduğunu inceleyelim.

```
Fonksiyonun_Döndürdüğü_Değer Fonksiyon_Adı (aldığı_argüman)
{
    Kod Bloku
}
```

Burada main fonksiyonunu `int main(void)` olarak belirttiğimizden bu fonksiyonun `int` (tamsayı) tipinde bir değer döndürdüğünü ve `void` yani boş değer aldığını görmekteyiz. Main fonksiyonunun bu şekilde yazılmasının bir kuralı vardır. İlerleyen konularda fonksiyonların nasıl çalıştığını size ayrıntısıyla anlatacağız.

printf fonksiyonu

`printf("Merhaba Dünya!\n");` komutunun "Merhaba Dünya!" yazısını ekrana yazdırdığını uygulamalı olarak gördünüz. Yalnız bu bir satır kodda incelenmesi gereken pek çok unsur vardır. Öncelikle `printf` komutunun bir fonksiyon olduğunu ve özel bir iş yapmak için fonksiyonları kullanmamız gerektiğini bilelim. Bu fonksiyonları tanımanın yolu ise peşinden gelen parantezlerdir. (ve) işaretleri tanımadığımız bu komutun özel bir fonksiyon olduğunu belirtmektedir. `printf` fonksiyonu `#include <stdio.h>` önışlemci yönergesi ile programa dahil ettiğimiz `stdio.h` dosyasının içinde bulunmaktadır. Bu dosya derleyiciye göre değişse de belli standartlara sahiptir. Bu dosyanın içerisinde ne olduğunu şimdi incelemeye gerek olmasa da bizim yazdığımız `printf(...)` komutu dosyanın içinde yazılı olan `printf` fonksiyonunu çalıştırmaktadır. `Printf` komutunun görevi ise konsola veya terminale formatlanmış yazı yazdırmaktır. `Print`, yazdır anlamı taşırken `printf` "Print formatted" yani formatlanmış halde yazdır demektir. Bir fonksiyonu çağırırken şu şekilde bir komut kullanırız.


```
fonksiyon(deger);
```

Eğer argüman (değer) almayan fonksiyon olursa parantezin içi boş bırakılır fakat her zaman önce fonksiyon adı ve sonrasında iki parantez “()” kullanılması gereklidir. Burada büyük ve küçük harf ayırımına dikkat etmemiz gerekir.

Noktalı Virgöl

C dilinde her komut (statement) bittiği zaman noktalı virgöl “;” konur. C dili BASIC ya da Python dilleri gibi satır satır ilerleyen bir dil değildir. Derleyici satıra değil noktalı virgüle bakarak komutun bitip bitmediğini anlar. Bu bize program yazarken biraz esneklik tanımaktadır. Örneğin “Merhaba Dünya!” yazısını istersek tek satırda değil de şöyle yazdırabilirdik.

```
printf(  
    "Merhaba"  
    " Dünya!\n"  
);
```

Program yine aynı şekilde çalışsa da okunabilirliği düşecekti. O yüzden satırları iyi kullanmayı bilmek gereklidir. Noktalı virgülü koymayı unutmak en sık karşılaşılan programlama hatalarından biridir ve yıllar boyu programlama yapanların bile dalgınlığında yaşadığı bir durumdur.

Çift Tırnak

C dilinde karakter dizilerini belirtmek için çift tırnak işareti kullanılır. Çift tırnak arasına yazdığımız metnin sayı olmadığı ve karakter dizisi olduğunu fonksiyona belirtmek için “Merhaba Dünya!” yazmaktayız. Fonksiyona hangi değer tipini gönderdiğimiz ise çok önemlidir. Her fonksiyona her değer tipini gönderme imkanımız yoktur. printf fonksiyonu ilk değeri her zaman karakter dizisi biçiminde aldığı için bu fonksiyona göndereceğimiz değer ” ve ” işaretleri arasına yazılmalıdır.

Kaçış Sekansı

printf fonksiyonuna iki tırnak işareti arasında yazdığımız Merhaba Dünya! yazısı ekranda gösterilecektir. Fakat biz bir komut göndermek istediğimizde örneğin yeni satıra geçmesini istediğimizde metin gibi olan fakat sadece o amaç için kullanılan bir yapıyı göndermemiz gereklidir. Buna kaçış sekansı adı verilir. Örneğin printf(“Merhaba Dünya!\n”); derken \n komutu ekranda yazdırılmamaktadır. Bu fonksiyonun yapısından dolayı böyle gerçekleşmektedir. Fonksiyon önce \ işaretini metnin içerisinde arar ve bulduktan sonra o işaretin ardından “n” ile ifade edilen komutu yerine getirir. \n komutu burada yeni satıra geç anlamına gelmektedir. Kaçış sekanslarını ilerleyen konularda uygulamalı olarak göstereceğiz.

system(“PAUSE”);

Program ilk olarak printf fonksiyonunu çalıştırarak “Merhaba Dünya!” yazısını ekrana yazdırır. Çünkü main fonksiyonu içerisinde ilk yer alan fonksiyon oydu. Yalnız Visual Studio haricinde bir derleyicide (Eclipse, Dev-C++ gibi) program çalışır çalışmaz kendini sonlandırmaktadır. Bunun için sistemi beklemeye sokacak bir fonksiyona ihtiyaç vardır. Bu da stdlib.h yani standart kütüphane fonksiyonları içerisinde yer alan system fonksiyonudur. Bu fonksiyon içerisine

aldığı argüman ile (gönderilecek değer) programı beklemeye sokar ve “Devam Etmek İçin Bir Tuşa Basın . . .” yazısını ekrana yazdırır. Biz bir tuşa bastığımızda program kapanacaktır. Biz ekrandaki yazıyı okumak için bu fonksiyonu kullanıyoruz. Daha farklı fonksiyonları da kullanma imkanımız vardır ve bunu ileride fırsatımız olursa anlatacağız.

İlk programı bütün ayrıntısıyla sizlere anlattık. İlk programdan sonra veri okuma ve işleme konusu üzerinden devam edeceğiz. Verdiğimiz ilk program örneklerini anlamanız geri kalanı kolayca anlamanızı sağlayacaktır.

-9- Basit Program Örnekleri – 1

Temel C programlamada ilk örneği anlayarak C diline büyük bir adım attık. Şimdi ise basit örnekleri büyük bir ayrıntı ile açıklayarak C program yapısını anlamanız için çaba sarf edeceğiz. Bunun için üç basit program yazdık ve programları önce geliştirme ortamına yazıp deneyeceğiz ve nasıl çalıştığını gördükten sonra yazdığımız kodu inceleyeceğiz.

Sayı Girme ve Girdiği Sayıyı Gösterme

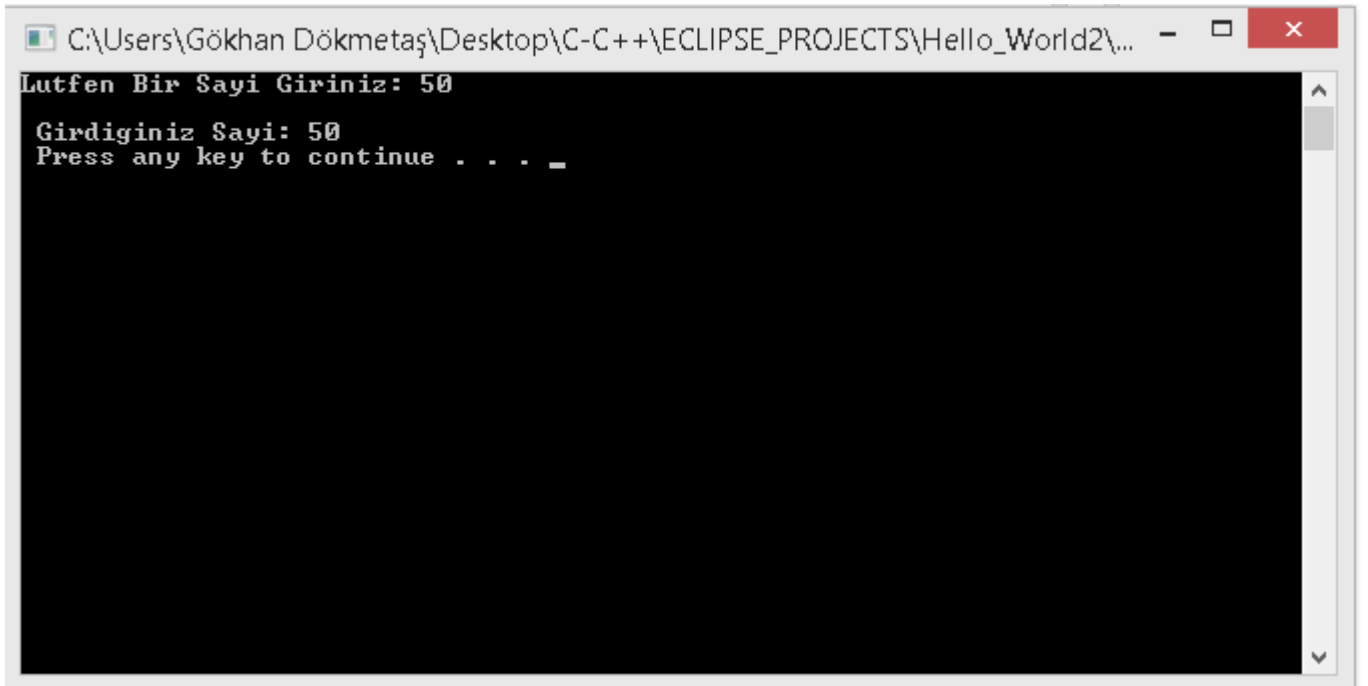
Önceki “Merhaba Dünya!” programında ekrana yazı yazdırmıştık. Peki, bizim ekrana yazdığımız yazıyı programın okuması için ne yapacağız?, sorusuna karşılık bu örneği yapıyoruz. Böylelikle C dilindeki temel giriş ve çıkış fonksiyonlarının ikisini de öğrenmiş olacağız.

```
// Sayı girme ve girdiği sayıyı görme

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

int main( void )
{
    int sayi = 0;
    printf( "Lutfen Bir Sayi Giriniz:" );
    scanf ("%i", &sayi);
    printf ( "\n Girdiginiz Sayi: %i \n ", sayi);
    system("PAUSE");
} // Main Sonu
```

Burada anlattığımız yerleri tekrar anlatmamaya çalışıyoruz fakat bazı kısımları yeri geldiğinde size hatırlatacağız. Bir de daha önceden yorum satırlarını size anlatsak da her kodu açıklamak için diğer eğitimlerde olduğu gibi yorum satırlarını bol bol kullanmayacağız. Çünkü öğrenci ilk öğrendiği sırada kodu okuyup anlamaya değil yorum satırlarını ezberlemeye çalışıyor. Bu kadar basit bir yerde yorum satırları ile her komutu tek tek açıklamak öğrencinin kafasını çalıştırmadan ezberlemesine sebep olur ve örneklerin öğretici niteliği azalır. O yüzden ben her zaman yaptığım gibi kod açıklamalarını kod üzerinde değil metinde yapacağım. Programı çalıştırdığımızda şöyle bir çıkış verecektir.

A screenshot of a Windows console window titled "C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...". The console has a black background with white text. It displays the following text: "Lutfen Bir Sayi Giriniz: 50", "Girdiginiz Sayi: 50", and "Press any key to continue . . . _". The window has standard Windows controls (minimize, maximize, close) in the top right corner.

int ile değişken tanımlama

Değişkenler içinde veri tutan bellek adreslerine verilen addır. Biz bir değişken tanımladığımızda derleyici bellekteki bilmediğimiz bir adresi sahiplenir ve burada bizim tanımladığımız değişkenin değerini tutar. Örnekte olduğu gibi `int sayi = 0;` dediğimizde `int` yani tam sayı tipinde ve `sayi` adında ve değeri 0 olan değişkeni belleğin örneğin `0x0000ACF5` adresinde tanımlar. Biz bu değişkene bir değer yüklemek istediğimizde bu bellekteki adrese değer yüklenir. Bu

bellek adresinin değeri normalde bizi alakadar etmez. Bellek adreslerine işaretçiler kısmında daha ayrıntılı şekilde bakacağız. Şimdilik bir değişken tanımladığımızda belli bir bellek adresine ad verdiğimizizi ve ad verdiğimiz bellek hücresinde de verinin tutulduğunu bilelim.

Bir değişken tanımlamak için anahtar kelime olan değişkenin tipini yazmakla işe başlarız. Bu değişkenin içine yazacağımız değer ondalıklı sayı mı olacak tam sayı mı olacak yoksa harf mi olacak burada belirleriz. Değişken adını yazdıktan sonra değişkeni ne maksatla kullanacaksak onunla alakalı bir ad veririz. Örneğin sonuç değerini içinde bulunduracaksa sonuc olabilir, isim bulunduracaksa isim olabilir. Bu adı kendimiz belli kurallar çerçevesinde veririz. Değişken isimlerini güzel seçmek önemlidir. İyi seçmediğimiz takdirde programı anlamak güçleşecek ve kafamız karışacaktır.

Sonrasında ise değişkene “=” operatörü ile ilk değer ataması yapılır. Tam sayı tipinde bir sayı değeri vermek istediğimiz zaman klavyeden sayıyı doğrudan yazabiliriz. Farklı değerler belli işaretler, önek ve soneklerle belirtilir. Örneğin değişkenin değeri bir karakter olacaksa ‘a’ şeklinde ondalıklı değer olacaksa 3.14F şeklinde olabilir. Bu değer tiplerini ilerleyen zamanda daha ayrıntılı olarak açıklayacağız. Şimdi örnek değişken tanımlamalarına bir bakalım.

```
int ogrenci_numarasi;  
int ogrenci_notu = 0;  
char harf_notu;  
unsigned int _IsaretsizSayi = 50000;  
char oyun_adi[20] = "Fallout";
```

Burada önce `ogrenci_numarasi` adında bir değeri belirsiz bir değişken tanımlanmıştır. Bunun ilk değerini vermediğimiz için ilk değer olarak rastgele bir değer alacaktır. Bellekte belirsiz bir alanda herhangi bir değer olabilir. Bu belirsiz alanı belirli yapmak için sadece elde tutmak değil içeriğini de değiştirmek gereklidir. O yüzden değişkene ilk değer atamak her zaman güvenli bir yoldur. `int ogrenci_notu = 0;` ile `ogrenci_notu` değişkeni tanımlanmasının yanı sıra buna ilk değer olan 0 değeri atanır. Geri kalan tanımlamaları görmemiz için yazsak da bunları değişken tiplerini anlattığımız yerde ayrıntısıyla işleyeceğiz.

scanf Fonksiyonu

C dilinde `stdio.h` kütüphane dosyası temel giriş ve çıkış işlemlerini yapmamızı sağlayan fonksiyonların bulunduğu kütüphane dosyasıdır. Bu kütüphane dosyası **ST**andar**D** Input **O**utput kelimelerinin harflerinden türetilmiştir. Bu kütüphanede en sık kullandığımız iki fonksiyon vardır. Bunlardan biri konsol ekranına veri yazdırmak için kullandığımız `printf` fonksiyonu, diğeri ise klavyeden yazılan veriyi okuduğumuz `scanf` fonksiyonudur. `printf` fonksiyonu nasıl “print formatted” deyiminin kısaltmasıysa `scanf` fonksiyonu da “scan formatted” yani formatlı bir şekilde tarama anlamına gelen bir fonksiyondur. Bu `scanf` fonksiyonu yine `printf` fonksiyonuna benzer bir şekilde çalışmakta fakat okuma işlemini yapmaktadır. Bizim klavyeden yazdığımız değerler konsol ekranında görünse de `scanf` fonksiyonu Enter tuşuna basana kadar çalışmamaktadır. Sonra ise ilk aldığı argümanı ikinci argümanın adresine aktarmaktadır. Şimdi yazdığımız komutu ayrıntısıyla inceleyelim.

`scanf (“%i”, &sayi);`

Bu başta karışık gibi görünse de oldukça kısa ve anlaşılır bir yapıdadır. Sadece fonksiyonun nasıl çalıştığını bilmemiz bunu anlamak için yeterlidir.

Burada scanf fonksiyonunun ilk aldığı değer “%i” adında bir değerdir. Burada iki tırnak arasındaki değer karakter dizisi olduğunu biliyoruz. Bu fonksiyon %i ile ikinci argümana erişmekte ve % işaretinden sonra girdiğimiz format değerini yani “i” değerini okuyarak girilen değeri formatlı bir şekilde ikinci argümana yazdırmaktadır. Bu printf fonksiyonlarının nasıl format aldığını ilerleyen konularda size açıklayacağız. Fonksiyonun aldıkları değerler arasında virgül bulunması gereklidir. Örneğin fonksiyon iki veya üç değer alıyorsa şu şekilde yazılmalıdır.

```
fonksiyon_adi(deger1, deger2);  
fonksiyon_adi(deger1, deger2, deger3);
```

Biz yukarıda `int sayi = 0;` diye bir sayı adında tam sayı değişkeni tanımladık. Fakat burada sayı değişkenini `&sayi` şeklinde kullanıyoruz. Bunun sebebi C dilinde `int`, `float`, `double` gibi tek değeri içeren değişkenlerin kendisini yazdığımızda değişkenin değeri verilmektedir. Biz ise değişkeni alarak değişkenin adresine bir veri yazmak istiyoruz. Bu durumda değişkene veri yazacağımız için değişkenin değeri ile bir işlem olmuyor. O yüzden değişkenin adres değerini `&` operatörü ile öğreniyoruz ve fonksiyona değer olarak aktarıyoruz. Fonksiyon ise bu adres değerine okunan veriyi yazıyor. Sonraki örneklerde göreceğimiz üzere her değişken için bu adres operatörünü kullanmak gerekli değildir. Çünkü dizi gibi değişkenlerde ilk değer dizinin adres değeridir.

printf (“\n Girdiginiz Sayi: %i \n “, sayi);

Burada programa girdiğimiz sayıyı göstermek için yine çıkış fonksiyonu olan `printf` fonksiyonunu kullanıyoruz. Öncelikle `\n` kaçış sekansı ile yeni satıra geçeriz ve Girdiginiz Sayi: diyerek kullanıcıya ekranda yazılacak değerin ne olduğunu belirtiriz. Burada `%i` olarak formatlanmış değer fonksiyona ilk

argümanın tam sayı değerinde formatlanarak yazdırılacağını belirtir. Değeri yazdırdıktan sonra ise \n fonksiyonu ile yeni bir satıra geçeriz. Burada alınan argüman sayı argümanıdır. Fark ederseniz burada adres operatörü ile değişkenin adres değerini değil kendisini gönderdik. Çünkü değişkenin değeri ile işlemiz var.

Burada ilk programı ayrıntılı bir şekilde inceledik ve sonraki başlıkta ise yine basit iki programı daha inceleyeceğiz. Bu üç programı anlamak sizin için çok önemli olacak. Biz elimizden geldiği kadarıyla anlatmaya çalışıyoruz.

-10- Basit Program Örnekleri – 2

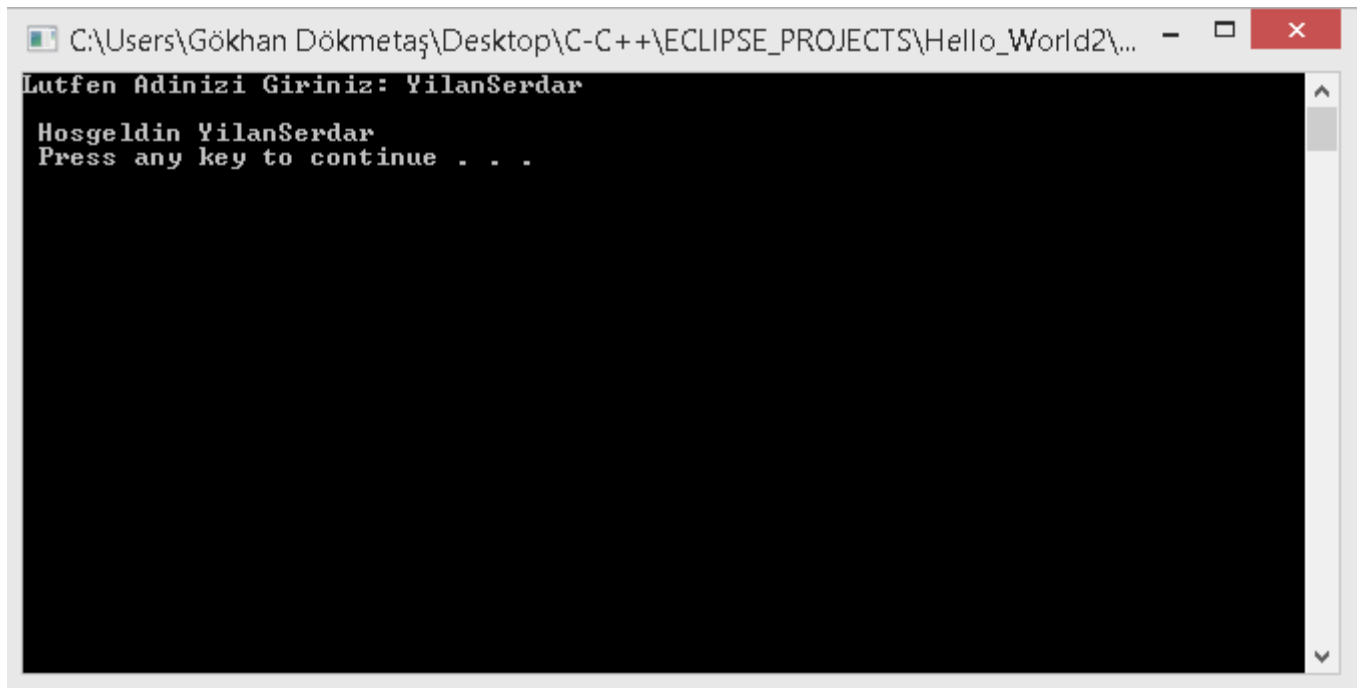
Bu başlıkta basit program örnekleri arasından iki örneğe yer vereceğiz ve sonrasında basit programları incelediğimiz konuyu bitireceğiz ve temel bilgilere devam edeceğiz. Bu basit örnekleri alıştırmalar olarak düşünebilirsiniz. Biz genel olarak konuları örnekler içerisinde değil kategoriler halinde vereceğiz ve aynı zamanda bu sizin için bir referans olacak.

İsim Girme ve Bunu Ekranda Gösterme

Bu program önceki örnekte olduğu gibi klavyeden bir değer alır ve bunu ekranda yazdırır. Fakat önceden sayı girsek de bu sefer bir isim girmek istiyoruz ve isim değerini int ile tanımladığımız tam sayı değişkenine aktarmamız mümkün değil. O yüzden isim için bir “karakter dizisi” tanımlarız ve buna değer aktarırız. Önce programı yazalım ve denedikten sonra nasıl çalıştığını inceleyelim.

```
// Adınızı Girme
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    char karakter_dizisi[20] = "";
    printf( "Lutfen Adinizi Giriniz:" );
    scanf ("%s", karakter_dizisi);
    printf ( "\n Hosgeldin %s \n ", karakter_dizisi);
    system("PAUSE");
} // Main Sonu
```

Program çalıştıktan sonra ekran görüntüsü şu şekilde olabilir.



```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...  
Lutfen Adinizi Giriniz: YilanSerdar  
Hosgeldin YilanSerdar  
Press any key to continue . . .
```

Bizim “YilanSerdar” değerini aktardığımız değişken bir karakter dizisi olmak zorundadır. Çünkü karakter değişkeni tek bir harf değeri içerisinde bulundurabilir. O halde sıra halinde bu karakter değişkenlerinden belli bir miktar lazımdır. Bunun için şu komutu kullandık.

char karakter_dizisi[20] = “”;

Burada karakter_dizisi adında ve 20 karakter uzunluğunda bir karakter dizisi tanımladık. Bunu tanımladığımız dizinin değerleri “YilanSerdar” değerini harf harf sırasıyla içerisine aktaracaktır. Dizileri anlattığımız kısımda daha ayrıntılı olarak bunu sizlere açıklayacağız.

scanf (“%s”, karakter_dizisi);

Burada ilginç nokta önceden scanf’de bir değer okuyup bunu da bir değişkene aktarmak istediğimizde değişkenin değerini değil adresini adres gösterme operatörüyle (&) aktarmamız gerekiyordu. Burada ise değer doğrudan aktarılmaktadır. Bu C dilinin bir özelliği olup karakter_dizisi değerinin dizinin ilk

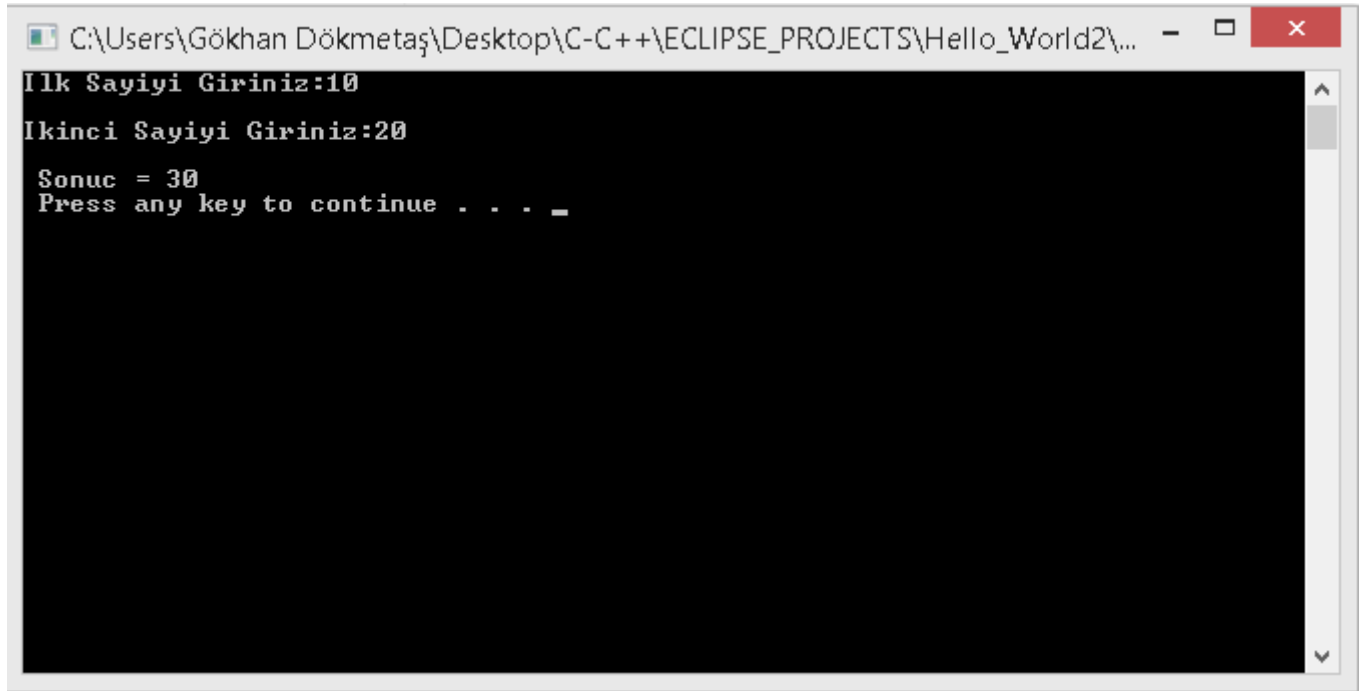
elemanını gösteren adres değeri olmasından kaynaklanır. Erişim operatörü yani [] arasına yazdığımız sayılarla bu dizinin elemanlarına erişiriz fakat dizinin kendisini yazdığımızda ilk elemanın adresine erişiriz. Bu durumda scanf için adres operatörü kullanmaya gerek kalmayacaktır. scanf fonksiyonunda format bölümüne %s yazıyoruz. Çünkü karakter dizisinde formatlanacağını belirtiyoruz.

İki Sayıyı Toplama ve Ekranda Yazdırma

Bu program programlama işlemini biraz daha gerçek anlamıyla yaptığımız programlardan biridir. Çünkü programlama sadece bir yerden veri okuma ve bir yere veri yazmaktan ibaret değildir. Veriyi işlemek de gereklidir. Veri işlemekten de programcılığın pek bir anlamı kalmaz. O yüzden en temel veri işleme yollarından biri olan toplama işlemini sizlere göstereceğiz.

```
// İki Sayıyı Toplama
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int sayi1, sayi2, sonuc;
    printf( "Ilk Sayiyi Giriniz:" );
    scanf ("%i", &sayi1);
    printf( "\nIkinci Sayiyi Giriniz:" );
    scanf ("%i", &sayi2);
    sonuc = sayi1 + sayi2;
    printf ( "\n Sonuc = %i \n ", sonuc);
    system("PAUSE");
} // Main Sonu
```

Programın muhtemel çıktısı şu şekilde olabilir.



```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...  
Ilk Sayiyi Giriniz:10  
Ikinci Sayiyi Giriniz:20  
Sonuc = 30  
Press any key to continue . . . _
```

Bu programı incelediğimizde önceki örneklerde gördüğümüzden çok farklı bir komut olmadığını görüyoruz. Şu iki noktada farklılığı görmemiz mümkün. Öncelikle değişken tanımlarken araya virgül koyarak değişken tanımladık. `int sayi1, sayi2, sonuc;` komutunda olduğu gibi birden fazla değişken tanımlamak için kolay yol aralarına virgül koymaktır. İkinci komutumuz ise `sonuc = sayi1 + sayi2;` komutudur. `scanf` ile klavyeden girdiğimiz değerleri `sayi1` ve `sayi2` değişkenlerine atadık ve bu iki değişkenin değerlerini toplayarak üçüncü değişkene sonucu atıp bunu da ekranda yazdıracağız. Öncelikle C dilinde aritmetik işlemlerin operatörlerle yapıldığını söyleyelim. Yani Assembly dilinde olduğu gibi her işlem için bir komut yoktur onun yerine matematikte kullandığımız operatörler kullanılır. Örneğin Assembly’de `ADD` komutu yerine burada `+` operatörü görev yapmaktadır. `sayi1 + sayi2` dediğimizde ise iki değişkenin değeri toplanır ve ortada bir sonuç kalır. Bu ortada kalan değeri bir yere atamak lazımdır. Bunu da atama operatörü olan `=` ile yaparız. Atama operatörü her zaman sağdan sola doğru işlemektedir. Burada `sayi1` ve `sayi2`

değerleri işlem sonrasında herhangi bir değişikliğe uğramaz. Değişikliğe uğrayan atama operatörünün solundaki sonuc değişkenidir. Toplama işleminin en sonunda elde edilen değer sonuc değişkenine aktarılarak komut işletilmiş olur.

Bu üç örneğin ardından C programlarının temel yapısını öğrendiniz ve artık ileride anlatacağımız konulara hazırlanmış oldunuz. Benim için de en zor kısmı başlangıç kısmıydı. Artık devam eden konularda daha rahat ilerleyeceğiz.

Temel C Programlama -11- Aritmetik Operatörler

Daha önceki örnekte basit bir toplama işlemini yapmıştık. Biz önceki yazılarımızda mikroişlemci mimarisinden, ALU'dan ve Assembly komutlarından bahsetmiştik. Bir mikroişlemcinin yaptığı matematik işlemleri dört işlemden öte gitmiyordu. Bilimsel hesap makinesinde bir tuşla yaptığımız karmaşık işlemler bu komutların bir araya gelip sırayla kullanılmasından ibaretti. Biz burada en temel işlemler olan dört işlem ve buna bağlı işlemlerle alakalı operatörleri anlatacağız. Karmaşık matematik işlemlerini bu komutların birlikte kullanılmasıyla yapıyoruz. C dilinde beş temel aritmetik operatör ve bunun yanında iki yan operatör bulunmaktadır. Şimdi bu operatörleri bir tablo halinde verelim ve size açıklayalım.

Operatör	Açıklama
+	Toplama
-	Çıkarma
*	Çarpma
/	Bölme
%	Mod (Kalan)
++	Artırma
--	Azaltma

Biz matematik işlemlerini bu işlemlere karşılık gelen operatörlerin sembollerini kullanmakla yaparız. Örneğin çarpma işleminde “x” işaretini kullanmak yerine “*” işaretini kullanmamız gereklidir. Bilgisayar programcılığında programlamanın en temel noktası değerler ve operatörler üzerinde yapılan

işlemdir. Bütün bu döngü, karar ve fonksiyon yapılarını bir kenara bıraktığımızda ana program yapısının değişken değerleri ve operatörler ile yapılan işlemlerden meydana geldiğini görebiliriz. Aynı işlemcinin ana fonksiyonları veya ALU fonksiyonları gibi programın temeli de bu operatörler ile yapılan işlemlerdir.

Şimdi tüm bu operatörleri kullandığımız bir programı çalıştıralım ve bunun üzerinden konumuza devam edelim.

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

    int a = 50;
    int b = 10;
    int c ;

    c = a + b;
    printf("C'nin Degeri : %i\n", c );

    c = a - b;
    printf("C'nin Degeri : %i\n", c );

    c = a * b;
    printf("C'nin Degeri : %i\n", c );

    c = a / b;
    printf("C'nin Degeri : %i\n", c );
```



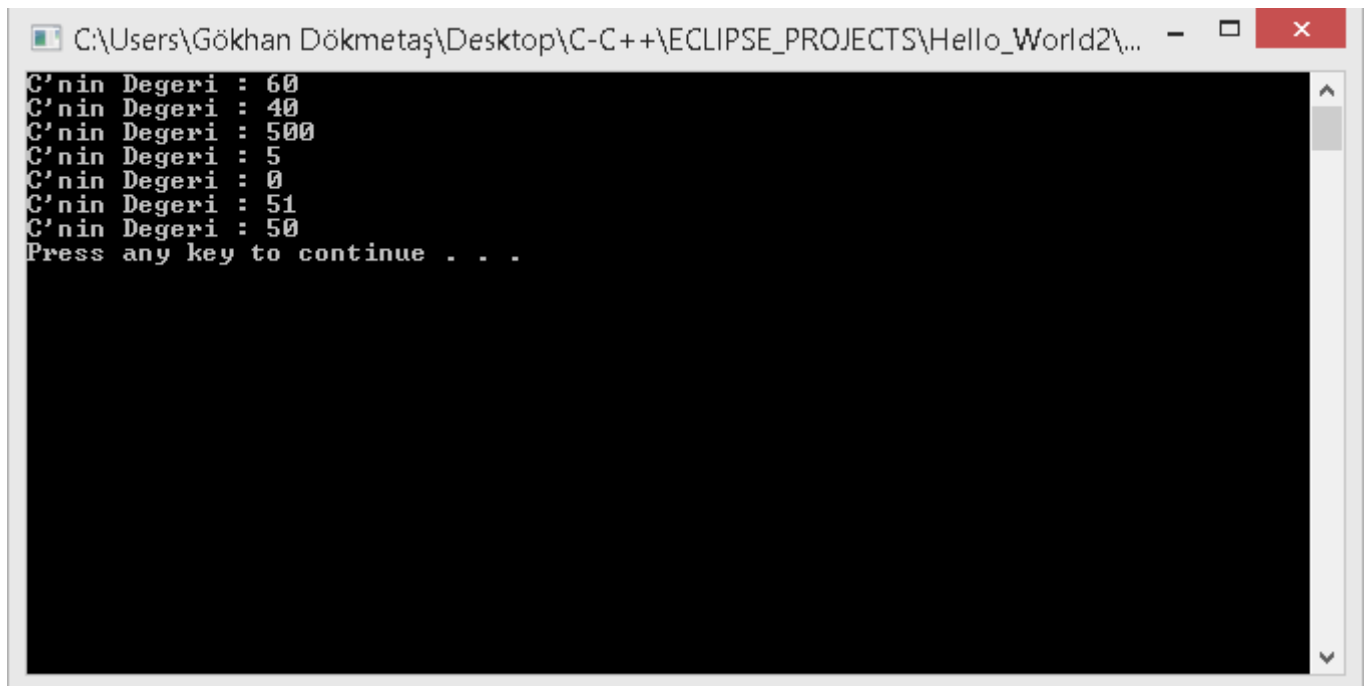
```
c = a % b;
printf("C'nin Degeri : %i\n", c );

c = ++a; // ++ önde bulunmalı işlem önceliği için.
printf("C'nin Degeri : %i\n", c );

c = --a;
printf("C'nin Degeri : %i\n", c );

system("PAUSE");
} // Main Sonu
```

Programı çalıştırdığımızda çıktısı şu şekilde olacaktır.



```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...
C'nin Degeri : 60
C'nin Degeri : 40
C'nin Degeri : 500
C'nin Degeri : 5
C'nin Degeri : 0
C'nin Degeri : 51
C'nin Degeri : 50
Press any key to continue ...
```

Öncelikle değişken olarak tanımladığımız tam sayı değerlere bir bakalım. a değişkeni 50 değerini, b değişkeni de 10 değerini içermekte. c değişkeni ise bir değer içermeyip bu işlem sonucunu barındıran sonuç değişkenidir. **c = a + b;** komutuyla a + b yani a ile b'nin toplamı c değerine aktarılır. Daha önceden a ve b değerinin işlem sonucunda değişmeyeceğini söylemiştik. Burada da c = 50 + 10 yani 60 sonucu c değişkenine aktarılır ve printf fonksiyonu ile c değişkeninin değeri yazdırılır.

Bir sonraki işlem ise **c = a – b;** yani çıkarma işlemidir. a'dan b çıkar ve c değişkenine aktarılır “=” operatörünün atama operatörü olduğunu unutmamanız gereklidir. c'nin önceki değerinin bir önemi yoktur ve elde edilen yeni değer c'nin değeri olur. Önceki değer silindiği için c'nin atama yapılmadan önceki değerinin 0 olduğunu düşünebiliriz. Bu durumda a değişkeni 50, b değişkeni de 10 olduğuna göre sonuç 50-10=40 olacaktır. c değişkenine 40 değeri aktarıldıktan sonra printf fonksiyonu ile ekrana yazdırılır.

Bir sonraki işlem ise **c = a * b;** şeklinde olan çarpma işlemidir. Bu noktaya kadar a ve b değişkenlerinin değerinin sabit kaldığına dikkat ediniz. Ne zaman atama operatörünün solunda olurlarsa o zaman değerleri değişir o yüzden şimdi 50 x 10 = 500 değerini elde etmemiz gereklidir. Bu işlem “*” operatörü ile yapıldıktan sonra c değişkenine aktarılmakta ve bir sonraki printf komutunda c değeri yazdırılmaktadır.

Dört işlemin son işlemi olarak **c = a / b;** komutunda bölme işlemini yaptık. a / b işlemi ile a değişkeni b değişkenine bölünür. a 50 değerinde b de 10 değerinde olduğu için 50 / 10 işlemini yaparız ve sonuç 5 olur. Ekranda da bunun sonucunu görmekteyiz.

Bundan sonra dört işlemi bitirsek de bu işlemlerde sıkça kullandığımız işlemleri yapan operatörler de C diline eklenmiştir. Aslına baktığımızda bilgisayar sistemlerinin ilkel hallerinde çarpma veya bölme işlemi yapan makine komutunun bile olmadığını görürüz. Çarpma işlemi toplama işleminin kısaltması, bölme işlemi de çıkarma işleminin kısaltması olduğundan bu işlemleri ilkel bilgisayarlar algoritma ile yapıyordu. Günümüzde ise “bir artırma” işlemi için bile ayrı işlemci komutu olduğunu görmekteyiz. Mod işlemi bize bölme işleminden kalan değeri verir ve % işareti ile gösterilir. Bölme işleminden kalan değeri elde etmek çarpan bulma işlemlerinde ve pek çok alanda işimize yaramaktadır. $c = a \% b$; komutunda ise a’nın b’ye bölümünden kalan c değerine aktarılmaktadır. $50 \% 10 = 0$ olduğuna göre 0 değeri ekrana yazdırılacaktır.

Görüldüğü gibi programcılıkta derin matematik bilgisi gerekmemektedir. Aslında matematik işlemlerini yaptığımız operatörler bu kadarla sınırlıdır. Bu operatörleri beraber kullanarak karmaşık matematik işlemlerini gerçekleştiririz. Bu durumda formül ezberlemek veya formül çözmeye değil matematik yeteneğine ihtiyaç vardır. Bu yüzden programcılıkta matematiği lisede formül ezberleyip matematik sorusu çözmeye benzetmeyiniz. Matematik uygulamada olduğu zaman sizin sürekli çözümler getirmeniz ve formül uydurmanız gerekecektir. Bunu isteseniz en basit yoldan isterseniz de en karışık yoldan yapabilirsiniz.

Son iki işlem ise bir artırma ve bir azaltma işlemidir. ++ operatörüne tabi tutulan bir değişkenin değeri bir artırılır. $c = ++a$; dediğimizde önce a değişkeni bir artırılır sonra c değerine aktarılır. Eğer $c = a++$; deseydik önce a değişkeni c’ye aktarılacak sonra a değişkeni bir artırılacaktı. İşlem önceliğinin ne kadar önemli olduğunu görmemiz açısından ilk örnek karşımıza çıkmaktadır. ++ operatörü aslında şu komut ile aynı işlemi yapmaktadır.

```
a = a + 1;
```

Fakat hem kullanım kolaylığı hem de optimizasyon açısından ++ operatörünü kullanmamız daha iyi olacaktır. Daha önce bahsettiğimiz gibi mikroişlemcilerde bir artırmanın ve bir azaltmanın karşılığı olan bir komut vardır. Bu komut ise derleyicide ++ operatörüyle ilişkilendirilir. $a = a + 1$ dediğimizde derleyici optimizasyonu çok iyi değilse derleyici bunu uzun yoldan yapacak ve buna göre Assembly kodu üretecektir. Gömülü sistemlerde bizim performanslı program yazmamız şart olduğu için böyle inceliklere dikkat etmemiz lazımdır.

En son komutumuz ise $c = -a$; komutudur. Burada bir artırılmış a değeri tekrar bir azaltılır. Önceden 51 çıktısı elde ettiysek burada 50 çıktısı elde ederiz. Dikkat ederseniz bu operatör doğrudan a değişkenine etki etmektedir. Unary yani tek operandı olan (tekli) operatör olduğu için bu böyle olmaktadır. Bu operatörün açılımı ise şu şekildedir.

```
a = a - 1;
```

Burada yine — operatörünü başta kullandık. Size bu noktada tüyo vermemiz gerekirse bu operatörü tek başına kullandığınız zamanlar sağda diğer işlemlerle beraber kullandığınızda solda kullanmanızı tavsiye ederiz. Böylelikle yanlış işlem sonucundan korunmuş olursunuz.

Bir diğer konu ise aritmetik operatörlerde işlem önceliğidir. Öncelikle matematikte işlem önceliğini bir hatırlayalım. İşlemler soldan sağa doğru gitse de toplama ve çıkarmanın, çarpma ve bölmenin ayrı işlem dereceleri vardı. Önce soldan sağa çarpma ve bölme işlemlerini sırasıyla yapar sonrasında ise toplama işlemlerini yapardık. Burada da aynı şekildedir. Örneğin şöyle bir işlemin sonucunu hesaplayalım.

```
sonuc = 50 + 10 / 2;
```

Burada sonuç değeri 60 değil 55 olacaktır. Çünkü önce $10/2$ işlemi uygulanarak 5 değeri elde edilir ve sonrasında 50 ile bu toplanır. En son olarak da 55 değeri sonuç değişkenine aktarılır. Eğer işlem önceliğinde oynama yapmak istiyorsak yine matematikte olduğu gibi parentezleri kullanırız.

```
sonuc = (50 + 10) / 2;
```

Burada önce parantez içindeki 50 ile 10 toplanır ve 60 değeri bulunur. Ardından elde edilen bu değer 2 ile bölünerek 30 değeri elde edilir. Parantezleri iç içe kullanmamız mümkündür. Bu noktada sizi yormamak için bu kadar ayrıntı ile yetiniyoruz. İlerleyen konularda karşımıza çıktığı noktada size daha ayrıntılı şekilde anlatacağız.

C'deki operatörler bunlarla hiç sınırlı olmasa da biz sadece aritmetik operatörleri vererek burada bırakalım. İlerleyen konularda lazım olduğu zaman bütün operatörleri sizlere sıra sıra anlatacağız.

-12- C Veri ve Değişken Tipleri

Veri tipleri ve bunlar arasındaki ilişki programcılıkta çok önemli bir nokta olduğu için bunu ilk konularda anlatma kararı aldım. Bu başlıkta C'deki veri ve değişken tiplerini bütün ayrıntısıyla anlatmaya çalışacağız. Daha önceki örneklerde int ile tam sayı değişkeni tanımlamış ve char ile karakter değişkeni tanımlayıp kullanmıştık. Bunları dağınık halde öğrenmeniz yerine tablo halinde ne kadar değişken var yok öğrenip bunlara bakarak kullanmanızın zamanı gelmiştir. İlerleyen örneklerde örneklerin yanında kendi programlarınızı yazabilmenin yolu referansı okuyabilmekten geçer. O yüzden derslerin bir referans niteliğinde olmasını hedefliyoruz.

Değişkenleri, değişken tanımlamayı ve değerleri iyi anlamak program yazmanın en önemli kısımlarından biridir. Bunları iyi anladığınız sürece karşılaştığınız sorunlar azalır ve yazdığınız programların kalitesi artar.

Değişkenler sadece int değişkenadı; şeklinde adlandırdığınız ve içerisine tamsayı koyduğunuz yapılar değildir. Pek çok değer ve değişken türü bulunmaktadır. Bu sabit değerleri kullanabilmek için de yine değişkenleri kullanabilmek gereklidir.

Sabit değer dediğimizde 'a', 500, "Merhaba Dünya!" değerlerini söyleyebiliriz. Bunlar sabit değer olarak bir değişikliğe uğrayamaz ve programcı tarafından yazılır. Bu sabit değerleri saklayabilmenin ve üzerinde işlem yapabilmenin yolu da belleğe aktarmaktır. Değişkenler bellek alanında adlandırılmış bölgelerdir ve bu bölgelere bu değerler aktarılmaktadır. Artık bu bölgelerin adını çağırdığımızda o bölgelerdeki değerler bize gelmektedir. Bellekteki değişkenlerin konumunu şöyle bir tablo yaparak size gösterebilirim.

Adres	0x01	0x02	0x03	0x04	0x05
Değişken	int toplam_deger	int sonuc_degeri	int x	long uzun_bir_sayi	char a
Değer	500	100	25000	500000000	'b'

Burada gördüğünüz gibi değişkenler değerlere verilen isim değil adres değerlerine verilen isimlerdir. Örneğin toplam_deger olarak adlandırdığımız değişkenin adres değeri 0x01 olduğu için program 0x01 adres hücreindeki veriyi okuyup bize vermektedir. Bu adresleri ve adreslerdeki veriyi tek tek bizim ezberleyip kullanmamıza gerek yoktur. Onun yerine keyfimize göre değişkenler tanımlayıp değişkenler üzerinde rahatça işlem yapabiliriz. Assembly dilinde değişkenler olmadığı için adresler ve sayısal değerler üzerinde işlem yapmamız gerekir ve bu oldukça kafa karıştırıcı olabilir. Bizim

değişkenleri hafızada yer tutulan yerlerin adları olarak özetleyip böyle anlamamız gereklidir.

Değişkenleri en önemli noktadan özet bir şekilde sizlere anlattığımıza inanıyoruz. Şimdi C dilinde bu değişkenlerin nasıl olduğuna ve nasıl tanımlandığına bir bakalım. Öncelikle en temel değişkenler tam sayı tipinde olup belli değer aralığına sahiptir. Burada farklılaşma işaretli olup olmadığına yani eksi değer alıp almadığına göre ve hangi aralıkta değer alabileceğine göredir. Aşağıdaki tablodan temel değişken yapılarını inceleyelim.

Ad	Açıklama	Format İşareti
char	En küçük adreslenebilir değişken tipidir. 8-bitlik genişlikte olup -127, +127 arası değer alır.	%c
signed char	char değişkeni ile aynı olsa da bunun işaretli olduğunu garanti altına almaktadır. C dilinde normalde eğer işaretsiz değer olarak belirtmiyorsa işaretli olarak tanımlamaktadır. Bu değer de -127 ve +127 arası onluk sayı değeri olabilir. Karakterler de sayı değeri olduğu için ve 8-bitlik ASCII formatında olduğundan dolayı char değişkeni karakter depolama maksadıyla kullanılmaktadır.	%c
unsigned char	char ile aynı boyutta olsa da işaretsiz sayıları bulundurduğundan 0-255 arası değer olabilir. Bu durumda genişletilmiş ASCII karakterleri kullanacaksak bunu kullanmamız gereklidir. Çünkü ASCII yani harf, rakam, sembol gibi karakterlerde negatif değer yoktur.	%c
short short int	Kısa tam sayı değeridir. Short kısa anlamı taşıdığı için normal int değerine	%hi

signed short signed short int	göre daha az bellek alanı tutmaktadır. 16-bitlik değer işaretli olarak -32767 ve +32767 arası değer tutmaktadır. Normalde int kullansak da daha az yer kaplamasını istediğimiz durumlarda bunu kullanabiliriz.	
unsigned short unsigned short int	Kısa işaretsiz tamsayı değeridir. 0 – 65535 arası tam sayı değer alır. 8-bit mikrodenetleyiciler üzerinde çalışırken int değerinin karşılığı budur. Ama 32-bit temelli bir sistemde bunu short diye belirtmek gerekir.	%hu
int signed signed int	Temel işaretli tam sayı tipidir. Normalde yukarıdaki gibi -32767 ve +32767 değerleri arasında olsa da 32-bit sistemlerde -2 147 483 648 ve + 2 147 843 647 arasında değer almaktadır. Bu değer 32-bit genişliğindedir ve çoğu tam sayı işinde bize oldukça yeterli olmaktadır.	%i ya da %d
unsigned unsigned int	Eğer işaretli tam sayı değişkeni olan int değişkeni yeterli olmazsa bunun işaretsiz versiyonunu kullanabiliriz. Bu durumda değer 0 ile 4 294 967 295 arasında olacaktır.	%u

long long int signed long signed long int	32 bitlik işaretli tam sayı değişkenidir ve -2 147 483 648 ile + 2 147 483 647 arasında değer almaktadır. 32-bit derleyicilerde int ile aynı değeri alsa da farklı bir sistemde bu değer için long ifadesini kullanmak gerekebilir. int ifadesi sisteme göre değişkenlik göstermektedir.	%li
unsigned long unsigned long int	Bilgisayarda unsigned int ile aynıdır ve 4 baytlık bir adres uzunluğuna sahiptir. 0 ile 4 294 967 295 arası değeri bulundurur.	%lu
long long long long int signed long long signed long long int	Bu sefer gerçekten uzun sayıları barındırabilecek bir değişken görüyoruz. Bu değişken 64 bitlik bir genişliğe sahiptir. Bu da toplamda 8 baytlık bir alan demektir. C99 standartı ile literatüre girmiştir. -9 223 372 036 854 775 807 ile +9 223 372 036 854 775 807 arasında değer bulundurabilir.	%lli
unsigned long long unsigned long long int	Bu long long tipindeki değişkenin işaretsiz halidir. Böylelikle kapasitesi pozitif değerlerde ikiye katlanmaktadır. 0 ile +18 446 744 093 709 551 615 arası değer saklayabilir.	%llu

Görüldüğü gibi bu veri tipleri oldukça zengin ve istediğimiz tamsayı değerini saklamada kapasite bakımından sıkıntı yaşatmaz. Program yazarken değişken tipini iyi seçmek her zaman için çok önemlidir. Yüksek kapasite değişkenleri gereksiz yere kullanmak bellekte gereksiz yere yer tutulmasına ve performans kaybına sebep olacaktır. Yetersiz değişken kullanmak da azami değerler konusunda sıkıntı çıkaracaktır. Örneğin bir neslin yakından tanıdığı Knight Online oyunu üzerinden örnek verelim. Bu oyunda envanterimizde veya kasada toplam **2 147 483 648** oyun parası saklayabiliyorduk. Bu değer size tanıdık gelmelidir. Bu oyun parası oyunun ekonomisinden dolayı bir dönem oldukça komik bir rakam haline gelmişti. Oyun parasının değeri olmayınca 10 milyar, 20 milyarlık eşyaların alışverişini ancak gerçek para ile yapabiliyorlardı. Böyle bir hatanın tek sebebi de geliştiricinin `int oyun_parasi;` şeklinde oyun parası değerini tanımlamasıdır. Oyun parası negatif değer almadığı halde neden negatif değer alan bir değişken kullanılsın? Eğer işaretsiz bir değişken kullanılsaydı kapasite iki kat artardı. Eğer `long long` cinsinden bir değişken kullanılsaydı hiç kapasite sorunu yaşanmayacaktı.

Tabloda sağ tarafta gördüğünüz format işareti `printf` fonksiyonundan bize tanıdık gelecektir. Biz her değişken tipini yazdırırken formatına uygun olarak yazdırıyorduk. Burada da eğer `printf`, `scanf`, `sprintf` gibi “formatlı” fonksiyonları kullanacaksak bu format işaretlerini bilmemiz gerekir. Bu giriş ve çıkış fonksiyonlarını ayrıntısıyla anlatacağım için daha sonra bu konuya tekrar döneceğiz. C dilinde veri tipleri bitmese de tam sayı tiplerini bu yazıda bitirelim. En son olarak bir uygulama yapalım ve bu değişkenleri sahada görelim.

```

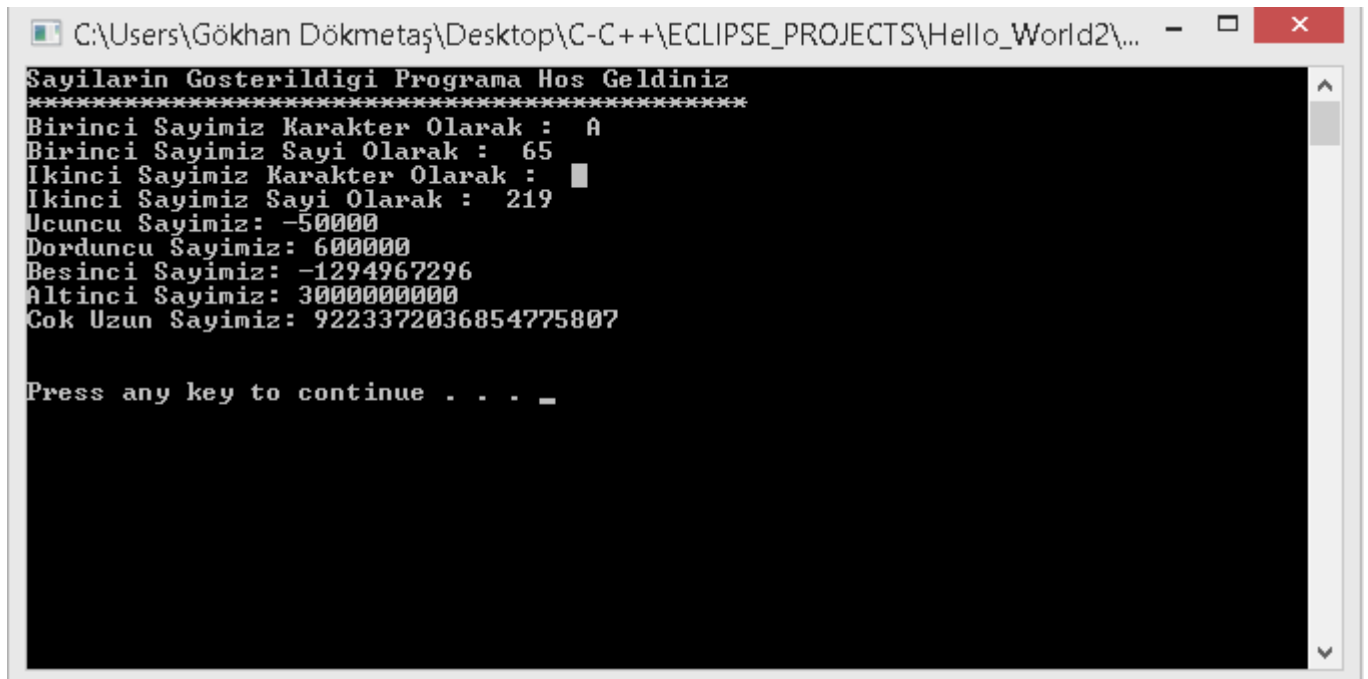
#include <stdio.h>
#include <stdlib.h>
int main() {
    char karakter = 'A';
    unsigned char genis_karakter = 219; // Kutu İşareti
    int sayi_1 = -50000;
    int sayi_2 = 600000;
    int sayi_3 = 3000000000;
    unsigned int u_sayi_3 = 3000000000;
    long long uzun_sayi = 9223372036854775807;

    printf("Sayilarin Gosterildigi Programa Hos Geldiniz\n");
    printf("*****\n");
    printf("Birinci Sayimiz Karakter Olarak :  %c \n",
karakter);
    printf("Birinci Sayimiz Sayi Olarak :  %i \n", karakter);
    printf("Ikinci Sayimiz Karakter Olarak :  %c \n",
genis_karakter);
    printf("Ikinci Sayimiz Sayi Olarak :  %i \n",
genis_karakter);
    printf("Ucuncu Sayimiz: %i \n", sayi_1);
    printf("Dorduncu Sayimiz: %i \n" , sayi_2);
    printf("Besinci Sayimiz: %i \n", sayi_3);
    printf("Altinci Sayimiz: %i \n", u_sayi_3);
    printf("Cok Uzun Sayimiz: %I64d \n \n \n", uzun_sayi);

```

```
system("PAUSE");  
} // Main Sonu
```

Program çalıştığında konsol ekranının çıktısı şu şekilde olacaktır.

A screenshot of a Windows console window. The title bar shows the file path: C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\... The console output is as follows:
Sayilarin Gosterildigi Programa Hos Geldiniz

Birinci Sayimiz Karakter Olarak : A
Birinci Sayimiz Sayi Olarak : 65
Ikinci Sayimiz Karakter Olarak : █
Ikinci Sayimiz Sayi Olarak : 219
Ucuncu Sayimiz: -50000
Dorduncu Sayimiz: 600000
Besinci Sayimiz: -1294967296
Altinci Sayimiz: 3000000000
Cok Uzun Sayimiz: 9223372036854775807

Press any key to continue . . . _

Konsol ekranında oldukça ilginç çıktıları görmekteyiz. Bazı çıktıların bizim yazdığımız değerlerle alakası yok. Öncelikle oluşturduğumuz değişkenlere ve bunlara atadığımız değerlere bakalım.

char karakter = 'A';

Burada karakter adlı char değişkenine 'A' değerini atadık. Bu sayısal bir değer olmasına rağmen harf değeri nasıl alıyor?, diye soracak olursanız bilgisayar sistemlerinde her harfin bir sayısal karşılığı vardır cevabını veririz. ASCII tablosuna baktığımızda A'nın 65 değerinin olduğunu görmekteyiz. Yani biz

elle 65 yazsak da eğer karakter formatında yazdırmak istiyorsak bilgisayar bunu A olarak yazdıracaktır.

unsigned char genis_karakter = 219; // Kutu İşareti

Normalde harfler ve rakamlar ASCII standartında 0 ile 127 arasındadır. Bu klavyedeki bütün rakam, harf ve sembollerin yanında bazı komutları da içinde bulunduracak kadar geniştir. Fakat iş başka dillere geldiğinde bu özel harfleri gösterecek değer yoktur. O yüzden Genişletilmiş ASCII ve Unicode kodlaması getirilmiştir. Biz unsigned char diye bir değer tanımlayarak genişletilmiş ASCII harflerini kullanabiliriz. Biz burada ilginç bir şekil olan kutu şeklini kullanmak istiyoruz. Klavyemizde kutu işaretli bir tuş olmadığından bunun rakamsal değeri olan 219'u yazacağız.

int sayi_1 = -50000;

Bu komutta int değişken tipinin negatif yani işaretli sayı alabildiğini göstermek için sayi_1 değişkeni tanımlayıp ilk değerini -50000 olarak verdik.

int sayi_2 = 600000;

Bu komutta int değişken tipine sıradan bir sayı değeri verdik.

int sayi_3 = 3000000000;

İşte bu nokta çok önemli. Çünkü biz burada int değişken tipine kapasitesinden fazla değer veriyoruz. Bu değişken tipine kapasitesinden fazla değeri vermenin sonucu ne olabilir?, sorusuna cevabı ise aşağıda göreceğiz.

```
unsigned int u_sayi_3 = 3000000000;
```

Burada yukarıda tanımladığımız değişkene aktardığımız aynı değeri unsigned yani işaretsiz tipine aktarıyoruz. Doğru değişken tipi seçmenin önemini konsol ekranının incelerken göreceğiz.

```
long long uzun_sayi = 9223372036854775807;
```

Burada ise çok uzun bir sayıyı değişkene aktarıp göstermek için böyle bir değişken tanımladık ve kendisine çok çok uzun bir sayı yükledik. Şimdi bir yandan printf fonksiyonlarını inceleyelim bir yandan da konsol ekranına satır satır bakalım.

```
printf("Birinci Sayımız Karakter Olarak : %c \n", karakter);
```

İlk fonksiyonda birinci sayıyı göstermek için karakter değişkenini %c formatı ile yazdırdık. Normalde tamsayı yazdırmak için %i formatını kullanıyorduk. %c formatı ile yazdırdığımız için ekranda "A" harfi olarak görüldü.

```
printf("Birinci Sayımız Sayı Olarak : %i \n", karakter);
```

Burada yine karakter değişkenini farklı bir formatla yazdırdık. Tam sayı formatında yazdırdığımızda yine sayısal değerini görme imkanımızın olduğunu gösterdik. Ekranda 65 değeri olarak "A" karakterinin sayısal değerini görüyoruz.

```
printf("İkinci Sayımız Karakter Olarak : %c \n", genis_karakter);
```

Bu sefer ekranda genis_karakter değişkenini %c formatında yani ASCII olarak yazdırdık. Ekranda bir kutu görmekteyiz. Böyle sembolleri kullanarak ilginç işler yapılabilir. Hatta ASCII sembolleriyle uzun yıllardır oyun yapılmaktadır.


```
You feel something fly from your pack! Its cries sound like "daddy."

-----
|.....|
|.....## -----
--.----- # |.....|
# #####|.....| -----#
# ##|.....##)###|<.....|#
# ##.....| #.....|###
# #----- |...%.....).# -----
# ### |.....|#####|.....|
# # -----### |.....|
# # ### # |.....|
## ----# ###|.....|
--.----- #. |% #.....).....|
|.....#####|. # |.....@.....|
|.....| | |
|.....| ----
|.....|
-----

Chomzee the Joshu St:16 Dx:17 Co:18 In:8 Wi:8 Ch:10 Lawful S:29707
Dlvl:11 $:1972 HP:100(100) Pw:25(42) AC:-4 Xp:10/6766 T:6628 Blind
```

Resim: ASCII Karakterleri ile görselleştirilmiş bir rol yapma oyunu

printf("İkinci Sayımız Sayı Olarak : %i \n", genis_karakter);

Burada yukarıda tanımladığımız "kutu" karakterinin sayısal değerini görmekteyiz. Değişken üzerinde bir değişiklik olmadı sadece formatı %c'den %i olarak değiştirdik.

printf("Üçüncü Sayımız: %i \n", sayi_1);

Burada olağan bir işaretli sayı yazdırma işlemini görmekteyiz.

printf("Dördüncü Sayımız: %i \n", sayi_2);

Burada da yine bir int değerini sınırları içerisinde yazdırdık. Daha önceki örneklerden alışkın olduğumuz için şimdi anlatacağımız bir şey kalmadı.

```
printf(“Besinci Sayimiz: %i \n”, sayi_3);
```

Burada ise işler değişmektedir. Biz bu sayıya “3000000000” değerini versek de bu sayı ekranda -1294967296 olarak gözükmemektedir. Küçük bir dikkatsizliğin nasıl bir sonucu olabileceğini buradan görebilirsiniz. Peki bu değer niçin böyle yazılmaktadır?, sorusuna cevabı burada verelim.

Öncelikle bilgisayarlarda negatif bir değer yoktur. Negatif değerleri temsil eden 1 ve 0 değerleri vardır. Bir değişken eğer işaretsiz olarak tanımlanırsa bütün bitlerini sayı depolamak için kullanır. Eğer işaretli yani negatif değer alırsa sayının negatif mi pozitif mi olduğunu belirtmek için bir işaret bitine ihtiyaç duyar. Bu da en sol taraftaki bit olmaktadır. Yani en sol bit “1” ise sayı negatif “0” ise pozitifdir. Negatiften pozitive çevirme işi de bu bitin değerini çevirmekle olur. Biz bu değişkene 3000000000 değerini yazmakla değişkene ikilik tabanda 10110010110100000101111000000000 değerini yazmış oluruz. 32-bitlik bu değer en başında 1 sayısı var değil mi ? O halde program bunu negatif sayı olarak değerlendirecektir ve geri kalan değeri okuyacaktır. Çünkü biz değişkeni tanımlarken bunun işaretsiz olduğunu programa belirtmedik. Geri kalan değeri okuduğunda ise -1294967296 sayısını bulacaktır. Program aslında gayet doğru çalışmaktadır burada bizim yaptığımız hata söz konusudur. Bunun sebebini bilmek bayağı ileri seviye bir bilgi olsa da şimdiden size anlatalım dedik.

```
printf(“Altinci Sayimiz: %i \n”, u_sayi_3);
```

Burada unsigned int olarak tanımladığımız değişkenin değeri olması gerektiği gibi gözüküyor. Değer aralıklarına dikkat etmenin önemini uygulama üzerinde görüyoruz.

```
printf("Cok Uzun Sayimiz: %l64d \n \n \n", uzun_sayi);
```

Burada bu çok uzun sayıyı yazdırmak için neden **%l64d** formatını kullandığımı sorabilirsiniz. Yukarıda %lli anlatsak da bizim kullandığımız MinGW derleyicisi maalesef bunu desteklememektedir. O yüzden bu derleyicinin anlayacağı dildeki parametreyi araştırıp bulduk. Standartlara uygun kod yazsak da derleyici tabanlı bir sorunla da bayağı erken karşılaşmış olduk. Öğrenme aşamasında karşılaşılan sorunlar ve hataları öğrenmek ve bunlara çözüm getirmek çok önemlidir. Çünkü hiçbir zaman öğrenirken her şey dört dörtlük ilerlemez. Öğrenci tek başına araştırma yaparak ve deneyerek hatanın çözümünü aramak zorundadır. Eğer hatalardan bunalıyorsanız bu sektörde çalışmak sizin için işkence gibi olacaktır. Çünkü hangi noktada olursanız olun hatalarla sürekli yüzleşmeniz gerekecektir.

Bir sonraki başlıkta veri tiplerine devam edeceğiz.

-13- C Veri ve Değişken Tipleri – 2

Önceki başlıkta veri tipleri olarak sadece temel tamsayı değişkenleri anlatsak da yazı bayağı uzadı ve yorucu oldu. Şimdi ise daha kalan kısmını bu başlıkta anlatacağız. Basit tamsayı dışındaki değişkenleri tamsayı değişkenlerini anlamaktan daha zor olacağını hesap edebiliriz. Burada en sık kullandığımız

değişken tipi kullanacağımız kayan noktalı (ondalıklı) sayı değeri olsa da öncelikle tam sayıları bitirmek adına sabit genişlikli tam sayı tiplerini sizlere aktaralım.

Sabit genişlikli tam sayıları anlamak için öncelikle temel tamsayı değişkenlerini incelememiz gereklidir. Biz bu değişkenleri anlatırken bunların değer aralığının kullanılan derleyiciye ve platforma göre değişiklik gösterdiğini söylemiştik. Yani bir platformda program yazarken int değişkeni -32768 ve 32,767 arasında değer alırken bir platformda bu değer aralığı -2 147 483 648 ve 2 147 483 647 arasında olabilir. Bu durumda birinde yazdığımız program çalışırken ötekinde sürprizlerle karşılaşabiliriz. C dilinde de C99 standartından itibaren stdint.h adında bir kütüphane dosyasında tamsayı değişkenlerinin genişlikleri belirlenmiş ve programcının kullanımına sunulmuştur. Biz bilgisayarda pek kullanmasak da gömülü sistemlerde bu değerleri öncelikli olarak kullanırız.

Gömülü C konusunda geldiğimizde bütün ayrıntısıyla inceleyecek olsak da şimdilik bu değişken tiplerinden en çok kullanılanları sizlere aktaralım.

Değişken Adı	Açıklama
int8_t	-127 ve +127 arası işaretli değer alan 8-bit tam sayı
int16_t	-32 768 ve 32 767 arası işaretli değer alan 16-bit tam sayı
int32_t	-2 147 483 648 ve 2 147 483 647 arası değer alan 32-bit işaretli tam sayı

int64_t	-9 223 372 036 854 775 807 ve +9 223 372 036 854 775 807 arası değer alan 64-bit işaretli tam sayı
uint8_t	0-255 arası değer alan 8-bit işaretli tam sayı
uint16_t	0-65535 arası değer alan 16-bit işaretli tam sayı
uint32_t	0 – 4,294,967,295 arası değer alan 32-bit işaretli tam sayı
uint64_t	0 ve +18 446 744 073 709 551 615 arası değer alan 64-bit işaretli tam sayı

Bu tablodan başka kütüphane dosyası içerisinde tip tanımlamaları ve makrolar yer alsa da bunlar gömülü C derslerinde işimize yarayacağı için o bu kadarıyla yetiniyoruz. Anladığınıza emin olmak için bu değişken tiplerini örnekler üzerinde gösterelim.

```
int8_t karakter;
uint8_t _bayt = 200;
uint64_t = cok_uzun_sayi = 999999999999;
```

Okunabilirliği düşük olup temel tamsayı değişkenlerinden gözümüze yabancı görünse de alt seviye programcılığa alıştığınız zaman int, char, long gibi tabirleri kullanmayı unutursunuz.

Veri tipleri konusunda en son olarak ondalıklı sayı değişkeni olan float, double ve long double değişken tiplerinden bahsedeceğiz. Örneğin PI sayısını

kullanarak bir hesaplama yapacağız fakat pi sayısını kaydetmemiz gereken bir değişken gerekiyor. Üstelik PI sayısını sabit olarak kullansak da (3.1415F gibi) bu sefer de PI sayısını kullanarak yaptığımız işlemin sonucu virgüllü olarak çıkmaktadır. Bu durumda bu değerleri tam sayı olan int, long, char gibi değişkenlere aktarmamız mümkün olmaz. Değerler aktarılsa dahi yuvarlatılarak bize verilir. Örneğin sonuç 3.6 çıkarsa bu 4 değeri olarak gösterilir. 3.4 çıkarsa bu 3 değeri olarak gösterilir. Bu elde edilen sonuç üzerinden bir işlem daha yapılırsa hesaplama çok yanlış elde edilir. O yüzden ondalıklı değer elde edilmesi muhtemel yerde float, double ve long double değişken tiplerini kullanmak gereklidir.

Sadece ondalıklı değer elde edeceğimiz değil elde edilme imkanı olan yerde de bunu kullanmakla önceden önlemi almak gereklidir. Şimdi bu değişkenlere bakalım.

Değişken Tipi	Açıklama	Format
---------------	----------	--------

float	IEEE 754 tek hassasiyette ikilik kayan noktalı formatında 32 bitlik kayan noktalı değişken. $1.2E-38$ ve $3.4E+38$ arasında virgülden sonra 6 haneye kadar rakam tutabilir.	%f dijital, %g bilimsel
double	64-bitlik kayan noktalı değişken. $2.3E-308$ ve $1.7E+308$ arası değer alabilir ve virgülden sonra 15 haneye kadar ondalıklı sayı destekler.	%lf
long double	10 baytlık ve $3.4E-4932$ ve $1.1E+4932$ arasında değer alabilen ondalık sayı değişkeni. Virgülden sonra 19 haneye kadar değer alabilir.	%Lf

Şimdi bir uygulama yapalım ve iki ondalıklı sayıyı birbiriyle çarpalım. Hem girişte ve hem çıkışta ondalıklı değer elde edeceğiz.

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

    double ilk_sayi, ikinci_sayi, sonuc;
    printf("İki Ondalikli Sayi Girin: ");

    scanf("%lf %lf", &ilk_sayi, &ikinci_sayi);

    sonuc = ilk_sayi * ikinci_sayi;

    printf("Sonuc = %.2lf \n", sonuc);

    system("PAUSE");
} // Main Sonu
```

Burada double tipinde üç değişken tanımlayarak programa başlıyoruz. Bu üç değişkenin de double olması çok önemlidir. Örneğin iki tam sayıyı bölüp ondalıklı sayı elde edeceksek sadece sonuç değişkenini double olarak tanımlamamız yeterli olurdu. Fakat burada “.” işareti ile değişkenlere ondalıklı değer atayacağız ve bunların çarpımını yine ondalıklı değer olarak kaydedeceğiz. Konsola ondalıklı sabit değerleri yazmak için 20.15, 3.1415 gibi nokta işaretini kullanmamız gereklidir. Matematikte virgöl kullanılsa da burada

nokta kullanılmaktadır. Bu nokta kullanılma durumu aslında ülkelere göre değişmektedir. Orada matematikte de virgülden sonrası değil noktadan sonrası kavramı vardır.

```
scanf("%lf %lf", &ilk_sayi, &ikinci_sayi);
```

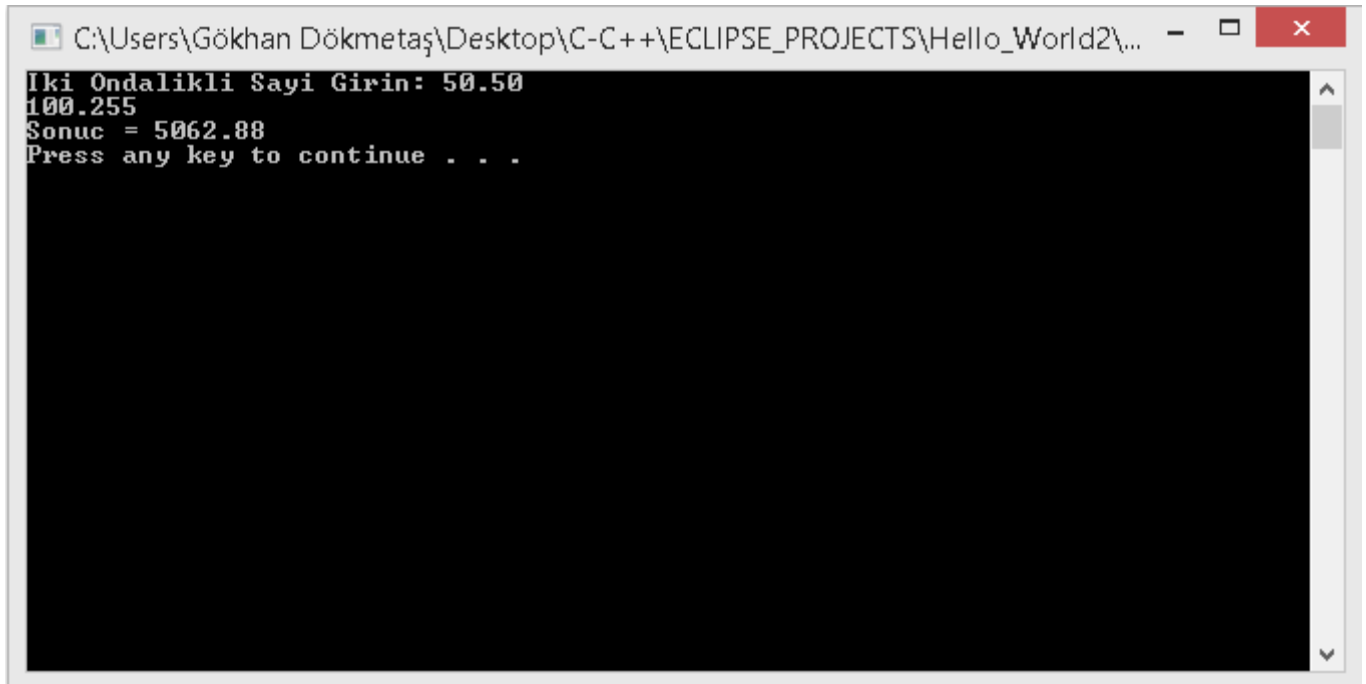
Bu komutta scanf fonksiyonunun daha önceden görmediğimiz bir kullanımını görüyoruz. Tek bir scanf fonksiyonu iki farklı değer alıp iki farklı değişkene değer ataması yapıyor. Bunları yaparken sırasını gözetmek ve format işaretini “%” koymak lazımdır. Burada format yukarıda belirttiğimiz üzere %lf şeklindedir. Bu formatları doğru yazmamız değişken tanımlamak kadar önemlidir.

```
sonuc = ilk_sayi * ikinci_sayi;
```

Bu çarpma işlemi aynı tamsayılarda olduğu gibi olsa da program bizim double şeklinde tanımladığımız değerleri bildiği için ondalık sayı çarpımı şeklinde işlemi yapmaktadır.

```
printf("Sonuc = %.2lf \n", sonuc);
```

Burada format kısmında %.2lf ifadesini kullanıyoruz. Yani virgülden sonra iki basamağı göstermesini fonksiyona söylüyoruz. Konsol ekranında da bu şekilde görmekteyiz.

A screenshot of a Windows console window titled "C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...". The console has a black background with white text. It displays the following text: "İki Ondalıkli Sayı Girin: 50.50", "100.255", "Sonuc = 5062.88", and "Press any key to continue . . .". The window has standard Windows title bar controls (minimize, maximize, close) in the top right corner.

```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...
İki Ondalıkli Sayı Girin: 50.50
100.255
Sonuc = 5062.88
Press any key to continue . . .
```

Buraya kadar veri tipleri hakkında anlatmamız gereken başka bir konu kalmadı. Daha ileri için `inttypes.h` dosyasındaki tanımları ve makroları incelemenizi tavsiye ederiz. Biz daha ileri seviye konularda bundan bahsedeceğimiz için temel seviye olarak bu anlattıklarımız bile ileri seviye kalmakta.

Temel C Programlama -14- Döngülere Giriş

Artık yapısal programlama dair temel kavramlara giriş yapıyoruz. Bu alanda ilk öğrenmemiz gereken konulardan biri döngülerdir. Döngüler routine (yordam) ve subroutine (altyordam) olarak Assembly dilinde makine komutları vasıtasıyla da yapılmaktadır. Çünkü bir programda en temel noktalar aritmetik işlem, mantık işlemleri, veri okuma ve yazma ve program döngüsüdür. Şu ana kadar programları yukarıdan aşağıya doğru satır satır bir çizgide yazıyorduk ve program sırayla komutları işleyerek devam ediyordu. Biz program dosyasında bir noktaya gitmek istediğimizde veya bir program blokunu devamlı ya da bir süre çalıştırmak istediğimizde döngülere müracaat ederiz. Önceki verdiğimiz program örnekleri döngü ve karar mekanizması olmadan çok basit programlardan ibaret kalıyordu. Döngüleri ve ardından kararları öğrendiğimiz vakit artık daha karmaşık program yazabileceğiz.

Döngü ve karar mekanizmaları algoritmanın temelini oluşturmaktadır. Sadece döngü ve karar mekanizmalarını öğrenmekle bunları etkin ve doğru kullanacağımız anlaşılmaz. Bizim algoritma bilgimiz dahilinde bunları etkin bir biçimde kullanabiliriz. Algoritma kısaca bir problemi çözmeye yönelik işletilecek komutların belli bir düzende sıralanması demektir. Bu algoritma talimatlarını günlük hayatımızdaki işlere de uygulayabiliriz. Örneğin kahvaltı hazırlamak yönünde bir algoritma yazmamız gerekirse şu işleri sırayla yürütmemiz gerekir.

- **Mutfağa Git**
- **Buzdolabını Aç**
- **Peyniri ve Zeytini al**
- **Çaydanlığa su koy**
- **Ocağı yak**
- **Demliğe çay koy**
- **Su kaynayana kadar bekle**
- **Su kaynadıysa çayı demle**
- **Ekmek dolabına bak**
- **Ekmek varsa kahvaltıya başla**
- **Ekmek yoksa ekmek almaya git**
- **Ekmeklerin tamamı dilimlenene kadar ekmekleri doğra**

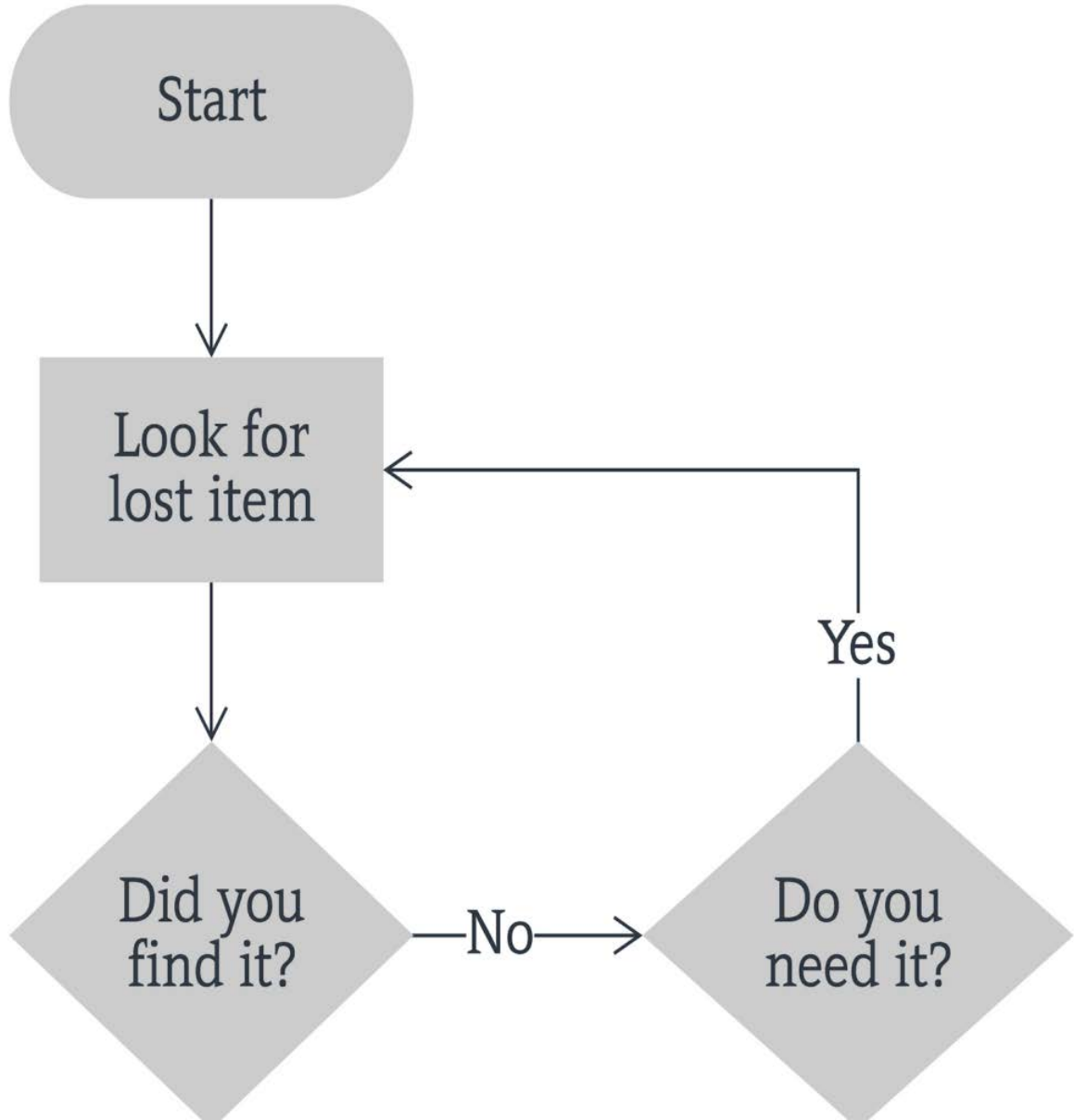
Burada sadece düz talimatları değil belli şart ve döngüye bağlı talimatları da görmekteyiz. Örneğin su kaynayana kadar bekle dediğimizde bekleme işini suyun kaynamasına bağlarız ve suyun kaynayıp kaynamadığını sürekli denetleriz. Su kaynadıysa çayı demle talimatı da çayın demlenmesini suyun kaynamasına bağlamaktadır. Su kaynamadığı zaman çay demlenmeyecektir.

Ekmeğin var olup olmadığına göre iki farklı iş yapılmaktadır. En sonda ise ekmeklerin tamamı dilimlenene kadar doğrama işine devam etmemiz söylenmektedir. Eğer böyle bir şart koşmasaydık aynı işi şu şekilde yapmamız gerekirdi

- Ekmeği dilimle
- Ekmeği dilimle
- Ekmeği dilimle
- Ekmeği dilimle
- ...

Ekmeklerin tamamı dilimlenene kadar aynı işi defalarca sıralamak gerekirdi. Görüldüğü gibi gerçek hayatta kurduğumuz algoritma şimdi anlatacağımız döngülere ihtiyacı doğurmaktadır. Yani bu karar ve döngü yapılarını gerçek hayatta biz işimizde de kullanmaktayız. Gerçek hayattaki olayları programa aktarırken de bu algoritmaya göre bir akış diyagramı çizeriz. Bu akış diyagramını programa uygulamak içinse programlama dilindeki öğeleri ve kuralları kullanırız.

Bilgisayar programı yazmadan önce ise “Yalancı Kod” adı verilen günlük hayat diline daha yakın fakat bilgisayar mantığını içeren kodları yazarız. Oturup doğrudan kod yazmaya başlamadan önce bunları kullanmamız günlük hayat işlerini algoritmaya aktarmak ve bunu da program kodu haline getirmek arasında bir köprü görevi görmektedir. Bir insanın günlük hayat işlerini doğrudan doğruya programa aktarması zor bir iştir. Aynı Google Translate’in eskiden yaptığı çeviriler gibi düzensiz ve kararsız bir program ortaya çıkartır. O yüzden yazacağımız programı iyi düşünmeli ve kafamızda kurgulamalıyız. Ardından yalancı kod ve akış diyagramları kurmak kodu yazma esnasında bize yardımcı olacaktır.



Resim: Basit bir akış diyagramı

Programlama mantığını anlamak için akış diyagramları üzerinde çalışmak faydalı olacaktır. Akış diyagramı programlarını kullanarak basit programlar yazmak günümüzde kullanılan kutucukları birleştirerek program yazma işlemine (kodlama) benzese de profesyonel bir yöntem olarak her programlama diline uygulanabilir yapıdadır. Çünkü hazır kod veya fonksiyon

değil en temel fonksiyonları yerine getiren diyagramları görürsünüz. Biz derslerimizde akış diyagramlarını anlatmayacağız.

Döngüler

Döngülerin yaptığı iş yukarıdan aşağıya doğru inen program akışını belli bir blok içerisinde bir şarta bağlı olarak sınırlandırmaktır. Bu aynı tren hatlarının makaslarla değiştirilmesi ve bir trenin bir hatta devamlı gitmesi gibidir. Bu hattın dairesel olduğunu yani dönüp durduğunu hayal edebilirsiniz. Bu durumda dönüp durulan bir hat olduğu için bunun adına döngü adı verilmektedir. Her döngünün başı ve sonu bulunmaktadır ve döngüden çıkılmadığı sürece döngünün sonuna gelindiğinde program akışı döngünün başına atlar ve döngünün sonuna kadar olan komutları işlemeye devam eder. Bir döngüyü şöyle tarif edebiliriz.

Döngü Başlangıcı -Komut -Komut -Komut -Komut Döngü Başına Git Komutu (Döngü Sonu)

Burada önce program akışı döngü başlangıcına girerek döngüye giriş yapmış olur. Sonrasında komutları habersiz olarak işlemeye başlar sonrasında ise döngü sonunda “Döngünün Başına Git” komutunu işlemek zorunda kalır. Bu durumda döngünün başına giderek az önce işlediği komutları tekrar işlemek zorunda kalır ve bu devamlı olarak böyle devam eder.

Sonsuz döngünün olmaması için döngü şartı kullanılır. Böylelikle döngüden çıkmak bir şarta bağlanmış olur. Örneğin 10 kere döngüde döndükten sonra döngüden çık, kullanıcı düğmeye basarsa döngüden çık gibi bir şart eklemek program akışını istediğimiz gibi denetlememizi sağlar. Döngü şartı ile döngü şu şekilde işleyecektir.

Döngü Başlangıcı -Komut -Komut -Komut Eğer Şart Sağlanmazsa Döngü Başına Git (Döngü Sonu)

Bu durumda program akışının devamı şarta bağlanmış olup bizi sonsuz döngüden kurtarmaktadır. Biz burada komutları belli bir şart sağlanana kadar şartı sağlayabilmek kaydıyla yazarız ve şart sağlanınca döngüde kalmanın bir anlamı zaten kalmaz. Döngünün çalışma sebebi şartı sağlamak olduğundan döngü şartının önemli bir işi denetlemesi gereklidir. Örneğin bir harfi 100 kere yazdırmak istiyorsak şöyle bir komut kullanırız.

Döngü Başlangıcı -Harf Yaz Eğer 100 Kere Yazılmazsa Döngü Başına Git

Bu durumda 100 kere harf yazıldıktan sonra döngünün çalışması için bir sebep kalmayacaktır ve program akışına devam edecektir. Biz 100 kere bir harf yazdırmak istediğimiz zaman döngüleri kullanmasaydık 100 kere “Harf Yaz” komutunu peşpeşe yazmamız gerekecekti. Bu durumda hem biz yorulacaktık hem de program hafızasında gereksiz yere program yer kaplayacaktı. Bu yüzden tekrarlanan komutları işletmek için döngüleri kullanmak hem işimizi kolaylaştırır hem de performanslı program yazmamıza yardımcı olur.

Yukarıda gördüğünüz gibi herhangi bir program komutu kullanarak kafanızı karıştırmadık. Bu yazdıklarımız kendimize ait olan yalancı kodlardı. Yalancı kodları herkes kendine göre istediği dilde yazabilir. Yalancı kodun konuşma diline yakın olması ve aynı zamanda bilgisayarın mantığını içermesi etkili olması için yeterlidir. Döngülere giriş konumuz yeterince uzadığı için burada bitiriyoruz sonraki başlıkta döngü şartlarını sağlayan ilişki operatörlerini anlatacağız.

Temel C Programlama -15- İlişki Operatörleri

Döngüler ve kararlarda bizim sürekli kullanacağımız ilişki operatörleri denen iki değeri karşılaştırıp 1 veya 0 olarak sonuç veren operatörler vardır. Bu operatörler normal komutların içerisine de yazılabilse de önemli nokta bunların döngü anahtar kelimelerinden sonra gelen parantezlerin içine yazılan operatörler olduklarıdır. Bu operatörler iki değeri karşılaştırır ve operatörün cinsine göre sonuç verir. Bazı kaynaklarda “İlişkisel operatörler” olarak bahsedilse de Türkçe’de -sel veya -sal kelimelerini kullanmak doğru bir yaklaşım değildir. Latince -al ekinden gelen bu nispet eki yerine zamanında Osmanlıca’da Arapça’dan gelen nispet î ‘si olarak kullanılmıştır. İşin aslında baktığımızda ise doğru Türkçe’de belirtisiz ad tamlaması olarak kullanıldığını veya -lık ekinin kullanıldığını görüyoruz. O yüzden biz “İlişkisel Operatör” yerine “İlişki Operatörü” demeyi tercih ettik. Bazı -sAl ekli tabirler dilimize o denli yerleşmiştir ki bunları alışkanlık ve zorunluluk icabı biz de kullanmaktayız.

Şimdi ilişki operatörlerini tablo üzerinde görelim ve bunlar üzerinden örnek yaparak nasıl çalıştıklarını öğrenelim.

Operatör	Açıklama	Örnek
==	“Eşittir” operatörü. İki değeri karşılaştırır ve iki değer eşitse DOĞRU yani 1 sonucunu verir.	(A == B)
!=	“Eşit Değil” operatörü. İki değeri karşılaştırır ve iki değer eşit değilse DOĞRU yani 1 sonucunu verir. Değerler eşitse YANLIŞ yani 0 sonucunu verir.	(A !=B)
>	“Büyüktür” operatörü. A değeri B değerinden büyükse DOĞRU yani 1 değilse YANLIŞ yani 0 değerini verir.	(A > B)
<	“Küçüktür” operatörü. A değeri B değerinden küçükse DOĞRU yani 1 değilse YANLIŞ yani 0 değerini verir.	(A < B)
>=	“Büyük veya Eşit Operatörü” A değeri B değerinden büyük veya B değerine eşitse DOĞRU yani 1 değilse YANLIŞ yani 0 değerini verir.	(A >= B)
<=	“Küçük veya Eşit Operatörü” A değeri B değerinden küçük veya B değerine eşitse 1 değerini değilse 0 değerini verir.	(A <= B)

Bu ilişki operatörlerini kullanarak bir program yazalım. Daha sonra bu operatörleri döngü ve kararlarda kullanacağız. Şimdi tek başına nasıl kullanılıyor ona bakalım.

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

    int a = 5;
    int b = 5;
    int sonuc;

    sonuc = a == b;
    printf("A = 5 ve B = 5, A == B sonucu : %i \n", sonuc);

    a = 2;

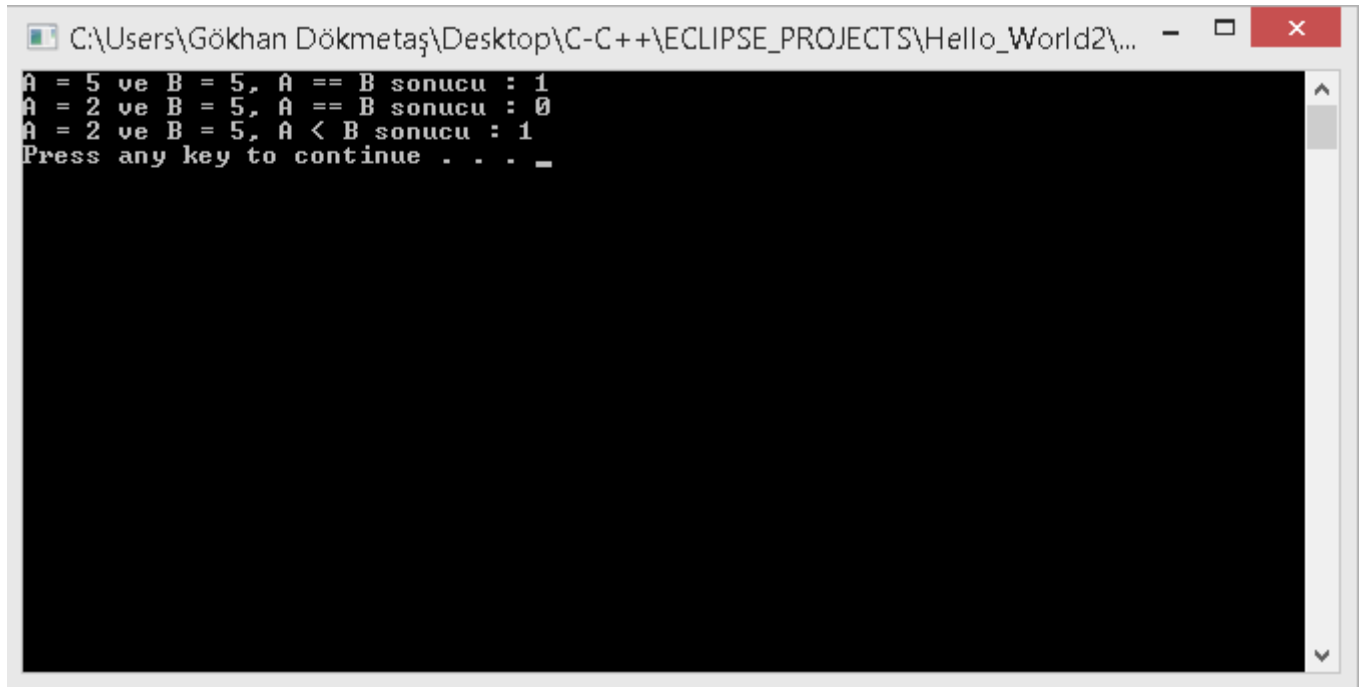
    sonuc = a == b;
    printf("A = 2 ve B = 5, A == B sonucu : %i \n", sonuc);

    sonuc = a < b;
    printf("A = 2 ve B = 5, A < B sonucu : %i \n", sonuc);

    system("PAUSE");
} // Main Sonu
```

Programda fazla zaman harcamamak adına bütün ilişki operatörlerini kullanmıyorum fakat bu kadarı bile anlamanız için yeterli olacaktır. Eğer kafanızda soru işareti oluşursa kendiniz operatörleri değiştirerek ve ona göre

kodu düzenleyerek uygulamada görebilirsiniz. Şimdi nasıl bir çıkış verdiğine bakalım.



```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...  
A = 5 ve B = 5, A == B sonucu : 1  
A = 2 ve B = 5, A == B sonucu : 0  
A = 2 ve B = 5, A < B sonucu : 1  
Press any key to continue . . . _
```

Görüldüğü gibi programda hangi işlemi yaptığımızı da açık açık yazdık. Böylelikle hangi işlemin ne sonuç verdiğini daha rahat takip etme şansımız oldu. Öncelikle a ve b adında iki değişken tanımlayıp sonuc değerine bu ilişki operatörlerinin işlem sonucunu atadık ve bunu ekranda yazdırdık. A == B dediğimizde A ve B'nin değerleri 5 olduğu için "Eşittir" durumu sağlandığından 1 sonucunu elde ederiz. a'nın değerini değiştirip yine A == B işlemine tabi tuttuğumuzda bu sefer 0 sonucunu alırız çünkü A ve B birbirine eşit değildir. Fakat A, B'den küçük olduğu için A küçüktür B işleminde 1 sonucunu elde ederiz.

Çok önemli bir nokta ise atama operatörü "=" ile eşittir operatörü'nün "==" birbirine karıştırılmasıdır. Bazen program yazarken bilgisizlikten dolayı olmasa bile dalgınlık veya tuşun basmamasından dolayı "==" yerine "=" yazdığımız olabilir Bu iki operatör de birbirine benzediği için kodları çok dikkatli incelemedikçe hata gözümüzden kaçacaktır. Bu hataya derleyici de uyarı

vermemektedir ve program o haliyle çalışmaktadır. Programcılıkta yapılan en kritik hatalardan biri olarak “=” operatörü ile “==” operatörünü birbirine karıştırmayı sayabiliriz ve bu konuda yeteri kadar dikkatli olmanız konusunda sizi uyarıyoruz.

Bu yazıda ilişki operatörlerini anlattık ve artık döngüleri öğrenmemiz konusunda önümüzde bir engel kalmadı. Bir sonraki başlıkta C dilindeki döngüleri sizlere anlatacağız.

-16- For Döngüsü

C dilinde döngüler aslında birbirine benzer yapıda olup döngü blokları içerisinde belli bir şarta bağlı olarak program akışını döndüren yapılardır. Yapı olarak döngü bakımından değil yöntem bakımından farklılıkları vardır. Çoğu zaman bu döngüleri birbiri yerine kullanma imkanımız olur. Yine de “yazılı

olmayan” belli başlı kurallara göre bu döngüleri kullanırsak daha kaliteli program ortaya çıkarırız. C dilindeki döngüler **for**, **while** ve **do..while** döngüleridir.

for döngüsü

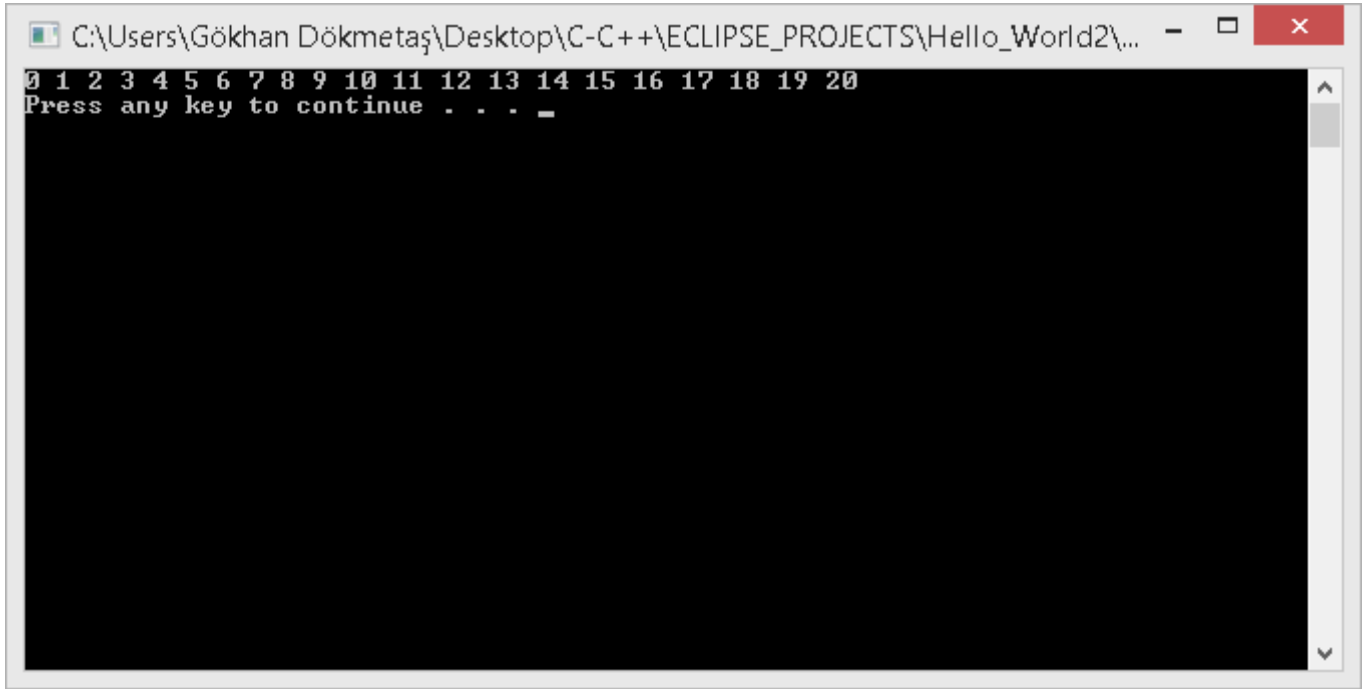
C dilinde for döngüsü anlaması en kolay döngüdür. O yüzden bu döngüyü en başta anlatma kararı aldık. for döngüsünün bütün kontrol elemanları tek bir yerde toplanır ve buradan bütün döngüyle alakalı işler yürütülür. Bu yüzden hazır yazılmış bir programa baktığınızda döngünün kaç kere çalışacağını, hangi şarta bağlı olduğu gibi durumları rahatça görebilirsiniz. Siz de program yazarken tek satırda bütün döngü işini bitirdiğinizden kafanız rahat eder. Acaba döngüden çıkamayacak mıyım?, Döngü şartlarını sağladım mı? gibi sorularla kafanızı yormazsınız. Fakat for döngüsünün biz dezavantajı (kullanımından dolayı olan dezavantajı) döngü blokunu belli bir miktar çalıştırmasıdır. Hemen bir for örneğin yapalım ve bunun nasıl işlediğini görelim. Siz de ne demek istediğimizi böylelikle anlayacaksınız.

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

    for (int i = 0; i <= 20; i++)
    {
        printf("%i ", i);
    }

    printf("\n");
    system("PAUSE");
} // Main Sonu
```

Programı derleyip çalıştırdığımızda şu şekilde bir ekran görüntüsü alırız.

A screenshot of a Windows command prompt window. The title bar shows the file path: C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\... The window contains a black background with white text. The first line shows a sequence of numbers from 0 to 20, each followed by a space. The second line says "Press any key to continue . . . _".

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Press any key to continue . . . _
```

Burada biz yazmasak da 0'dan itibaren saymaya başlandığını ve 20 ye kadar bu sayma işleminin devam ettiğini görüyoruz. Bizim yazdığımız komut sadece **printf("%i ", i);** komutuydu ve bu komut da i değişkenini ekrana yazdırıyordu. O halde bu yazdırma işini 0'dan 20'ye kadar tekrar tekrar yürüten ve her yürütmede i değişkenini bir artıran bir program yapısı olmalıdır.

for (int i = 0; i <= 20; i++) yapısı ile bütün döngü işlemlerini bir arada topluyoruz. for anahtar kelimesinin ardından parantez içerisine noktalı virgüller ile üç ayrı komut yazılıyor. Bunlardan birincisi tanımlama komutu, ikincisi şart komutu, üçüncüsü ise artırma komutudur. Bundan sonra noktalı virgül gelmediğine dikkat ediniz. for yapısını sınırlandırmak için ise süslü parantezleri kullanmaktayız. Şimdi örnek bir for yapısını inceleyelim ve anlattıklarımızı söz diziminde toplayalım.

```
for ( ilk_tanim; şart; artırma)
{ // for başlangıcı
```

```
    komutlar;  
    komutlar;  
    komutlar;  
} // For bitişi  
  
for ( ilk_tanim; şart; artirma)  
komut; // tek komut yapısı
```

Öncelikle for iskeletini üç ana unsurun oluşturduğunu görüyoruz. İstersek bunları esgeçebiliriz artırmayı blok içinde yapabiliriz veya ilk tanım değerini önceden tanımladığımız bir değişken olarak alabiliriz. Bu yönde bir esneklik bize verilse de olması gereken for yapısını başlangıçta biz inceleyelim. for döngüsü bir değişkenin değerine göre alt tarafta süslü parantez ile sınırlandırdığımız kod blokunu çalıştırmaya devam eden döngü yapısıdır. Biz ilk tanımda döngü değişkenini tanımlarız ve buna ilk değeri veririz. Örneğin burada `int i = 0` değerini verdik. Artık döngü değişkeninin adını da değerini de biliyoruz. Değeri 0 olan bir değişken üzerinde çalışıyorsak bu değişkenin bir değere ulaşması bizim ilk tercihimiz olmalıdır. O halde **i <= 20** diyoruz. Şart yapısındaki ilişki operatörleri 1 sonucunu verdikçe bizim döngümüz dönmeye devam edecektir. Bu döngünün sonsuza kadar dönmemesi için şartın bir şekilde bozulması gereklidir değil mi?, o halde biz şartı bir süre sonra bozma adına i değişkenini `i++` diyerek her döngü döndükçe birer birer artırıyoruz. Sonuçta önceden hesapladığımız gibi döngümüz 21 kere dönüyor ve döngü değişkeni her döngü dönünce ekrana değerini yazdırıyor. Sonrasında 21 olunca ise program döngüden çıkıyor ve akışına devam ediyor.

Biz şimdi programı sonsuz döngüye sokarak biraz eğlenelim. Bunun için döngü şartına devam etmesi için gereken rakamı yani 0'dan farklı bir rakamı

yazmamız yeterlidir. Bu ilişki operatörleri olmadan döngünün nasıl çalıştığını görmeniz adına güzel bir örnek olacaktır.

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

    for (int i = 0; 20; i++)
    {
        printf("%i ", i);
    }
    printf("\n");
    system("PAUSE");
} // Main Sonu
```

Biz burada for (int i = 0; 20; i++) diyerek for döngüsünün şart kısmına 20 yazdık. Mantıksal olarak 0'dan farklı her değer DOĞRU anlamında olduğunu düşünürsek bu döngüyü durdurmamız mümkün değildir. Programı çalıştırdığımızda ise program ileriye doğru bütün hızıyla saymaya başlayacaktır. Bilgisayarı biraz daha açık bırakırsanız milyarları bulduğunu görebilirsiniz.

```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\... - [X]
120513 120514 120515 120516 120517 120518 120519 120520 120521 120522 120523 12
0524 120525 120526 120527 120528 120529 120530 120531 120532 120533 120534 12053
5 120536 120537 120538 120539 120540 120541 120542 120543 120544 120545 120546 1
20547 120548 120549 120550 120551 120552 120553 120554 120555 120556 120557 1205
58 120559 120560 120561 120562 120563 120564 120565 120566 120567 120568 120569
120570 120571 120572 120573 120574 120575 120576 120577 120578 120579 120580 120
581 120582 120583 120584 120585 120586 120587 120588 120589 120590 120591 120592
120593 120594 120595 120596 120597 120598 120599 120600 120601 120602 120603 12
0604 120605 120606 120607 120608 120609 120610 120611 120612 120613 120614 12061
5 120616 120617 120618 120619 120620 120621 120622 120623 120624 120625 120626 1
20627 120628 120629 120630 120631 120632 120633 120634 120635 120636 120637 1206
38 120639 120640 120641 120642 120643 120644 120645 120646 120647 120648 120649
120650 120651 120652 120653 120654 120655 120656 120657 120658 120659 120660 120
661 120662 120663 120664 120665 120666 120667 120668 120669 120670 120671 120672
120673 120674 120675 120676 120677 120678 120679 120680 120681 120682 120683 12
0684 120685 120686 120687 120688 120689 120690 120691 120692 120693 120694 12069
5 120696 120697 120698 120699 120700 120701 120702 120703 120704 120705 120706 1
20707 120708 120709 120710 120711 120712 120713 120714 120715 120716 120717 1207
18 120719 120720 120721 120722 120723 120724 120725 120726 120727 120728 120729
120730 120731 120732 120733 120734 120735 120736 120737 120738 120739 120740 120
741 120742 120743 120744 120745 120746 120747 120748 120749 120750 120751 120752
120753 120754 120755 120756 120757 120758 120759 120760 120761 120762 120763 12
0764 120765 120766 120767 120768 120769 120770 120771 120772 120773 120774 12077
5 120776 120777 120778 120779 120780 120781 120782 120783 120784 120785 120786 1
20787 120788 120789
```

Böyle kontrolden çıkmış bir programı yazmayı kimse istemez. O yüzden döngü şartlarına dikkat etmemiz gereklidir. Şimdi ise döngü şartını değişken değeri ile elde edelim. Döngüden çıkmak için 0 değerinin elde edilmesi gerektiğini biliyoruz. Bunun için ilişki operatörlerine çok daha ihtiyaç yoktur. Önemli olan döngü değişkeninin bir şekilde 0 olması gereklidir. Şimdi bunu ilişki operatörleri olmadan yapalım.

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

    for (int i = 20; i; i--)
    {

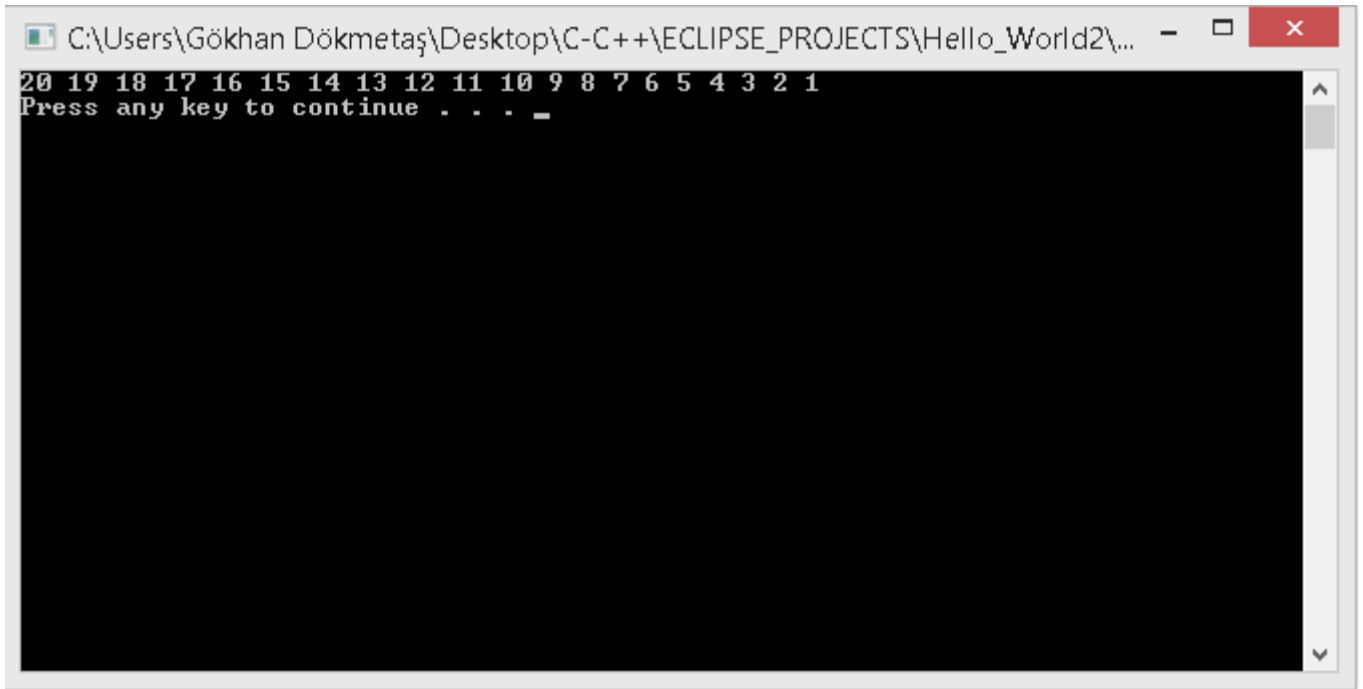
        printf("%i ", i);

    }

    printf("\n");
    system("PAUSE");
}
```

```
} // Main Sonu
```

Burada programa dokunmadık ama sadece tek satırlık döngü yapısını yani for (int i = 20; i; i–) komutunda değişiklik yaptık. Ne kadar literatürde artırım dense de i++ yazmamız şart değildir. Burada şimdi biz i– yazdık ve i değişkenine ilk değeri 20 olarak verdik. Programda döngü dönecek fakat bir noktaya kadar bu sürecektir. Bu nokta da i değerinin 0 olmasıdır. Bunu sağlamak için her döngü başında i– ifadesiyle i değerini bir azaltıyoruz. Program her döngüde printf(“%i”, i); komutu ile döngü değişkenini yazdıracaktır. Şimdi programın nasıl çalıştığına bir bakalım.

A screenshot of a Windows command prompt window. The title bar shows the file path: C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\... The window contains the output of a C++ program. The first line shows the numbers 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 in a single line. The second line says "Press any key to continue . . . _". The rest of the window is black, indicating the program has finished execution and is waiting for a key press.

Görüldüğü gibi aslında ilişki operatörlerine de gerek yokmuş. Ama ilişki operatörlerini kullanmak mantıksal işlemleri ve günlük hayattaki olayları programa aktarmada işimizi kolaylaştırmaktadır. Assembly dilinde programlama yaparken döngülerde kullandığımız dallanma komutlarını denetleyen ALU durum bitleri bulunur. Döngüler bu durum bitlerine göre çalışmaktadır. En sık kullanılan durum biti de sıfır değeri bayrağıdır. Yani makine komutlarıyla bir program yazdığımızda sıfır elde edilene kadar döngü

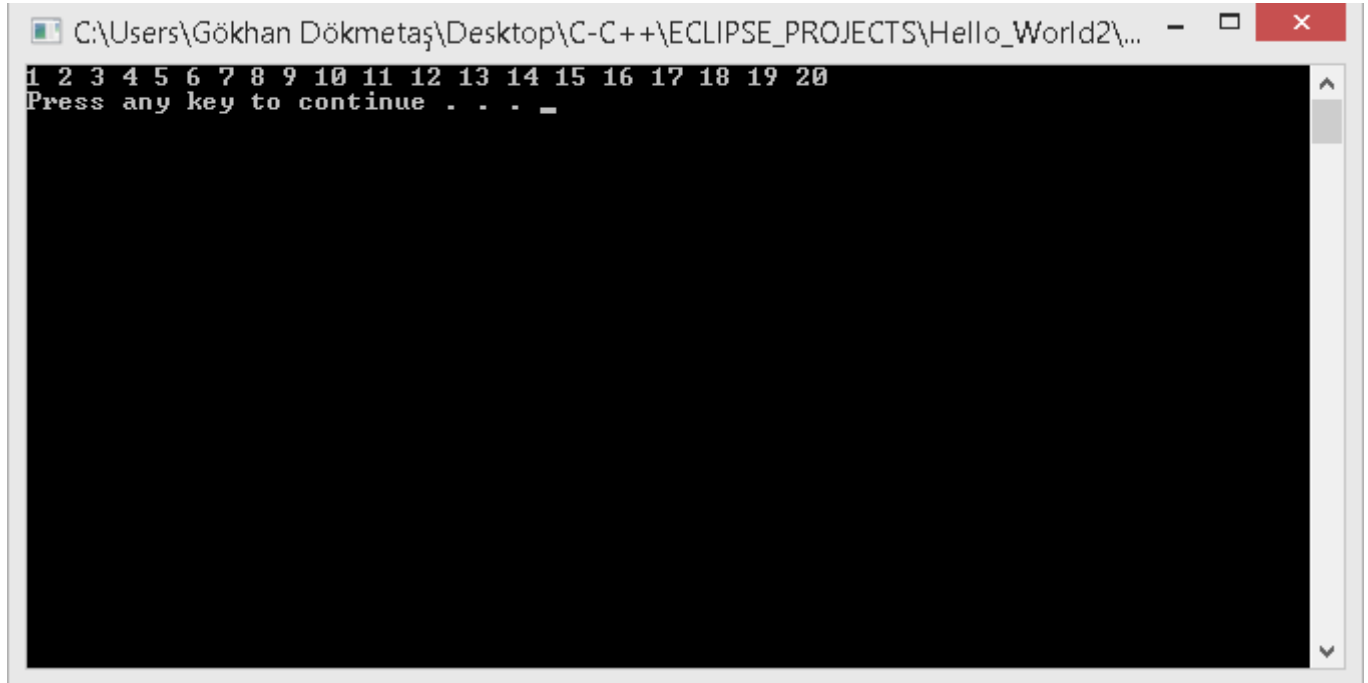
çalışsın şeklinde bir komutu kullanırız. Burada da buna benzer bir kullanımda bulunduk. Fakat yüksek seviye programlamada bu ilişki operatörleri bizim program akışını daha iyi anlayabilmemizi sağlar. Biz burada for döngüsünün esnekliğini ve döngü mantığını anlatmak için böyle kod yazıyoruz.

Şimdi ise döngüyü biraz daha esnetelim ve döngü yapısını programın farklı yerlerine taşıyalım.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i = 1;
    for ( ; i <= 20 ; )
    {
        printf("%i ", i);
        i++;
    }
    printf("\n");
    system("PAUSE");
} // Main Sonu
```

Görüldüğü gibi burada nasrettin hoca ve leylek fıkrasında olduğu gibi ortada for döngüsüne benzer bir şey kalmadı. Ama bu halde bir sonraki anlatacağımız döngüye oldukça benzedi. Artık for döngüsü sadece i değerinin 20'den küçük veya 20'ye eşit olup olmadığına bakıyor ve aradığı şartı sağlıyorsa döngüyü çalıştırmaya devam ediyor. i değişkenini de yukarıda yani döngüye girmeden önce main fonksiyonu içerisinde tanımladık. Bu i'yi artırma

işlemini de döngünün içerisinde yaptık. Sonuçta aynı işi yapan bir programı yazmış olduk.



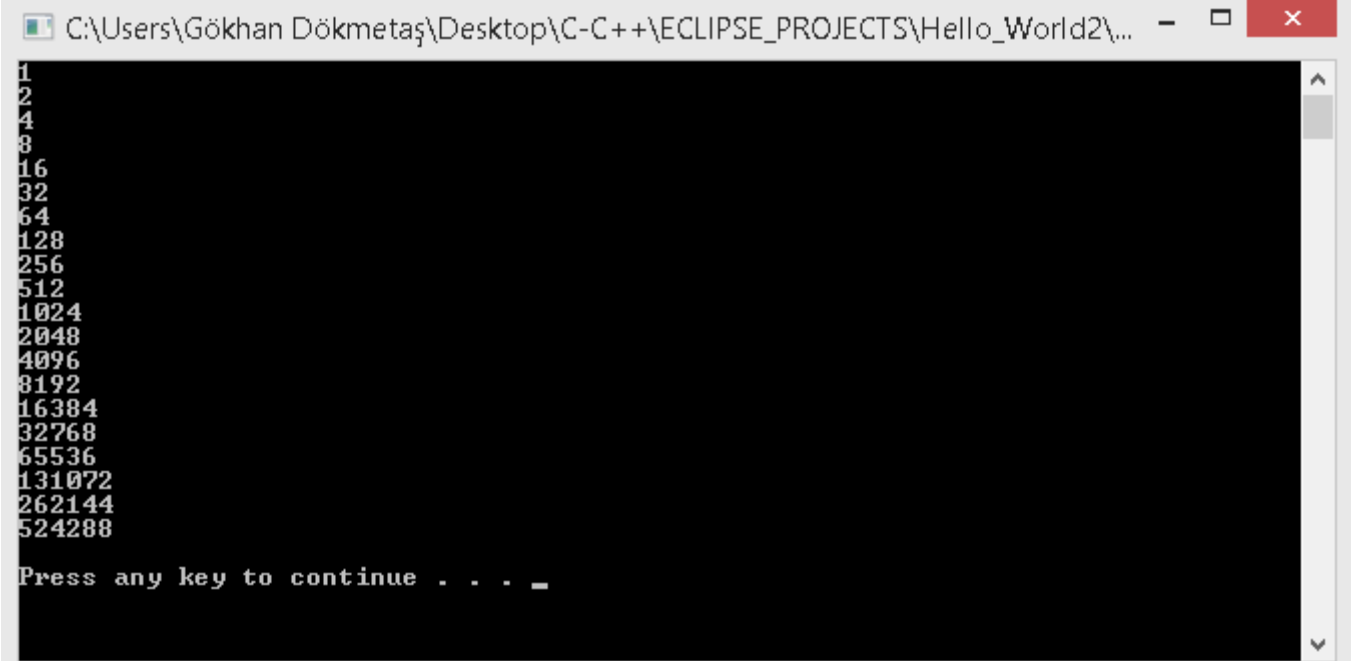
```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20  
Press any key to continue . . . _
```

Genelde eğitimlerde for döngüsü belli bir sayıda komutu çalıştırmaya yönelik döngü olarak anlatılsa da aslında sadece bir döngüdür. Farklı diğer döngülerden daha derli toplu olmasıdır ve belli bir amaca yönelik kullanılmasıdır. Bu amaç da döngü blokunu belli bir sayıda çalıştırmak üzeredir. Fakat bu alışkanlığa bağlı bir durumdur. for döngüsünde artırma ya da azaltma işleminin birer birer yapıldığını gördünüz. Aslında bu nokta ne artırma ne de azaltma noktasıdır. İstediğiniz karmaşıklıkta matematiksel işlemi yapabilirsiniz.

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

```
int main() {  
  
    for (int i = 1; i <= 1000000; i = i + i )  
    {  
  
        printf("%i \n", i);  
  
    }  
  
    printf("\n");  
    system("PAUSE");  
} // Main Sonu
```

Bu program 1 milyona kadar ikinin katlarını döngü değişkeninde hesaplar ve bize bildirir. Programın çıktısı şu şekildedir.



```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...  
1  
2  
4  
8  
16  
32  
64  
128  
256  
512  
1024  
2048  
4096  
8192  
16384  
32768  
65536  
131072  
262144  
524288  
Press any key to continue . . . _
```

Siz de artık for döngüsü üzerinde oynamalar yaparak pek çok şeyi yazdırabilirsiniz. Görüldüğü gibi for döngüsü oldukça esnek ve pek çok işlemi döngüler sayesinde zaman kaybetmeden yapabiliyoruz. Döngülerin kullanım alanına dair daha işe yarar bir örnek olarak sizlere faktöriyel hesaplama

programını gösterebiliriz. Bu program kullanıcının girdiği sayının faktöriyelini for döngüsüyle hesaplamakta ve ekrana yazdırmaktadır.

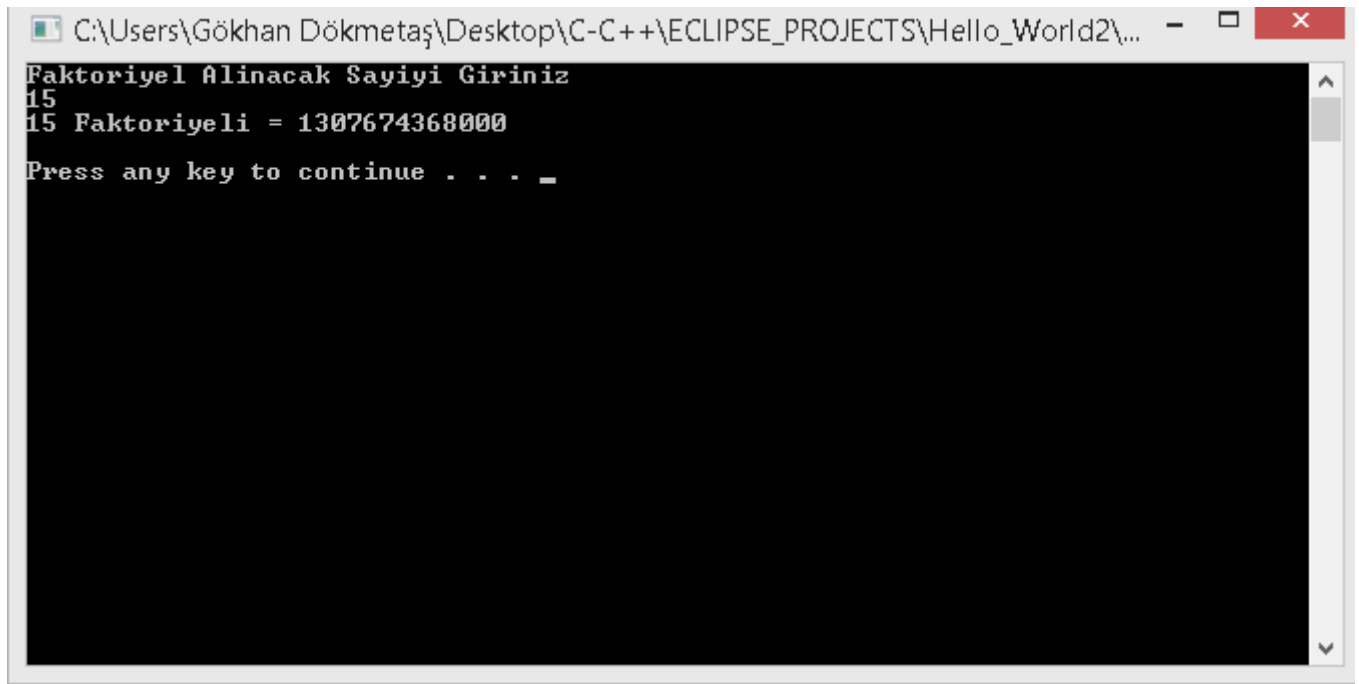
```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int c, n;
    long long faktoriyel = 1;

    printf("Faktoriyel Alinacak Sayiyi Giriniz\n");
    scanf("%d", &n);

    for (c = 1; c <= n; c++)
        faktoriyel = faktoriyel * c;

    printf("%d Faktoriyeli = %I64d \n", n, faktoriyel);
    printf("\n");
    system("PAUSE");
} // Main Sonu
```

Bu program gerçekten işe yarar bir programdır çünkü faktöriyel değerini long long olarak tanımladık. Bilindiği üzere 20'nin faktöriyeli bile kentrilyonları geçtiği için çok uzun değerleri taşıyabilecek bir değişken lazımdır. Diğer faktöriyel hesaplama programlarında int olarak tanımlanmakta ve hemen taşma yapmaktadır. Şimdi programın çıktısını verelim ve sonrasında bu hesaplamayı nasıl yaptığını inceleyelim.

A screenshot of a Windows console window titled "C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...". The console has a black background with white text. It displays the prompt "Faktoriyel Alinacak Sayiyi Giriniz", the input "15", and the output "15 Faktoriyeli = 1307674368000". At the bottom, it says "Press any key to continue . . . _".

```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...
Faktoriyel Alinacak Sayiyi Giriniz
15
15 Faktoriyeli = 1307674368000
Press any key to continue . . . _
```

Görüldüğü gibi faktöriyelimiz hesaplanıyor ve ekrana yazdırılıyor. Şimdi fonksiyonda önemli olan hesaplama kısmını inceleyelim.

```
for (c = 1; c <= n; c++)
    faktoriyel = faktoriyel * c;
```

Görüldüğü gibi bir for döngüsünden ibaret olan fonksiyon bütün işi yapmakta. Öncelikle c değeri 1'e eşitlenmekte ve c değeri n değerinden küçük ya da n değerine eşit oldukça birer birer adım döngü komutu yürütülmektedir. Döngü komutu ise faktoriyel değerine sürekli faktöriyel ve c'nin çarpımlarını ekler. Burada c değerlerini ve faktöriyel değerini daha iyi görüp anlamamız için bu değerleri ekrana yazdırarak komutu düzenleyelim.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int c, n;
    long long faktoriyel = 1;
```



```
printf("Faktoriyel Alinacak Sayiyi Giriniz\n");
scanf("%d", &n);

for (c = 1; c <= n; c++)
{
    printf("\n Faktoriyel: %I64d \n C: %i \n",
faktoriyel, c);
    faktoriyel = faktoriyel * c;
}

printf("\n %d Faktoriyeli = %I64d \n", n, faktoriyel);
printf("\n");
system("PAUSE");
} // Main Sonu
```

Şimdi bütün program aşamalarını ekranda görebiliriz.

```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\... - □ ×
Faktoriyel Alinacak Sayiyi Giriniz
10

Faktoriyel: 1
C: 1

Faktoriyel: 1
C: 2

Faktoriyel: 2
C: 3

Faktoriyel: 6
C: 4

Faktoriyel: 24
C: 5

Faktoriyel: 120
C: 6

Faktoriyel: 720
C: 7

Faktoriyel: 5040
C: 8

Faktoriyel: 40320
C: 9

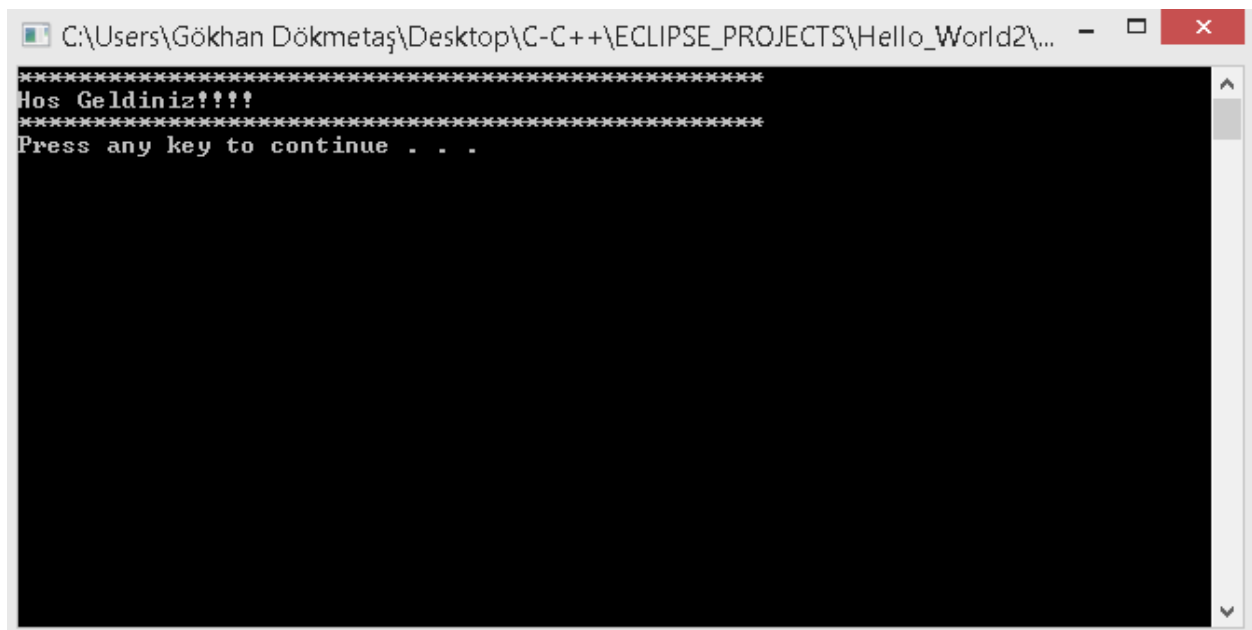
Faktoriyel: 362880
C: 10

10 Faktoriyeli = 3628800
Press any key to continue . . . _
```

Görüldüğü gibi faktoriyel değeri her defasında C değeri ile çarpılıyor ve katlanarak devam ediyor. $\text{faktoriyel} = \text{faktoriyel} * c$; komutu faktoriyel değerini c ile çarpıp bunu faktoriyel içerisine tekrar atmaktadır. Görüldüğü gibi döngüler olmadan bizim girdiğimiz değerin kaç defa çarpıma uğrayacağı bilinemezdi. O yüzden döngüler ve şartlar programlamanın olmazsa olmaz parçalarındandır. Şimdi for döngüsünü bu matematik işlemleri ile değil basit işleri yapmak için kullanalım. Bir konsol ekranına “Hoş Geldiniz” mesajı yazmak istiyoruz fakat bunu süslemeye ihtiyacımız var. Bunun için şöyle bir program yazalım.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    for (int i = 0; i < 50; i++)
    {
        printf("*");
    }
    puts("\nHos Geldiniz!!!!");
    for (int i = 0; i < 50; i++)
    {
        printf("*");
    }
    printf("\n");
    system("PAUSE");
} // Main Sonu
```

Şimdi programın nasıl bir çıkış verdiğine bakalım.



```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...
*****
Hos Geldiniz!!!!
*****
Press any key to continue . . .
```

Burada yeni bir fonksiyon kullandığımızı görebilirsiniz. O da puts(“\nHos Geldiniz!!!!”); komutunda gördüğünüz üzere puts() fonksiyonudur. Bu aslında printf fonksiyonuna benzer bir fonksiyon olup “Put String” yani karakter dizisi koy anlamına gelmektedir. printf fonksiyonuna göre karakter dizisi yazdırmada daha güvenli olduğu söylenen bu fonksiyonu kullanarak da karakter dizisi yazdırabilirsiniz. Bu fonksiyon printf’den farklı olarak satır atlama ifadesini otomatik olarak koyar. Bu fonksiyonu kullanmaya alışmanızı tavsiye ediyoruz.

for döngüsünü kullanırken genellikle yeni başlayan programcılar döngü şartını yanlış yazmaktan dolayı döngüyü bir eksik veya bir fazla çalıştırırlar. for döngüsünü kullanırken bunun farkında olmanız lazımdır. Örneğin şöyle bir ifade 10 kere değil 9 kere çalışacaktır.

```
for (int i = 1; i < 10; i++)  
{  
    komutlar;  
}
```

Bunu 10 kere çalıştırmak için küçük eşittir ifadesini kullanmanız gereklidir. Döngünün şart kısmına $i \leq 10$ yazıldığı zaman 1’den 10’a kadar ve 10 dahil anlamı taşımaktadır.

for döngüsü hakkında yeterli anlatımı ve örneği verdiğimizize inanıyoruz. Bir sonraki yazıda while döngüsünü anlatarak döngüler konusuna devam edeceğiz.

-17- While Döngüsü

Daha önceki başlıkta for döngüsünü anlatmıştık. for döngüsünün en önemli yanı for döngüsündeki komutlar belli bir sayıda çalıştırılmaktadır. Ne kadar

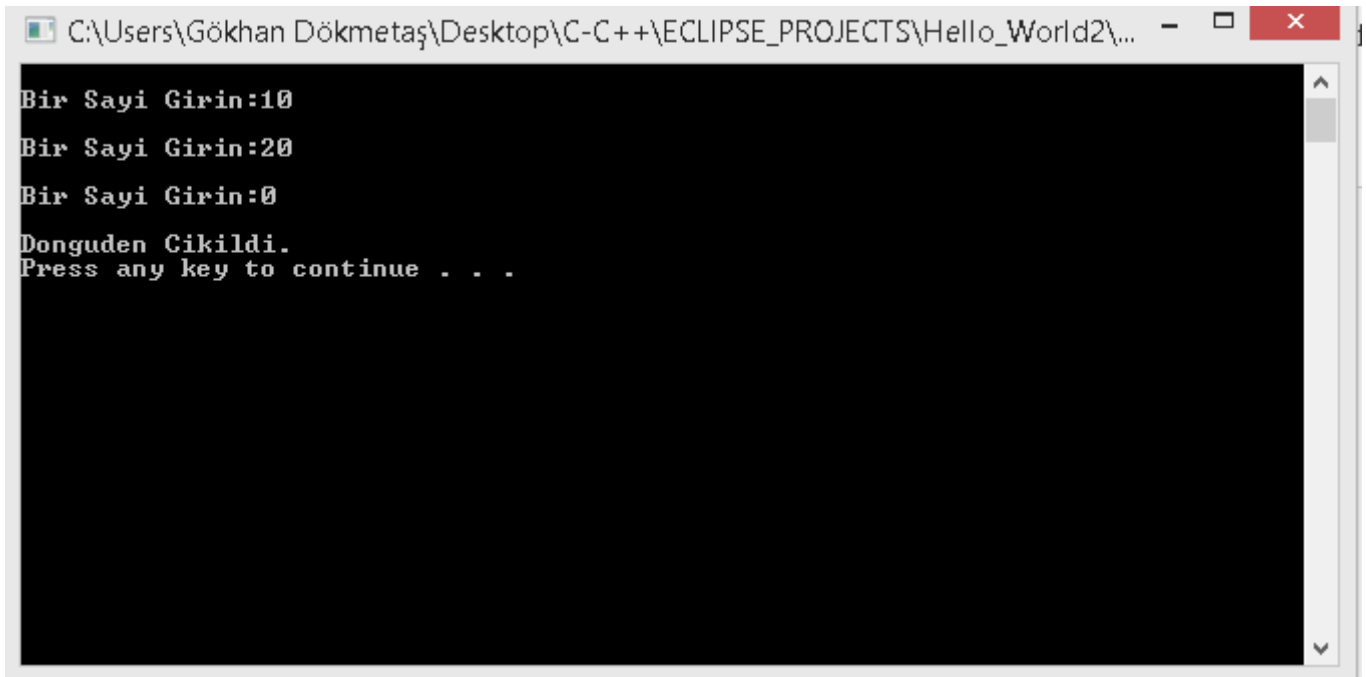
sayıda olacağı ise for döngü yapısı içerisinde belirtilmiştir. for döngüsünü örneklerde esnetip bu amacın dışında kullansak da for döngüsünün gerçek amacı bütün döngü elemanlarını tek bir satırda toplamak ve belli bir sayıda çalıştırmaya yönelik bir döngü hazırlamaktır.

while döngüsüne baktığımızda ise while döngüsünün sadece döngü şartından oluştuğunu ve bu döngü şartının da program içerisinde belirsiz bir zamanda sağlandığını görürüz. Yani while döngüsünde döngünün ne kadar çevirim boyunca çalışacağından haberimiz yoktur. Döngü şartı sağlanana kadar döngü devamlı olarak çalışır. Şimdi bununla alakalı bir örnek yaptığımızda ne demek istediğimizi daha iyi anlayacaksınız.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int c = 1;
while ( c != 0 )
{
printf("\nBir Sayi Girin:");
scanf("%i", &c);
}

printf("\nDonguden Cikildi. ");
printf("\n");
system("PAUSE");
} // Main Sonu
```

Bu program öncelikle c adında bir döngü değişkeni tanımlar ve döngü değişkeninin sıfır olmadığından emin olmak için ona ilk değeri verir. Eğer döngü değişkeni sıfır değerinde olursa döngü baştan şartı sağladığı için hiç çalışmayacaktır. Döngü değişkenlerinin ilk değerini takip etmenin önemini burada görebiliriz. Sonrasında kullanıcıdan bir sayı girmesini ister ve klavyeden girilen sayıyı döngü değişkenine aktarır. Eğer sayı döngü şartını sağlamazsa bir daha sayı girilmesini ister ve bu sürekli devam eder. Burada dikkat edilmesi gereken nokta programcının kullanıcının kaç denemeden sonra doğru sayıyı gireceğini bilmemesidir. Elde sadece belirli bir şart olduğu için döngü tamamen bu şarta bağlıdır. Eğer klavyeden 0 değerini girersek programdan çıkacaktır.



```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\... - [X]
Bir Sayi Girin:10
Bir Sayi Girin:20
Bir Sayi Girin:0
Donguden Cikildi.
Press any key to continue . . .
```

Bizim burada incelememiz gereken önemli bir komut olarak while (c != 0) komutu vardır. Şimdi bu döngüler için neden for ya da while adlarının kullanıldığını düşünebilirsiniz. İngiliz dilinde for kelimesi belli bir süre için kullanılır. Örneğin “for 30 years” dediğimizde 30 yıl boyunca anlamına gelir. for döngüsünde de for anahtar kelimesinin ardından belli bir miktar belirtildiği için burada İngiliz dili mantığına uygun for kelimesi kullanılmıştır. Yani bunu

Türkçe’de “İçin” kelimesi olarak düşünmemiz doğru olmaz. “While” kelimesi de “iken” anlamı taşımaktadır. Yani `while (c != 0)` dediğimizde `c != 0` iken döngüyü çalıştır anlamına gelmektedir. Tam dile çevirisi ise `c` değeri 0’a eşit değil iken döngüyü çalıştır olmaktadır. `while` döngüsünün söz dizimi şöyle olmalıdır.

```
while(şart)
{
    komutlar;
    komutlar;
    komutlar;
}
```

```
while(şart)
tekkomut;
```

While döngüsünü ana dili İngilizce olanlar için cümlelerin başına `while...` kelimesini koyarak yalancı kod üretmemiz kolay olsa da `while` döngüsü için yalancı kodu Türkçe’de şu şekilde üretebiliriz.

Kapı kapalı iken kombiyi çalıştır. A sıfıra eşit değil iken a’yı birer birer azalt.

Görüldüğü gibi İngiliz dilinde önce `while` ifadesi sonra şart koyulması gerekirken bizde önce şart sonra iken ifadesi ve ardından talimatlar eklenmektedir. Buna dikkat ettikten sonra bu kalıpta yalancı kodları oluşturabilirsiniz.

`while` yapısını kullanırken en çok dikkat etmemiz gereken nokta programın sonsuz döngüye sokulmaması ve döngü bloku ile alakalı bir şartın döngü şartı

kısmına yazılmasıdır. Eğer döngü bloku içerisinde döngü şartını bozacak bir nokta yer almıyorsa program döngüden çıkamayacaktır. Ayrıca döngü şartı döngünün çalışma sebebi olduğu için döngü şartı ile alakasız bir değer eklemek döngünün işleyişini bozacaktır. O yüzden döngü şartı seçilirken iyi düşünülmesi gereklidir.

Şimdi while döngüsü ile alakalı biraz daha karmaşık bir program verelim. Fibonacci sayılarını veren programı for döngüsü ile yapabilirsek de bunu belirli bir sayı sınırına göre yapmamız gerekirdi. Burada ise kullanıcının girdiği sınıra kadar olan Fibonacci sayılarını while döngüsü ile göstereceğiz.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int t1 = 0, t2 = 1, sonraki_terim = 0, n;

    printf("Bir Sayi Giriniz: ");
    scanf("%d", &n);

    printf("Fibonacci Sayilari: %d, %d, ", t1, t2);

    sonraki_terim = t1 + t2;

    while(sonraki_terim <= n)
    {
        printf("%d, ", sonraki_terim);
        t1 = t2;
        t2 = sonraki_terim;
        sonraki_terim = t1 + t2;
    }
}
```



```
}  
puts("");  
system("PAUSE");  
} // Main Sonu
```

Fibonacci dizisi adını italyan matematikçi Leonardo Fibonacci'den almıştır. 0 ve 1 ile başlayan iki sayının kendinden önceki gelen iki sayı ile toplanması işlemidir. Bu basit bir matematik işlemi gibi görünse de bu işlemin altın oran ile bir alakası vardır ve matematikçilerin ilgisini çekmektedir.

```
int t1 = 0, t2 = 1, sonraki_terim = 0, n;
```

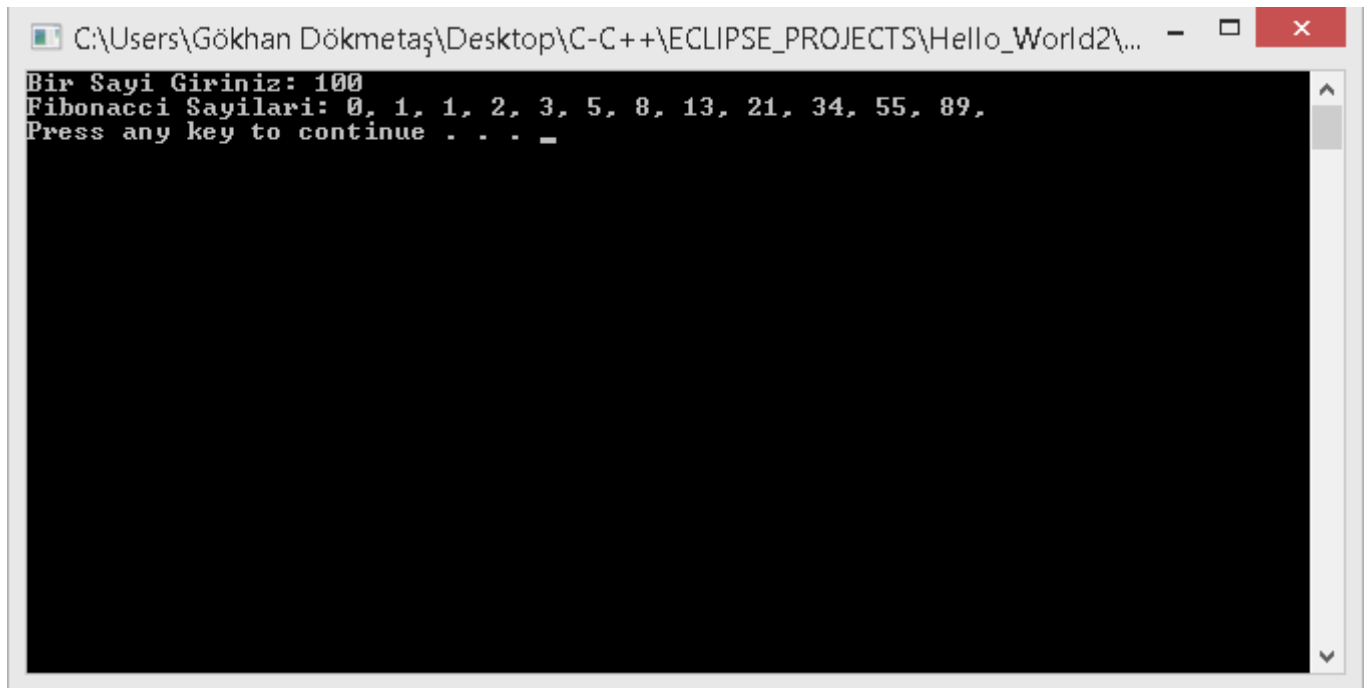
Biz burada gereken matematik işlemi için dört sayı yani dört değişken tanımladık. Bu değişkenler t1, t2 ve sonraki_terim olarak matematik işleminde kullanılacak. n değeri ise bizim ekranda girdiğimiz sınır değeri olacaktır. Eğer sonuç değeri sınır değerini aşarsa program akışı döngüden çıkacak ve programı sonlandıracaktır.

```
while(sonraki_terim <= n)
```

Burada bir ileride toplanacak değerin n sayısı ile karşılaştırılmasını görmekteyiz. Eğer o seferki sonuç değeri n değerinden küçük ya da n değerine eşitse döngü devam etmektedir. Büyük olduğu durumda ise döngüden çıkılmaktadır.

```
puts("");
```

Burada daha önce bahsettiğimiz puts fonksiyonunu farklı bir işlem için kullandık. Size puts fonksiyonunun otomatik olarak yeni satır ifadesi bıraktığını söylemiştik. O halde yeni satır komutu olarak puts("") fonksiyonunu kullanmak kısa ve etkili bir çözümdür.

A screenshot of a Windows console window titled "C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...". The console displays the following text: "Bir Sayi Giriniz: 100", "Fibonacci Sayilari: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,", and "Press any key to continue . . . _". The window has standard Windows controls (minimize, maximize, close) in the title bar.

Burada dikkat ederseniz program bir kere çalışıyor ve ardından kapanıyor. Biz programın sürekli çalışmasını ve bizim için hesap yapmasını istersek programı döngü içerisinde döngüye sokmamız gerekecektir. Bunun için programı şu şekilde düzenleyerek çalıştırabiliriz.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    while(1)
    {
        int t1 = 0, t2 = 1, sonraki_terim = 0, n;

        printf("Bir Sayi Giriniz: ");
        scanf("%d", &n);

        // displays the first two terms which is always 0 and 1
        printf("Fibonacci Sayilari: %d, %d, ", t1, t2);
```

```
sonraki_terim = t1 + t2;

while(sonraki_terim <= n)
{
    printf("%d, ",sonraki_terim);
    t1 = t2;
    t2 = sonraki_terim;
    sonraki_terim = t1 + t2;
}
puts("");
}
system("PAUSE");
} // Main Sonu
```

Bu programda sadece tek bir komut ekledik o da main'den sonraki while(1) komutudur. Bu döngü içerisinde 1 sayısı olduğundan döngü şartı sürekli sağlanmış olur ve program sonsuz döngüye sokulmuş olur. Biz programın sonsuz döngüye sokulmasını tavsiye etmesek de ana programda böyle bir istisna vardır. Eğer bütün komutları sürekli başa sararak çalıştırma ihtiyacı duyuyorsak böyle bir sonsuz döngü kullanabiliriz. Gördüğünüz gibi while(1) döngüsünün tarama alanı süslü parantezlerle belirtilmiştir ve bütün komutları kapsamaktadır. Artık programımız sürekli çalışan bir program haline gelmiştir.

```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\... - □ ×
Bir Sayi Giriniz: 100
Fibonacci Sayilari: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,
Bir Sayi Giriniz: 200
Fibonacci Sayilari: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
Bir Sayi Giriniz: 300
Fibonacci Sayilari: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,
Bir Sayi Giriniz: 500
Fibonacci Sayilari: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
Bir Sayi Giriniz: 0
Fibonacci Sayilari: 0, 1,
Bir Sayi Giriniz:
```

Bu döngüler hakkında daha fazla örneği ilerleyen konularda sizlere anlatacağız. Anladığınızı varsayarak biraz da sizi aptal yerine koymadığım için çok da fazla giriş seviyesinde anlatmıyorum. Her kodu sürekli satır satır açıklamıyorum ve anlamanızı bekliyorum. Eğer yazının anlaşılabilirliğinde bir sıkıntı varsa lütfen yorumlarda belirtiniz ilerleyen dersleri daha basit seviyede anlatmaya çalışırım.

-18- do while Döngüsü

Döngüler konusunda en son ele alacağımız yapı do...while ya da kısaca do döngüsüdür. Bu döngü while döngüsünün birebir aynısı olsa da bir yönüyle farklı olmaktadır. while döngüsü döngü şartını üst kısma almışken do döngüsü döngü şartını en alta alır ve döngü şartı sağlansın ya da sağlanmasın döngü blokundaki komutlar en az bir kere işletilir. do bildiğiniz üzere “Yap” anlamı taşımaktadır. İngiliz dil mantığına göre YAP şunu, şunu İKEN şu şartlar altında şeklinde bir dizilimi vardır. Bunu çok kolay ingilizce cümleye çevirebiliriz “Do some work while all conditions are met” dediğimizde bütün şartlar sağlandığında bazı görevleri yap anlamı çıkmaktadır. Burada dikkatinizi tekrar çekmek istediğim nokta bu yapının İngiliz dil mantığına nasıl uygun olarak tasarlandığıdır. Bunu biz anlamak istiyorsak bu mantığı hesaba katmamız gereklidir. Yoksa başka türlü anlamak biraz zor olacaktır.

Bunu Türk dil mantığına çevirirsek “Şu işi **yap**, şartlar sağlandığında bir daha yap.” Burada işaretlediğim kısımlar programın anahtar kelimeleridir. while kelimesi “-dığına” ekinin yerine de geçmektedir. Programlama dillerinde dilbilimini kullanmakla gerçek dil ve programlama dili arasındaki geçişleri daha rahat yaparız. Bu da gerçek hayattaki olayları programa aktarmamızda bize yardımcı olur.

Şimdi do..while döngüsüne dair bir örneği çalıştırıp inceleyelim ve ardından konumuza devam edelim.

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

```

char c;
printf("Bir Sayi Giriniz:");
do
{
    c = getchar();

}while(c != '0');
system("PAUSE");
} // Main Sonu

```

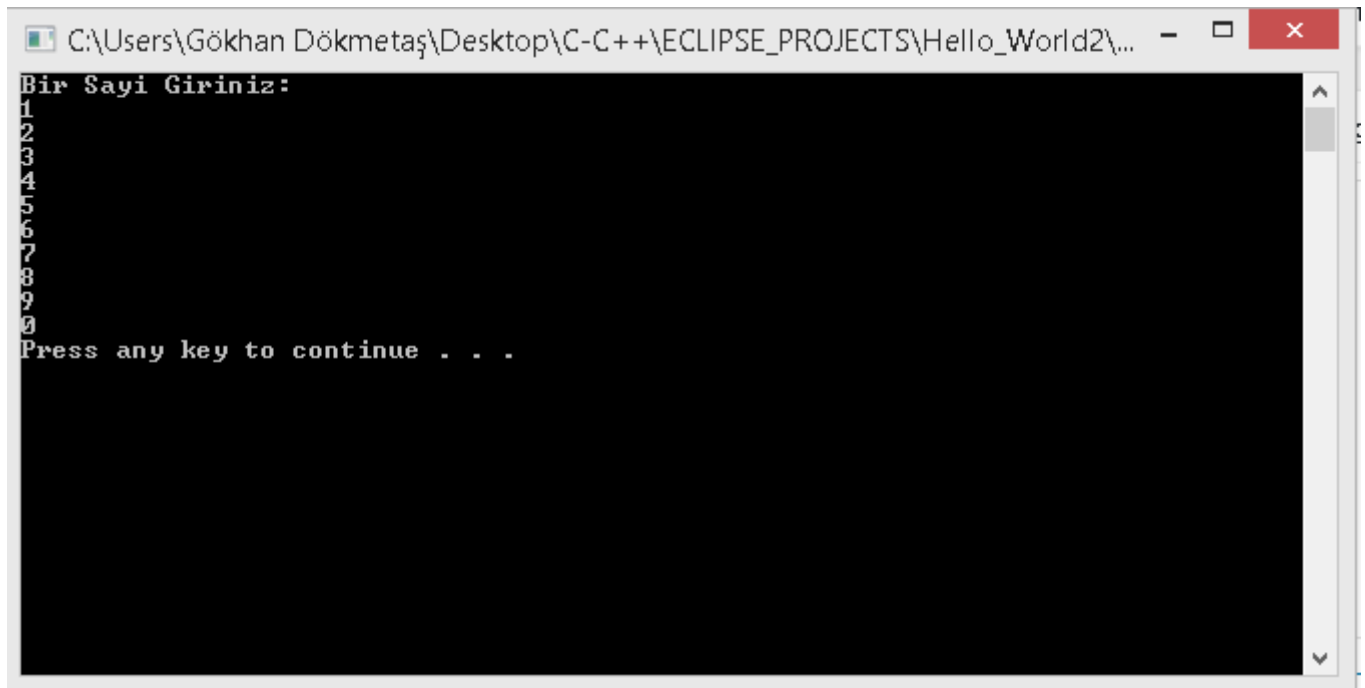
Burada döngü şartı sağlansın sağlanmasın bizden bir sayı girmemiz istenmekte ve girdiğimiz sayı döngü şartını bozarsa program döngüden çıkmaktadır. While döngüsünde döngü değişkeninin ilk seferde muhakkak döngü şartını sağlaması için değişkene ilk değeri elle veriyorduk. Burada ise böyle bir kullanıma ihtiyacımız kalmamıştır. do döngüsünün getirdiği en büyük özelliklerden biri budur. Şimdi do döngüsünün örnek söz dizimini yazalım.

```

do
{
    komutlar;
    komutlar;
    komutlar;
} while (şart) ;

// NOKTALI VİRGÜLE DİKKAT!!!

```



```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\... - □ ×
Bir Sayı Giriniz:
1
2
3
4
5
6
7
8
9
0
Press any key to continue . . .
```

Burada örnek bir program çıktısını görmekteyiz. Biz klavyede 0 harfini (değerini değil) yazmadıkça programdan çıkış yapmamaktadır. Bunu da şu komut ile sağlamaktayız.

while(c != '0'); Yani c değişkeni '0' işaretine (değerine değil) eşit olmadıkça programı döngüyü devam ettir demektir.

c = getchar();

Burada farklı bir komut kullanıyoruz. Biz normalde scanf ile klavyeden girilen değerleri okurduk. Fakat bu scanf'in argümanlarını ve formatlarını yazmak bizi uğraştırıyordu. getchar() fonksiyonu ile tek bir fonksiyonda tek karakteri okuyoruz ve klavyeden okunan karakter fonksiyon tarafından geri döndürülüyor. Bunu c değişkenine atayarak aynı işi yapabiliriz. Dikkat etmeniz gereken nokta getchar() fonksiyonunun sadece karakter formatında tek bir karakter okuyabildiğidir. O yüzden tek bir tuşu okurken bunu kullanmanız gereklidir. Ayrıca **putchar(c)** dersek bu sefer c değişkenindeki karakter printf fonksiyonunda olduğu gibi karakter formatında ekrana yazdırılmaktadır.

Bunlar stdio.h kütüphanesinin fonksiyonları olduğu için ilave bir kütüphane eklemeye gerek yoktur.

Hangi döngünün ne zaman kullanılmasına gelince bunu kesin sınırları yoktur diyebiliriz. Programcılıkta tecrübe edindikçe doğru döngüyü doğru yerde kullanmayı zamanla öğreneceksiniz. Genel olarak for döngüsü bir kod blokunu belli bir sayıda işletmek için, while döngüsü önce şarta bağlı olarak program blokunu belirsiz sayıda işletmek için, do döngüsü de şart fark etmeksizin en az bir kere işletmek için kullanılır.

Döngü konusunu bitirmiş olduk ve sonraki yazımızda kararlara giriş yapacağız.

-19- Kararlar ve If Karar Yapısı

Döngülerin ardından programlamaya dair en temel bileşenlerden biri de karar yapılarıdır. Karar yapıları sınıflar gibi üst seviye bir yazılım özelliği değildir. Eski veya yeni bütün programlama dillerinde istisnasız olarak karşımıza çıkmaktadır. Üstelik bu özellik sadece programlama diline mahsus da değildir. Makine komutları arasında karar komutlarını görmemiz mümkündür. Yapay zekanın temeli kararlar ve döngülerdir. Eğer karar mekanizması olmasaydı yazdığımız programlar eksik olacak ve program yazmanın pek fazla bir anlamı kalmayacaktı. Biz günlük hayatta da pek çok karar durumuyla karşılaşırız ve aklımızı kullanarak bu kararları veririz. Makinenin karar vermesi ise bizim talimatımızla olmaktadır. Bu noktada programcı program yazarken aklını kullanmakta ve ona göre bir karar mekanizması oluşturmaktadır. Karar mekanizmaları insan dilinde de dil yapısı içerisinde ve sıkça kullanılır. Örneğin İngilizce’de “If” ifadesiyle bir karar cümlesi ortaya koyulur. “If the door is open, close the door.” yani “Eğer kapı açıksa kapat” yapısını dilde kolaylıkla oluştururuz. Türk dilinde karar yapıları “Eğer” ile veya -sA eki ile sağlanmaktadır. Bizim -sA harfini A harfini büyük halde yazmamız dil bilgisi kitaplarında böyle görmemizden dolayıdır. Yani -sA yazdığımızda bunun hem -se hem -sa olarak okunabileceğini belirtmiş oluruz. Şimdi konuşma dilindeki karar yapıları ile yalancı kod yazalım.

- A Sayısı 5’den büyükse 5’e eşitle.
- Zaman değeri 100 saniyeye eşit olursa ekrana “Zamanınız Doldu.” yazdır.
- Kullanıcı tuş takımındaki sol düğmesine basarsa robot kolu sola çevir.
- Eğer A değeri B’den büyükse ekrana “Büyüktür” yazdır değilse “Küçüktür yazdır.”

- Eğer A değeri B'den büyükse “Büyüktür”, değilse eğer eşitse “Eşittir”, değilse “Küçüktür” yazdır.
- İleride engel varsa sola dön.
- Batarya %10 olursa uyarı ver.

Görüldüğü gibi burada aynı döngülerde olduğu gibi bir şart ifadesi kullanılmakta fakat bu şart ifadesinin ardından tek bir komut işletilmektedir. Biz de karar verirken iki durum arasında bir seçim yaptığımız gibi iki duruma göre iki farklı işi yaptığımız da olmaktadır. Örneğin telefonun pil durumuna bakarız ve batarya %10 olursa şarja takarız. Fakat batarya %10'dan fazlaysa bir iş yapmayız. Burada bataryanın %10'dan fazla dolu olup olmaması gibi iki durumla karşılaşırız. Fakat telefonda oyun oynamak istediğimizde iki duruma göre iki farklı iş yaparız. Örneğin batarya %10'dan fazla dolu ise oyun oynarız, değilse şarja takarız. Görüldüğü gibi bu karar yapılarını günlük hayatımızda devamlı olarak kullanmaktayız.

Sırada ise bu karar yapılarını bilgisayar programında kullanmak vardır. Bilgisayarda karar yapıları parantez içerisindeki sonucun Doğru ya da Yanlış olmasına bakar. Aynı döngülerin arasına yazdığımız parantezlerde ilişkisel operatörler yardımı ile 1 ve 0 değerlerini elde ediyorsak karar yapıları da bu ilişkisel operatörler üzerine kurulmuştur. Daha karmaşık sonuçlar elde etmek için mantık operatörlerini de kullanabiliriz.

if karar yapısı

Karar yapılarına baktığımızda if ve türevlerinden başka bir de switch deyimini görmekteyiz. switch yerine de if-else-if yapısını kullanmak mümkün olduğundan karar yapılarını if'den ibaret görebiliriz. Bu karar yapılarını basit veya karmaşık kullanmamıza bağlı olarak yapının karmaşıklığı artacaktır. Bu

yapıları peşpeşe ve birbiri içinde kullanmamız mümkündür. Aynı döngü içinde döngü yapabildiğimiz gibi if içerisinde if de yapabiliriz.

Şimdi basit bir program yazalım. Öğrenci notu 45'e eşit veya 45'den yukarı olduğu zaman öğrencinin geçtiği bilgisini ekrana yazdıralım.

```
if ( not >= 45 ) {  
printf( "Gecti\n" );  
}
```

Görüldüğü gibi if deyiminin ardından parantez açıp ilişkisel operatörler yardımıyla Doğru veya Yanlış değerlerini elde ettik ve parantez içerisinde doğru olması şartıyla if blokunda yer alan yani süslü parantezlerin arasında olan komutlarımız çalıştı. If deyiminin içindeki kodun çalışması için parantez içindeki değerlerin 0'dan farklı bir sayı olması gereklidir. If içerisine bir rakam da yazabiliriz fakat bunu böyle yazmanın bir anlamı yoktur. Şimdi if karar yapısının söz dizimini yazalım.

```
if (şart){  
    komutlar;  
    komutlar;  
    komutlar;  
}
```

```
if (şart)  
{  
komutlar;  
komutlar;  
komutlar;  
}
```

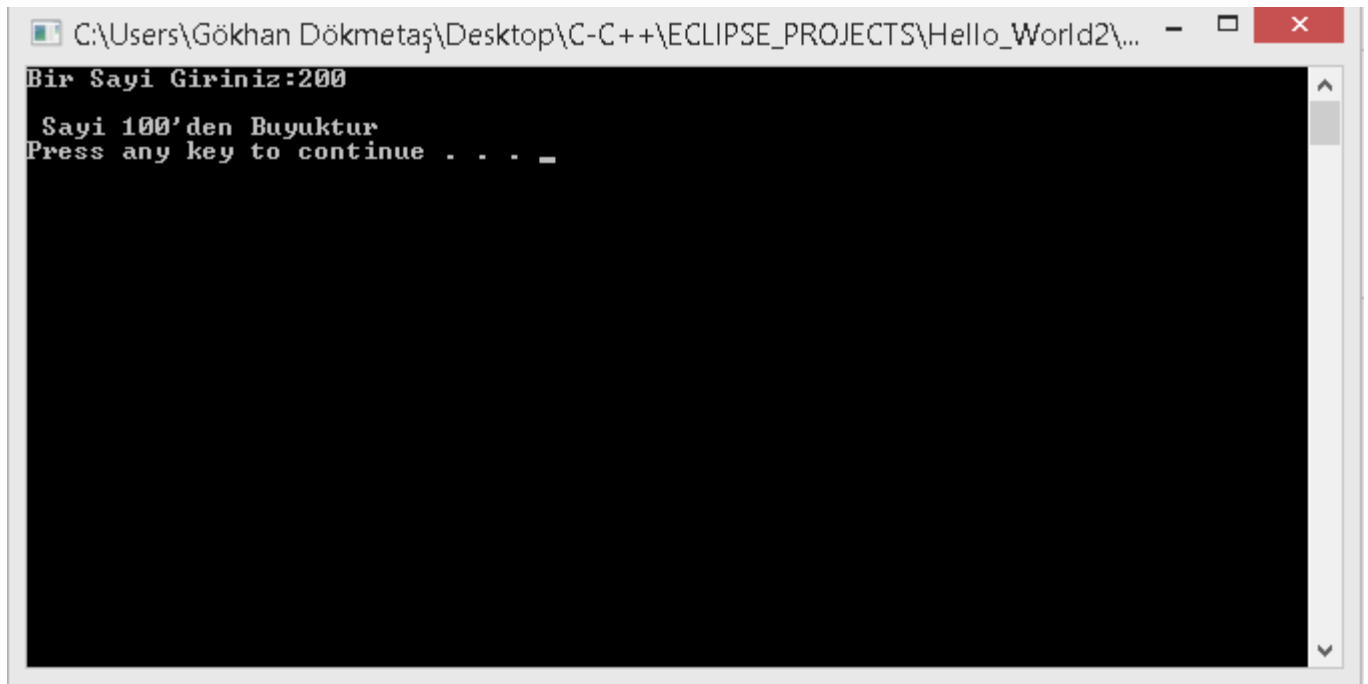
```
if (şart)
tek_komut;
```

Burada ilk kullanımda `şart` ifadesinin hemen ardından süslü parantezi açtık. Bunu aşağıda kullanmama sebebimiz `if` ifadesinde bu tarz bir kullanım alışkanlığı olduğu ve bu yönde kullanımın tavsiye edildiğidir. Alttaki ile aynı olsa da kodunuzu genel kabul görmüş kurallara göre yazmak istiyorsanız bu inceliklere dikkat etmeniz gerekir.

Şimdi `if` yapısını en basit halde kullanarak bir program yazalım ve deneyelim.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int sayi;
printf("Bir Sayi Giriniz:");
scanf("%i", &sayi);
if (sayi > 100){
    printf("\n Sayi 100'den Buyuktur \n");
}
system("PAUSE");
} // Main Sonu
```

Burada sayı 100'den büyükse ekrana "Sayı 100'den Buyuktur" ifadesi yazdırılacaktır. Eğer sayı 100'den büyük değilse bu komut işletilmeyecek ve `if` yapısı atlanacaktır. Sayının 100'den büyük olup olmadığını denetleyen karar mekanizması ise parantezlerin içerisinde `(sayi > 100)` olarak yer almaktadır. Buradaki şartı istediğimiz gibi değiştirip kendi karar yapımızı düzenleyebiliriz. Programın çıktısı şu şekilde olacaktır.



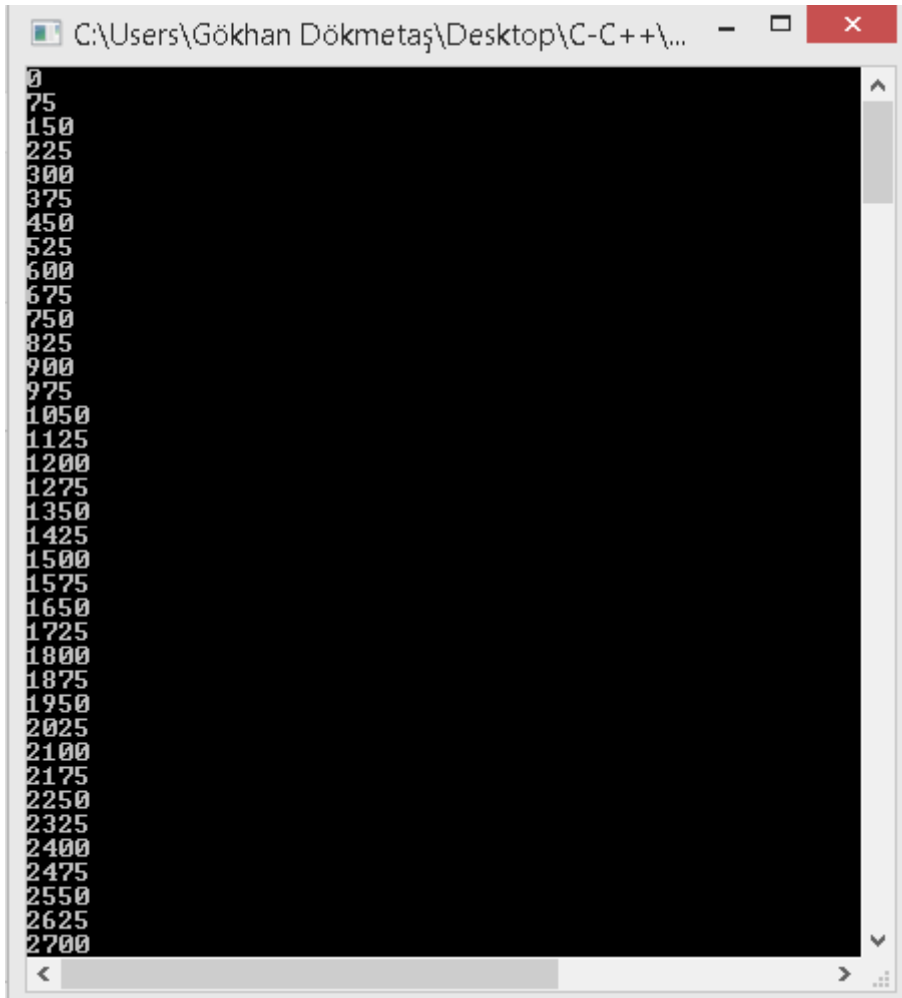
```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...
Bir Sayi Giriniz:200
Sayi 100'den Buyuktur
Press any key to continue . . . _
```

if yapısını döngülerin içerisinde kullanmakla etkili bir program yazmış oluruz. Örneğin 1 ile 10000 arasında 75'in katları olan sayıları bulmak için şöyle bir algoritma yazabiliriz.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int limit = 10000;
int i = 0;
while (i < limit)
{
if ((i%75) == 0){
printf("%i \n", i);
}
i++;
}
system("PAUSE");
```

```
} // Main Sonu
```

Burada limit adını verdiğimiz kaç'a kadar sayı olacaksa onun sınırını belirlemek amacıyla bir değişken tanımladık ve ona 10000 değerini yükledik. *i* değeri ise bizim döngü değişkenimiz olmaktadır. *i* değişkeni limit değerinden küçük olduğu sürece döngüye devam etme talimatı verdik. Döngü içerisinde *i* değeri her döngü çevriminde birer birer artırılabacaktır. Bunu *i++* komutu ile sağladık. Döngü içerisindeki *if* komutu ise her döngü çevriminde çalıştırılır. *if* ((*i*%75) == 0) ifadesinden göreceğiniz üzere *if*, *while* ve fonksiyonların içerisinde aritmetik işlem yapabiliriz. Biz de *i*%75 ile *i* değerinin 75'in katı olup olmadığını öğreniyoruz. Eğer 75'in katı ise bize sıfır değerini veriyor. Eğer sıfıra eşitse *printf* fonksiyonu ile sayımızı yazdırıyoruz. Programın ekran görüntüsü şu şekilde olacaktır.



The screenshot shows a Windows command prompt window with the title bar "C:\Users\Gökhan Dökmetaş\Desktop\C-C++\...". The window contains a list of numbers printed in a monospaced font, starting from 0 and increasing by 75 up to 2700. The numbers are: 0, 75, 150, 225, 300, 375, 450, 525, 600, 675, 750, 825, 900, 975, 1050, 1125, 1200, 1275, 1350, 1425, 1500, 1575, 1650, 1725, 1800, 1875, 1950, 2025, 2100, 2175, 2250, 2325, 2400, 2475, 2550, 2625, 2700.

Ekranın tamamını buraya alamasak da burada 10 bine kadar 75'in katlarını görebiliriz. Bir sonraki örnekte ise birden fazla if ifadesinin kullanımının nasıl olduğunu görelim.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int sayi;
printf("Bir Sayi Giriniz:");
scanf("%i", &sayi);
if (sayi > 100){
    printf("\n Sayi 100'den Buyuktur \n");
}
if (sayi < 100){
    printf("\n Sayi 100'den Kucuktur \n");
}

if (sayi == 100){
    printf("\n Sayi 100'e Esittir \n");
}

system("PAUSE");
} // Main Sonu
```

Burada yukarıdaki programın benzerini görmekteyiz. Fakat yukarıdaki program sadece sayının 100'den büyük olup olmadığını kontrol ediyordu. Burada ise sayının tüm ihmal durumlarını hesaba kattık ve programı bu şekilde yazdık.

Bizim anlattığımız for, while, if gibi deyimler başlangıç seviyesi olmayıp en ileri seviye programcılar ile programlarında etkin olarak bunları kullanmaktadır.

Yani ileride size lazım olmayacak bilgileri asla size anlatmıyoruz. Bir sonraki yazımızda karar yapılarına devam edeceğiz.

-20- Else ve Else If Yapıları

Önceki yazıda karar yapılarını anlatırken sadece basit bir if karar yapısını sizlere anlattık. If karar yapısında şart sağlanırsa if blokundaki komutlar işletiliyordu. Şart sağlanmadığında herhangi bir komutun işletilmemesi çoğu zaman büyük bir eksiklik olarak karşımıza çıkacaktır. O yüzden C dilinde if ile beraber else yani “Değilse” karar yapısı kullanılmaktadır. Bununla ilgili bir kodu şöyle yazabiliriz.

Eğer A, B’den Büyükse A B’den Büyüktür Yazdır Değilse A B’den küçüktür Yazdır.

Bu durumda karşılaştığımız iki duruma göre ayrı işletilecek kodu belirlemiş oluruz. If ifadesini anlatırken öğrencinin not durumuna göre geçti yazan bir komut yazmıştık. Fakat geçmediği zaman ekranda bir şey yazmıyordu. Kaldı ifadesini yazmak için şöyle bir komut kullanabiliriz.

```
if ( not >= 45 ) {  
printf( "Gecti" );
```



```
}  
else {  
printf( "Kaldi" );  
}
```

Burada not değeri 45'e eşit veya 45'den büyükse printf fonksiyonu ile ekrana "Gecti" yazdırılır. Eğer şart sağlanmıyorsa not değeri 45'den düşük demektir ve öğrenci kalmıştır. Bunu yazdırmak için de printf fonksiyonu ile "Kaldi" yazdırırız. Şimdi if/else yapısının söz dizimini yazalım.

```
if (şart) {  
komutlar;  
komutlar;  
komutlar;  
}  
else  
{  
komutlar;  
komutlar;  
komutlar;  
}
```

```
if (şart)  
tekkomut;  
else  
tekkomut;
```

Şimdi if ve else yapılarının beraber kullanıldığı çok basit bir program yazalım ve programın çalışma mantığını inceleyelim.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int sayi;
printf("Bir Sayi Giriniz:");
scanf("%i", &sayi);
if (sayi >= 100){
    printf("\n Sayi 100'den Buyuktur / Esittir \n");
}
else
{
    printf("\n Sayi 100'den Kucuktur \n");
}

system("PAUSE");
} // Main Sonu
```

Burada önceki örneklerde olduğu gibi if karar yapısıyla başlayarak sayının 100'den büyük veya 100'e eşit olup olmadığını denetledik. Bunu if (sayi >= 100) komutuyla yaptık. Sonrasında printf fonksiyonuyla şart sağlandıysa sayının 100'den büyük veya 100'e eşit olduğunu yazdırdık. Eğer bu şart sağlanmadıysa yani sayı 100'den büyük veya 100'e eşit DEĞİLSE sayının 100'den küçük olduğunu anlarız. Bu durumda else ifadesinin ardından süslü parantezlerle bir kapsam (blok) açıp else yani DEĞİLSE şartı gerçekleştiğinde işletilecek komutu yazdık. Bu komut printf fonksiyonu yardımıyla sayının 100'den küçük olduğunu ekrana yazdırmaktadır.

Buraya kadar else deyiminin ne işe yaradığını anlamış olduk. Bu noktaya kadar her şey basit görünürken else if deyimi karşımıza çıkmaktadır. else if “DEĞİLSE EĞER” ifadesinin karşılığıdır. Yani değer önce bir şart ile karşılaşmalı sonra o şartı sağlamazsa başka bir şart ile karşılaşmalıdır. Bu durumda peş peşe onlarca şart yazma imkanımız olur. Örnek bir else if programını şöyle yazabiliriz.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int sayi;
printf("Bir Sayi Giriniz:");
scanf("%i", &sayi);
if (sayi > 100){
    printf("\n Sayi 100'den Buyuktur\n");
}
else if (sayi == 100 )
{
    printf("\n Sayi 100'e Esittir\n");
}
else
{
    printf("\n Sayi 100'den Kucuktur \n");
}

system("PAUSE");
} // Main Sonu
```

Öncelikle karar yapısını her zaman if deyimi ile başlatıyoruz. İlk ve en önemli şartı buraya yazmamız gereklidir. Sonrasında şart karşılanmazsa ikinci bir şarta değeri tabi tutuyoruz. Bu da else if (sayı == 100) komutuyla burada yapılmıştır. Eğer sayı 100'den büyük değilse ve eğer 100'e eşitse şartını bu program yerine getirmektedir. else if ifadesi ile başlayan şartları yukarıdaki if ile başlayan şarttan bağımsız tutmamak gereklidir. else if şartı da sağlanmazsa en sonunda else if'in DEĞİLSE şartı olan else ifadesini kullanırız ve sayi değişkeninin 100'den küçük olduğuna hükmederiz.

Else if yapılarını peşpeşe kullanarak daha karmaşık bir karar yapısı meydana getirmek mümkündür. Şu program bir robot kumanda programının bir parçası olabilir.

```
if ( yon == sol){
    sola_git();
}
else if (yon == sag )
{
    saga_git();
}
else if (yon == ileri)
{
    ileriye_git();
}
else if (yon == geri)
{
    geriye_git();
}
```

Bu durumda else if yapısının söz dizimini şu şekilde özetleyebiliriz.

```
if (şart)
{
komutlar;
}
else if (şart)
{
komutlar;
}
else if (şart)
{
komutlar;
}
else // isteğe bağlı
{
komutlar;
}
```

Şimdi bu komutlarla konsol tabanlı bir rol yapma oyununun koordinat tabanlı haritasını yapalım ve bu harita üzerinde gezinelim. Klavyemizde aynı diğer oyunlarda olduğu gibi WASD tuşlarına basarak programı kontrol edeceğiz. W kuzey, A batı, S güney ve D harfi de doğu harfine karşılık gelecek. Bizim haritada ilerlediğimizi ve haritada ne konumda olduğumuzu ise printf fonksiyonu ile ekrana yazdıracağız. Bu basit oyun şimdilik boş bir haritada geçse de konumuz ilerledikçe buna ilaveler yaparak daha eğlenceli bir hale getireceğiz.

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

```

int main() {

// KARŞILAMA YAZISI
for (int i=0; i < 50; i++ )
printf("*");
printf("\n");
printf("ROL YAPMA OYUNUNA HOS GELDINIZ \n");
printf("WASD TUSLARI ILE HARITADA ILERLEYINIZ \n");
for (int i=0; i < 50; i++ )
printf("*");
printf("\n");
// KARŞILAMA YAZISI BITIS
int x = 0, y = 0; // kordinatlar
while (1)
{
char yon;
yon = getch();
if (yon == 'w'){
    y++;
}
else if (yon == 's'){
    y--;
}
else if (yon == 'a'){
    x--;
}
else if (yon == 'd') {
    x++;
}
}
}

```

```

}
else
printf("\n  Lutfen Dogru Tusa Basiniz \n");

printf(" X : %i    Y : %i  \n", x, y);

} // sonsuz döngü sonu

system("PAUSE");
} // Main Sonu

```

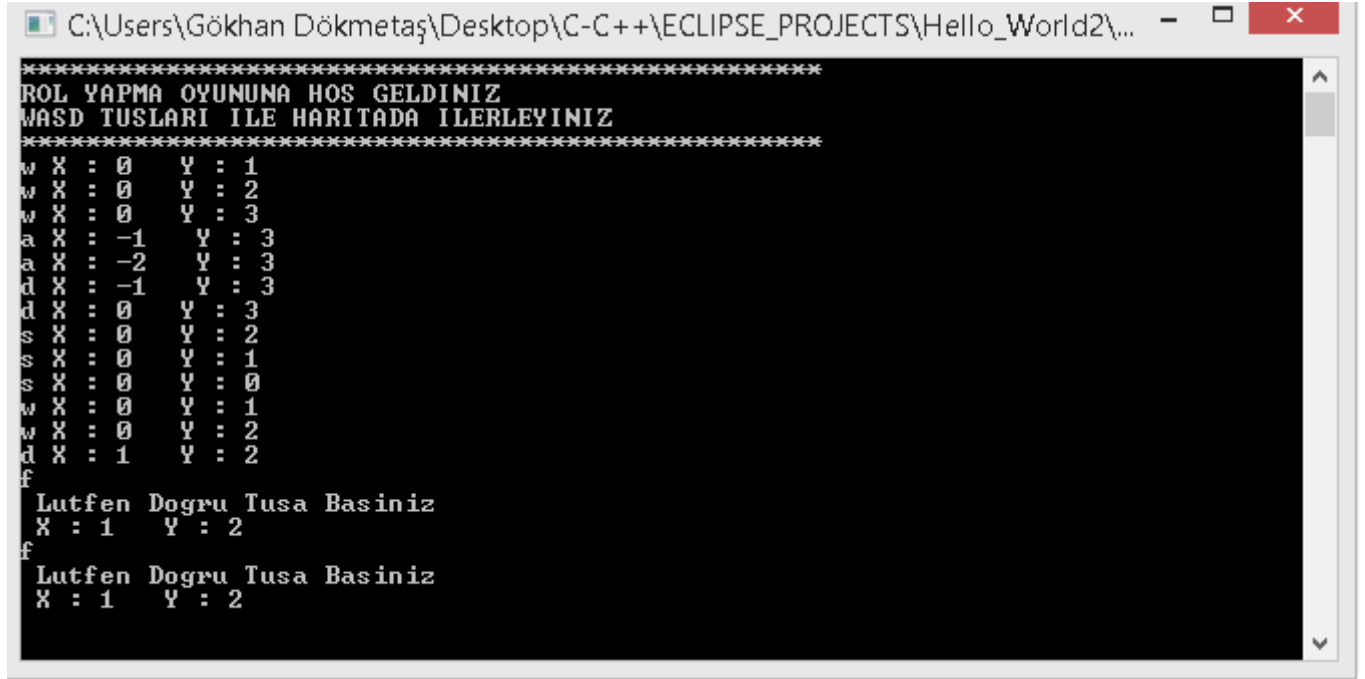
Burada öncelikle programı tanıtmaya yönelik bir karşılama yazısı ekliyoruz. Gömülü sistemler üzerinde çalışırken de LCD, TFT gibi bir çıkış birimi kullandığımızda program açılırken programı tanıtıcı bir açılış metnini ekleriz. Program yazarken bunu alışkanlık haline getirmeniz sizin için iyi olacaktır. #include <conio.h> yönergesi ile yeni bir başlık dosyasını programa ekledik. Bu programda kullanacağımız getch() fonksiyonunu kullanmak için şarttır.

yon = getch();

getch() fonksiyonu scanf fonksiyonu gibi olmayıp klavyede bir tuşa bastığımızda tuşa basar basmaz girilen harfi veya rakam değerini okumakta ve bunu değer olarak döndürmektedir. Eğer klavyede hiçbir tuşa basmazsak getch() fonksiyonu işletildiği anda programı beklemeye almaktadır. while(1) sonsuz döngüsünün içerisine yazdığımız bu komut ile her tuşa bastığımızda döngü bir çevirimde bulunacaktır.

if (yon == 'w'){ y++; }

Eğer W tuşuna basarsak kordinat sistemindeki Y ekseninde değer bir artacaktır. Eğer bastığımız tuş “w” değilse diğer kontrol yapılarından değer geçecek ve ilgili yerde ilgili değeri artıracaktır. Eğer başka bir tuşa bastıysak program hata verecektir. Programın en sonunda ise X ve Y değerlerini ekrana yazdırıyoruz. Programın çıkışı şu şekilde olabilir.



```
*****
ROL YAPMA OYUNUNA HOS GELDİNİZ
WASD TUŞLARI İLE HARITADA İLERLEYİNİZ
*****
w X : 0   Y : 1
w X : 0   Y : 2
w X : 0   Y : 3
a X : -1  Y : 3
a X : -2  Y : 3
d X : -1  Y : 3
d X : 0   Y : 3
s X : 0   Y : 2
s X : 0   Y : 1
s X : 0   Y : 0
w X : 0   Y : 1
w X : 0   Y : 2
d X : 1   Y : 2
f
Lutfen Dogru Tusa Basiniz
X : 1   Y : 2
f
Lutfen Dogru Tusa Basiniz
X : 1   Y : 2
```

Buraya kadar else ve else if yapılarını anlatmış olduk. Sonraki konuda switch ve mantıksal işlem operatörlerini anlatarak bu örnek üzerinde daha karmaşık işlemler yapacağız.

-21- Switch Karar Yapısı, break ve continue

Önceki yazıda bir değeri belli aralıklar ve koşullarda okumak için alt alta else if yapılarını kullanmıştık. Bu else if yapılarını böyle kullanmak programcı için pratik bir yöntem değildir ve programın hata verme ihtimalini artırır. Bu yüzden C dilinde bu sık kullanılan else if yani seçim ağacı yapısını switch deyimi ile kullanmak hem programı daha anlaşılır kılacak hem de daha kolay kod yazmamıza imkan sağlayacaktır. Yalnız baştan söylememiz gerekir ki switch yapısı bize asla if else yapısı kadar esnekliği vermemektedir. Bu yapının neden fazla esneklik vermediğini ise uygulamada sizlere göstereceğiz. switch yapısını anlatmakla döngü ve karar konusunu bitirsek de döngü ve kararlara ait ayrıntıları en son olarak anlatacağız. Böylelikle eksik konu bırakmadan bir sonraki konumuz olan fonksiyonlara geçeceğiz.

Switch komutuna çoklu seçim komutu adı da verilir. Switch komutunda if yapısında olduğu gibi bir karşılaştırma operatörü veya mantıksal bir işlem yoktur. Switch sadece bir değişkeni alır ve bu değişkenin değerine göre belki onlarca farklı komutu işletebilir. Biz önceki yazımızdaki basit rol yapma oyununu burada switch ile yaparak programı geliştirelim.

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

// KARŞILAMA YAZISI
for (int i=0; i < 50; i++ )
```

```

printf("*");
printf("\n");
printf("ROL YAPMA OYUNUNA HOS GELDINIZ \n");
printf("WASD TUSLARI ILE HARITADA ILERLEYINIZ \n");
for (int i=0; i < 50; i++ )
printf("*");
printf("\n");
// KARŞILAMA YAZISI BITIS
int x = 0, y = 0; // kordinatlar
while (1)
{
char yon;
yon = getch();
switch(yon)
{
case 'w':
    y++;
    break;
case 's':
    y--;
    break;
case 'a':
    x--;
    break;
case 'd':
    x++;
    break;
default:

```

```
printf("\n Lutfen Dogru Tusa Basiniz \n");
break;
}
printf(" X : %i   Y : %i   \n", x, y);

} // sonsuz döngü sonu

system("PAUSE");
} // Main Sonu
```

Burada yon adlı değişkenin içerdiği değere göre X ve Y değerlerini artırıp azaltıyorduk. Tek değişiklik else if yapılarının silinip switch yapısının getirilmesi olduğu için sadece switch yapısını inceleyerek devam edelim.

switch(yon)

Burada switch yapısına hangi değişkeni alacağımızı belirliyoruz ve süslü parantezleri açarak yapı blokunu oluşturuyoruz.

case 'w':

Bu komutun tam türkçe karşılığı ” ‘w’ durumunda” anlamına gelir. Yani aldığımız değişken w durumundayken ne iş yapılacağını burada belirtiriz. case ifadesinden sonra bir sabit değer girmemiz gereklidir. if yapılarında olduğu gibi farklı değişkenler ve operatörler burada kullanılamaz. Tam sayı, karakter gibi sabit bir değer yazmamız gereklidir. Biz burada tek tırnak arasında ‘w’ karakterini yazıyoruz.

```
y++;
```

Değerin 'w' karakteri olması durumunda yani klavyeden w karakteri okunması durumunda y değişkeni bir artırılıyor.

break;

Bu komut kapsamdan çıkış komutudur. break komutu ile beraber bütün switch yapısından çıkılır ve program akışına devam edilir. if, while hatta ana fonksiyonda bile break; komutunu kullandığımızda break komutu program akışını ilk süslü parantezin kapandığı noktaya götürecektir (})

default:

Burada yukarıdaki değerler karşılanmazsa standart olarak işletilecek komutlar yer almaktadır. Buraya işimize yararsa bir komut ekleyebiliriz veya break; yazıp es geçebiliriz. Biz doğru tuşa basma konusunda biz uyarı ekledik.

Switch yapısının söz dizimi şu şekildedir.

```
switch(degisken)
{
case deger1:
komutlar;
komutlar;
break;
case deger2:
komutlar;
komutlar;
break;

default:
```

```
break;  
}
```

Switch yapısı belli bir veriyi işleme ve bu verideki değerleri ayıklama ve değerleri sayma konusunda çok işimize yaramaktadır. Yanlız biz kümelenmiş if else yapısında switch’de yapamadığımız pek çok işlemi yapabilme şansımız vardır. Örneğin if parantezleri içerisinde mantık operatörleri, aritmetik operatörler hatta fonksiyonlar kullanılabilir. Bunlar trigonometri, karekök alma fonksiyonları bile olabilir. Fakat Switch’in case yapısında sadece değerler okunur ve o değerlere göre işlem yapılır. Örneğin case deger == 5: diye bir komut yazmamız mümkün değildir.

Durum Operatörü

Durum operatörünü açık söylemek gerekirse ne ben kullanıyorum ne de çoğu programcı kullanıyor. İncelediğim kütüphanelerde ve hatta gelişmiş programlarda bile bu operatöre pek rastlamadım. Rastladığım nadir durumlarda da bunu bilmenin faydasını gördüm. O yüzden konuyu eksik bırakmama adına durum operatörünü size anlatacağım.

C dilinde aşırı derecede sık kullanılan bir karar yapısı olduğu için C tasarımcıları bu karar yapısını tek satıra indirip bir operatör ile basitleştirmek istemiştir. Bu karar yapısı iki değeri karşılaştırır ve karşılaştırma operatörüne göre doğru olan değeri değişkene aktarır. Bunun kodu şu şekildedir.

```
if ( deger1 == deger2 )  
    deger3 = deger1;  
else  
    deger3 = deger2;
```

Burada bir şart ve şartın durumuna göre bir atama işlemi gerçekleşmektedir. Aynı kodu durum operatörü ile şu şekilde yazabiliriz.

```
deger3 = ( deger1 == deger2) ? deger1 : deger2 ;
```

Bu kod yapısı pek alışkın olmadığımız için pratikte kolay olsa da anlaşılabilirlikte yukarıdaki if else yapısı gibi değildir. O yüzden uygulamada çok fazla görme imkanınız yoktur.

break ve continue ifadeleri

Ben C dilini öğrenirken break komutunu rahatça öğrensem de continue deyimini anlamam zaman aldı. Açıkçası İngilizce kaynakları okuyana kadar continue deyimini tam olarak anlamış değildim. Şimdi sizlere bu iki deyimi tam anlamıyla açıklamaya çalışacağım.

break ifadesi Türkçe “Ayrıl” anlamına gelmektedir. Yani program akışı bir kapsama bağlı ise bulunduğu kapsamdan ayrılıp bir üst kapsamda program akışına devam eder. Bu kapsamların süslü parantezlerle ayrıldığını buraya kadar öğrendik. Şimdi uygulamada break’ın nasıl davranacağını görelim.

```
int main ()
{ // ANA PROGRAM KAPSAMI

while(1)
{ // SONSUZ DÖNGÜ KAPSAMI
komutlar;
komutlar;
break;
komutlar;
```

```
komutlar;  
} // KAPSAMDAN CIKIŞ  
// BREAK komutunun ardından devam edilen nokta  
komutlar;  
komutlar;  
komutlar;  
  
}
```

Burada önce program int main dediğimiz ana program fonksiyonunun kapsamı içerisinde yer alıyor. Sonrasında while(1) döngüsü ile sonsuz döngünün kapsamı içerisinde sürekli dönüp duruyor. Bu sonsuz döngünün kapsamı while(1)'den sonraki { işaretinden bir sonraki } işaretine kadar devam eder ve bu aradaki komutlar çalıştırılır. Eğer ortadaki break komutunu çalıştırsak alttaki iki komut çalıştırılmaz döngü kapsamından hemen çıkılır ve alttaki } işaretinden itibaren program akışı devam eder. break komutu acil çıkış komutu olarak da nitelendirilebilir. Herhangi bir şarta ve kurala bağlı olmadan istediğimiz program kapsamından atlayıp çıkmamızı sağlamaktadır. Genelde de döngülerde bir şarta bağlanarak kullanılır.

continue deyiminin çevirisi ise “Es Geç” ya da “Atla” olarak ifade edilebilir. continue deyimi break; de olduğu gibi kural ve şart tanımadan kapsamdan çıkış yapmaz. Döngü akışında altındaki komutları atlayarak döngü başına tekrar döner. Bu durumda bir döngü çeviriminde kendinden sonra gelen komutların işletilmesini iptal etmektedir. Bunu da şu şekilde ifade edebiliriz.

```
for (int i = 0; i < 50; i++)  
{ // CONTINUE BU NOKTAYA ATLAMA YAPAR  
komutlar;  
komutlar;
```

```
komutlar;  
continue;  
komutlar; // CONTINUE ÇALIŞIRSA BU KOMUTLARI İŞLETMEZ  
komutlar; // CONTINUE ÇALIŞIRSA BU KOMUTLARI İŞLETMEZ  
komutlar; // CONTINUE ÇALIŞIRSA BU KOMUTLARI İŞLETMEZ  
}
```

Burada continue programı ne döngüden ne de kapsamdan çıkarmaktadır. continue ifadesi geçen yerde program döngü başına atlama yapmaktadır ve devamındaki komutları es geçmektedir. Bu ifadeler döngü içerisinde şart durumlarına bağlanarak kullanılır. Bir sonraki yazımızda mantık operatörlerini, operatör önceliğini ve birleşik operatörleri anlatarak konumuza devam edeceğiz.

-22- Mantık Operatörleri, Birleşik Operatörler ve Operatör Önceliği

Döngü ve kararları bitirmeden önce şu ana kadar anlatmadığımız noktaları sizlere anlatacağız. Önceki yazılarda size aritmetik operatörleri anlatmıştık sonrasında ise döngü ve kararlarda sıkça kullandığımız ilişki operatörlerini

anlattık. Bu ikisinden farklı olarak bit tabanlı operatörler ve mantık operatörlerini sayabiliriz. Bit tabanlı operatörler alt seviye programcılığı ilgilendirdiği ve ileri bir seviye olduğu için ilerleyen zamanlarda sizlere anlatacağız. Bit tabanlı operatörleri gömülü sistemler üzerinde çalışırken aşırı derecede fazla kullanırız. Hatta bit tabanlı operatörler olmadan gömülü sistemler üzerinde program yazmak imkansız gibidir. Fakat biz bilgisayar programı yazarken bunlara çok da ihtiyaç duymayız. Ne zaman ileri seviye programcılık algoritmaları yazmaya başlarsak o zaman bit manipülasyonuna ihtiyaç duyarız.

Mantık operatörleri ise bizim döngü ve karar yapılarını birleşik ve karmaşık hale getirme yollarından biridir. Mantık operatörlerini sürekli kullanmasak da muhakkak kullanma ihtiyacı hissederiz. Çünkü ilişkisel operatörler doğru veya yanlış ifadeleri verse de bu ifadeler tek mantık ifadesi şeklinde olmaktaydı. Bu durumda daha karmaşık mantık uygulamaları yapmak için mantık operatörlerinden faydalanırız. Mantık operatörleri sayesinde birden fazla mantık durumu üzerinde işlem yaparız. Bilgisayar programcılığına yeni başlayanlarda yanlış bir algı vardır. Bütün bu programcılık mantığının, yapay zekanın ve karmaşık programların matematik bilgisi tabanlı olduğunu ve iyi matematik bilen birinin iyi program yazacağını düşünürler. Aslında bilgisayar programları matematik değil mantık tabanlıdır. Biz programlama yaparken mantık bilgisini defalarca kullansak da matematik işlemleri sadece aritmetik operatörler ile sınırlı kalmaktadır. Matematik değil mantık ön plandadır.

Şimdi mantık operatörlerine bir göz atalım.

Operatör	Açıklama
----------	----------

&&	Mantıksal AND (VE) Operatörü. Bütün operandlar 1 ise 1 çıkışı verir.
 	Mantıksal OR (VEYA) Operatörü. Operandlardan biri veya ikisi 1 ise 1 çıkışı verir.
!	NOT (DEĞİL) yani tersleme operatörü. 1 ise 0, 0 ise 1 yapar.

Burada mantık operatörlerinin sadece üç tane olduğunu görebilirsiniz fakat dijital elektronik ve programlanabilir mantıkta anlattığımız üzere bu operatörleri beraber kullanarak istediğiniz mantıksal sonucu elde edebilirsiniz. Örneğin VE DEĞİL işlemi için `!(A&&B)` ifadesini, VEYA DEĞİL işlemi için `!(A||B)` işlemini kullanabiliriz. Bu operatörleri yan yana kullanarak giriş sayısını artırabiliriz. Örneğin A, B ve C değerlerini “VE” işlemine tabi tutmak için `A && B && C` ifadesini kullanabiliriz. İstersek bir parantez içinde iki mantık işleminin sonucunu yine bir mantık işlemine tabi tutabiliriz. Örneğin `(A&&B) || (C&&D)` gibi bir işlem yazma imkanımız vardır. Şimdi bu operatörleri uygulamada görelim.

Biz switch konusunda bir rol yapma oyununa ait boş bir harita tasarlamıştık ve WASD tuşlarıyla bu haritada geziyorduk. Fakat bu harita bomboş olduğu için herhangi bir heyecanı kalmıyordu. Biz bu haritanın belli bir kordinatına hazine, bir kordinatına da ejderha ekleyelim. Böylelikle biraz heyecan katalım.

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

```

int main() {

// KARŞILAMA YAZISI
for (int i=0; i < 50; i++ )
printf("*");
printf("\n");
printf("ROL YAPMA OYUNUNA HOS GELDINIZ \n");
printf("WASD TUSLARI ILE HARITADA ILERLEYINIZ \n");
for (int i=0; i < 50; i++ )
printf("*");
printf("\n");
// KARŞILAMA YAZISI BITIS
int x = 0, y = 0; // kordinatlar
while (1)
{
char yon;
yon = getch();
switch(yon)
{
case 'w':
    y++;
    break;
case 's':
    y--;
    break;
case 'a':
    x--;
    break;

```

```

case 'd':
    x++;
    break;
default:
printf("\n Lutfen Dogru Tusa Basiniz \n");
break;
}
printf(" X : %i   Y : %i   \n", x, y);
if( (x==15) && (y==2))
{
printf("\n Tebrikler!! \n Hazineyi Buldunuz!");
printf("\n GAME OVER!");
break;
}
if( (x==12) && (y==3))
{
printf("\n Tebrikler!! \n Ejderhayi Buldunuz Gecmis Olsun
:))!");
printf("\n GAME OVER!");
break;
}
} // sonsuz döngü sonu

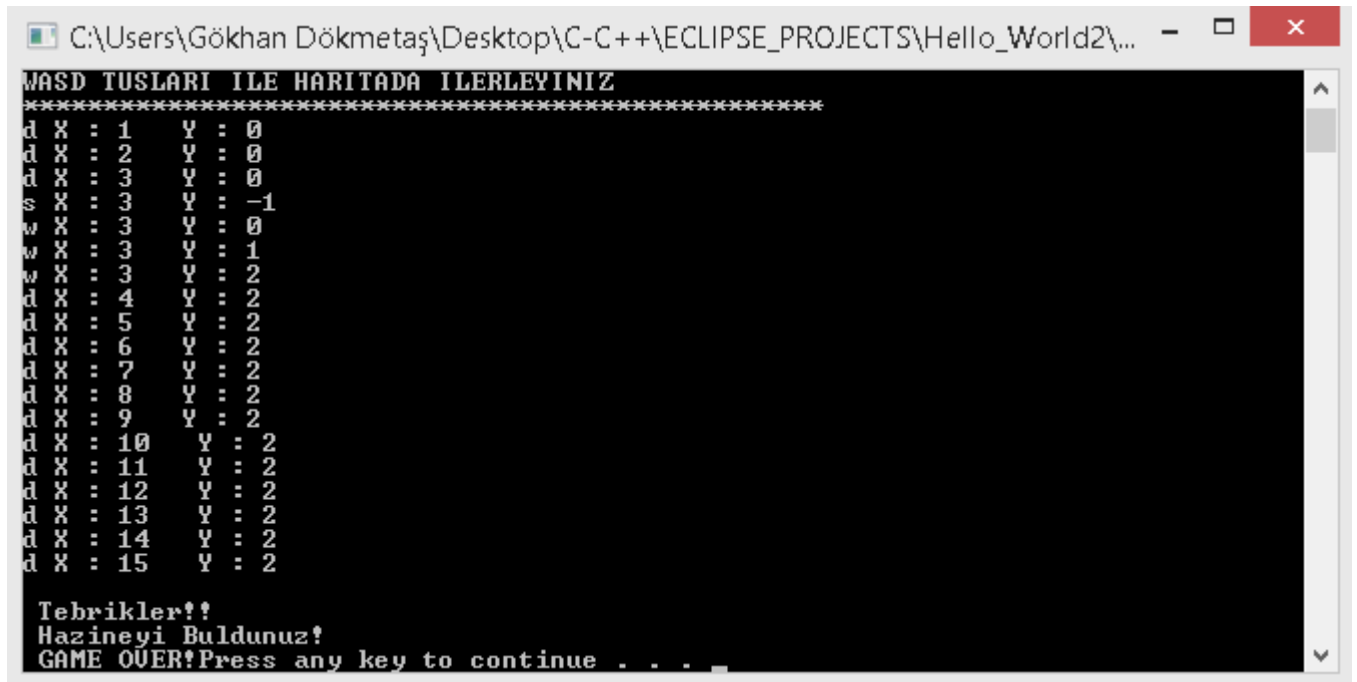
system("PAUSE");
} // Main Sonu

```

Yazdığımız program önceki programın aynısı olsa da biz buraya iki adet if yapısı ekledik. If karar yapıları hem x değerini ve hem y değerini değerlendiriyor ve ikisi de istenilen şartı karşılırsa karar yapısını çalıştırıyor.

if((x==15) && (y==2))

Böyle bir komutun Türkçe karşılığı **EĞER X 15'e Eşitse VE Y 2'ye Eşitse** şeklindedir. Görüldüğü gibi koordinat sisteminde bir noktayı böylelikle belirliyoruz. Oyunda hem hazineyi hem de ejderhayı bulmamız mümkün. Ejderhayı bilerek hazinenin yakınına koyduk. Maksudımız ejderhaya yem olmadan hazineyi bulabilmek. İkisinde de break; komutu ile kapsamdan (sonsuz döngüden) çıkıyoruz ve system("PAUSE") fonksiyonu işletilerek beklemeye alınıyor. Programın bir çıktısı şu şekilde olabilir. Siz isterseniz hazinenin nerede olduğuna dair bir ipucunu programın başında kullanıcıya bildirebilirsiniz.



```

C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...
MASD TUSLARI İLE HARITADA İLERLEYİNİZ
*****
d X : 1    Y : 0
d X : 2    Y : 0
d X : 3    Y : 0
s X : 3    Y : -1
w X : 3    Y : 0
w X : 3    Y : 1
w X : 3    Y : 2
d X : 4    Y : 2
d X : 5    Y : 2
d X : 6    Y : 2
d X : 7    Y : 2
d X : 8    Y : 2
d X : 9    Y : 2
d X : 10   Y : 2
d X : 11   Y : 2
d X : 12   Y : 2
d X : 13   Y : 2
d X : 14   Y : 2
d X : 15   Y : 2

Tebrikler!!
Hazineyi Buldunuz!
GAME OVER! Press any key to continue . . . .
```

Bizim burada bir sıkıntımız haritanın oldukça geniş olmasıdır. x ve y değerleri integer tipinde olduğundan + milyarlardan – milyarlar kadar gitmektedir. Bu

durumda oyuncu belki hazineyi bulmak için yıllarını harcaması gerekebilir. O yüzden haritayı sınırlarını aştığını oyuncuya bildirmek için bir komut yazmamız lazımdır.

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

// KARŞILAMA YAZISI
for (int i=0; i < 50; i++ )
printf("*");
printf("\n");
printf("ROL YAPMA OYUNUNA HOS GELDİNİZ \n");
printf("WASD TUSLARI İLE HARITADA İLERLEYİNİZ \n");
for (int i=0; i < 50; i++ )
printf("*");
printf("\n");
// KARŞILAMA YAZISI BITİSİ
int x = 0, y = 0; // kordinatlar
while (1)
{
char yon;
yon = getch();
switch(yon)
{
case 'w':
y++;
```

```

        break;
case 's':
    y--;
    break;
case 'a':
    x--;
    break;
case 'd':
    x++;
    break;
default:
printf("\n Lutfen Dogru Tusa Basiniz \n");
break;
}
printf(" X : %i   Y : %i   \n", x, y);
if( (x==15) && (y==2))
{
printf("\n Tebrikler!! \n Hazineyi Buldunuz!");
printf("\n GAME OVER!");
break;
}
if( (x==12) && (y==3))
{
printf("\n Tebrikler!! \n Ejderhayi Buldunuz Gecmis Olsun
:)))!");
printf("\n GAME OVER!");
break;
}

```

```

if ((y>50) || (x>50) || (y<-50) || (x<-50) )
    printf("\n Harita Sinirlarini Astiniz!");
} // sonsuz döngü sonu

system("PAUSE");
} // Main Sonu

```

Burada if ((y>50) || (x>50) || (y<-50) || (x<-50)) yapısını kullanarak 100×100 bir harita ortaya çıkardık. Yani x değeri -50'den +50'ye y değeri de +50'den -50'ye kadar gidebilir. Eğer harita sınırları aşılsa kullanıcıya uyarı verilir. Bu komutun Türkçe'si EĞER Y 50'den büyükse VEYA X 50'den Büyükse VEYA Y -50'den küçükse VEYA X -50'den küçükse demektir. Programın çıktısı şu şekilde olacaktır.

```

C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...
a X : -46    Y : 47
a X : -47    Y : 47
a X : -48    Y : 47
a X : -49    Y : 47
a X : -50    Y : 47
a X : -51    Y : 47

Harita Sinirlarini Astiniz!a X : -52    Y : 47
Harita Sinirlarini Astiniz!a X : -53    Y : 47
Harita Sinirlarini Astiniz!a X : -54    Y : 47
Harita Sinirlarini Astiniz!d X : -53    Y : 47
Harita Sinirlarini Astiniz!d X : -52    Y : 47
Harita Sinirlarini Astiniz!d X : -51    Y : 47

Harita Sinirlarini Astiniz!f
Lutfen Dogru Tusa Basiniz
X : -51    Y : 47

Harita Sinirlarini Astiniz!d X : -50    Y : 47

```

Mantık operatörleri gömülü sistemlerde sıkça kullandığımız operatörler olmaktadır. Özellikle ! operatörünü 1'i 0 yapmak için 0'ı da 1 yapmak için

kullanmaktayız. Bit bazlı operatörler ile mantık operatörlerini birbirine karıştırmamak gereklidir. O yüzden bit bazlı operatörleri anlatmayı daha sonraya bırakıyoruz.

Operatör Önceliği

Daha önceden aritmetik operatörleri anlattığımızda çarpma ve bölme işleminin toplama ve çıkarma işlemine önceliği olduğunu söylemiştik. Fakat işin içerisine farklı operatörler girince öncelik değerini daha iyi hesaplamak gerekiyor. Ben bu öncelik meselesini farkında olmadan gözetir oldum ve hiç tabloya bakma ihtiyacı hissetmedim. Çok kafa karıştırmayacak şekilde işlemleri yazdığınızda sizin de bu tabloyu ezberlemenize pek gerek kalmıyor. Parantezleri bol bol kullanmanız daha okunur bir program ortaya çıkardığı gibi öncelik konusunda sıkıntı çıkmasına da engel olmaktadır.

Kategori	Operatör
Sonek	() [] -> . ++ --
Tekli	+ - ! ~ ++ -- (type)* & sizeof
Çarpma ile alakalı	* / %
Ekleme ile alakalı	+ -
Kaydırma	<< >>
İlişki	< <= > >=
Eşitlik	== !=
Bit tabanlı AND	&

Bit tabanlı XOR	^
Bit tabanlı OR	
Mantıksal AND	&&
Mantıksal OR	
Durum	?:
Atama	= += -= *= /= %= >>= <<= &= ^= =
Virgül	,

Birleşik Operatörler

Biz programlamada aritmetik operatörleri kullanırken tek bir değişken üzerinde işlem yapmak için $a = a + b$ ya da $a = a / b$; gibi komutları aşırı derecede fazla kullanırız. Bu yüzden bu komutları uzun uzun yazmak yerine kısaca yazmak için C dilinde birleşik operatörler diye tabir edilen operatör kategorisi vardır. Bu operatörler şu şekildedir.

Operatör	Açıklama	Örnek
+=	Topla ve Ata Operatörü. Sağ taraftaki değeri sol taraftaki değere ekler.	A += B işlemi A = A + B 'ye Eşittir.
-=	Çıkar ve Ata Operatörü. Sağ taraftaki değeri sol taraftaki değerden çıkarır.	A -= B işlemi A = A – B 'ye Eşittir.

*=	Çarp ve Ata Operatörü. Sağ taraftaki değerle sol taraftaki değeri çarpar.	A *= B işlemi A = A*B'ye eşittir.
/=	Böl ve Ata Operatörü. Sağ taraftaki değerle sol taraftaki değeri böler.	A /= B işlemi A = A / B 'ye eşittir.

Bundan başka bit bazlı operatörler için de kullanılan birleşik operatörler vardır fakat biz bit bazlı operatörler konusunu ayrı bir konu olarak ele alacağız.

goto komutu

Assembly programlarında döngü yapıları yani yordamlar ve alt yordamlar while, for gibi yapıları içermemektedir. Biz program akışında istediğimiz bir noktaya etiket ekleriz ve goto komutuyla program bu etikete sıçrama yapar. etiket ve goto yapısını kullanmak bir dönem programcılarının başına bela olmuştur ve o yüzden yapısal programlama dillerinde yerini fonksiyonlara ve döngülere bırakmıştır. Yine de goto komutuyla ileri seviye noktalarda etkili programlar yazabilirsiniz. C dilinde normalde goto komutuna hiç ihtiyaç olmasa da performans istenilen noktalarda kullanılabilir. goto üzerinden örnek yapmayacağız fakat söz diziminden bahsedeceğiz.

Etiket:

```
komutlar;
```

```
komutlar;
```

```
komutlar;
```

```
goto etiket; // Etiket: noktasına git
```

Buraya kadar döngü ve kararlardan itibaren öğrenmemiz gereken hemen her noktayı öğrenmiş olduk. Temel komutların hepsini bildiğimiz için artık kendi programlarımızı yazıp bunları parçalara bölebiliriz. Bu yüzden fonksiyonları anlatarak konumuza devam edeceğiz. Gelişmiş veri tiplerini ise fonksiyonlardan itibaren anlatacağız ve bu arada işaretçileri de anlatmayı ihmal etmeyeceğiz.

-23- Fonksiyonlara Giriş

Artık bu noktadan itibaren giriş seviyesini bıraktık ve orta seviyeye doğru ilerliyoruz. Okuyuculardan ricamız bu noktaya kadar anlattıklarımızı tam anlamıyla anlamış olmaları ve yazılan program kodlarını okuyabilecek seviyeye gelmeleridir. Eğer dersleri takip etmenize rağmen bu konuda bir eksikliğiniz varsa hiç çekinmeden farklı kaynaklardan faydalanabilirsiniz. Video, kitap ya da eğitim yazısı olsun hepsi aynı konuları anlatsa da farklı örnekleri inceleme ve farklı anlatımlardan yararlanma imkanınız olur. Ben gömülü sistem geliştiricisi olduğum için konuya kendi bakış açımdan bakabilirim. Siz farklı yazarlardan faydalandıkça konuya bakış açınız daha genişleyecektir. Ben öğrendiğimi pekiştirmek için aynı konu hakkında yazılmış farklı İngilizce kitapları okumaktayım.

Fonksiyonlar bizim karmaşık program yazmamızdaki ilk adımımızdır. Ayrıca C dilinde en çok kullanılan yapılardan biridir ve dilin olmazsa olmazıdır. C dilinde

yazılan programların hepsi fonksiyonlar üzerine bina edilmiştir. Biz fonksiyon konusunu anlatmadan önce bile ilk programda `main()` ya da `printf()` fonksiyonlarını kullanarak fonksiyonlara giriş yapmış olduk. Fonksiyon olmayan bir C programını düşünmemiz imkansızdır çünkü en azından bir `main` fonksiyonunu içinde bulundurmaktadır.

Fonksiyonları bölünmüş program blokları olarak düşünebiliriz. Biz 100-200 satır programı geçtikten sonra yazdığımız program aşırı derecede büyük ve bunu bölümlere ayırmamız gerekir. Aynı bir kitap yazarken başlıklara ayırmak ya da öğrencileri sınıflara ayırmak gibi kodları da fonksiyonlara ayırırız ve blok blok kullanırız. Birbiriyle alakalı komutlar veya bir işi yapmaya yönelik komutlar bir fonksiyon içerisinde toplanır. Fonksiyonların en önemli kullanım sebebi kodun organize etmektir. Ayrıca tekrar eden kod bloklarını da fonksiyon olarak belirterek sürekli aynı program blokunu yazmaktan kurtuluruz. Bir kere fonksiyonu yazdıktan sonra defalarca kullanabiliriz ve bellekte tekrar tekrar yer kaplamaz.

Fonksiyonların kullanıldığı ve kodların tekrar etmediği bir program bu yönden optimize edilmiş bir programdır. Böylece bellekte daha az yer tutmaktadır. İşletilen kod sayısında bir değişiklik olmasa da özellikle gömülü sistemlerde kısıtlı hafıza alanında çalıştığımız için bellek miktarı sıkıntı olmaktadır.

Bellek yiyen programlar bazen bilgisayarlarda da sıkıntı haline gelmektedir. Çok basit yapıları programlar bile belki yanlış programlama dilinden belki de optimizasyondan dolayı aşırı derecede bellek tüketmektedir ve yüksek seviye sistemlerde bile istenilen performansı verememektedir.

C’de programların modüllere fonksiyonlar ile ayrıldığını söylemiştik. Bu modüller aynı lego parçaları veya yapboz parçası gibi belli bir şekilde ve amaca yönelik olmaktadır. Bölünmüş programlar üzerinde hakimiyet kurmak daha kolay olmaktadır. Bu program parçalarını çeşitli dizilimlerde bir araya getirerek programı

düzenleyebileceğimiz gibi farklı programlar da yazabiliriz. Örneğin `printf` fonksiyonu ekrana formatlı yazı yazdıran bir program modülüdür. Fakat bu fonksiyonu biz yazacağımız yüzlerce örnekte kullanabiliyoruz. Sadece fonksiyon adını çağırarak birbirinden alakasız programlara entegre edebiliriz.

C dilinde hazır fonksiyonlar standart kütüphanelerle beraber gelse de istersek biz bu fonksiyonların taklitlerini yazabileceğimiz gibi tamamen kendimize ait bambaşka bir fonksiyon da yazabiliriz. Fonksiyonlar `printf` fonksiyonundan gördüğümüz üzere argüman adı verilen değerleri almaktadır. Bu değerler fonksiyonun program blokuna gider ve ilgili yerlerde işletilir. Ayrıca fonksiyonların değer döndürdüğünü de gördük. Örneğin `getche()` fonksiyonu klavyeden girilen karakter değerini bize bildiriyordu.

Fonksiyon konularını geniş bir alana yayacağız ve çeşitli örneklerle zenginleştireceğiz. Çünkü fonksiyonları anlamak ve etkili kullanmak iyi program yazmanın temelidir. Fonksiyonları iyi kullanabilmek için bolca pratik ve tecrübe de gerekmektedir. Fakat bunlardan önce kapsamlı bir teorik bilgiye ihtiyaç vardır.

-24- Basit Fonksiyonlar

Önceki başlıkta fonksiyona giriş yapmıştık ve fonksiyon mantığını sizlere anlatmıştık. Bu başlıkta ise basit fonksiyonları ve bunların kullanımını sizlere anlatacağız. Biz önceki yazıda hazır fonksiyonların C kütüphanelerinde yer aldığından bahsetmiştik. Aslında bu fonksiyonları kullanmada hiç bir sıkıntımız yok çünkü ilk programdan itibaren printf ve scanf gibi fonksiyonları kullanmaktayız. Bu fonksiyonların tamamını öğrenmek için kütüphanenin başlık dosyasına ve referans kılavuzuna bakabiliriz. C standart kütüphanesini ilerleyen konularda anlatacak olsak da konumuzda yapacağımız örnek fonksiyonları cmath.h kütüphanesindeki fonksiyonlarla karşılaştıracğız. Şimdi en basit bir işlemi kısa yoldan yapmak için önceden yazdığımız programa göz atalım.

else if yapısını anlattığımızdan beri bir rol yapma oyununun taslağı üzerinde çalışıyorduk. Burada giriş yazısını yazdırırken printf fonksiyonunu aynı iş için iki kere kullanmıştık. O kısmı size gösterelim.

```

for (int i=0; i < 50; i++ )
printf("*");
printf("\n");

printf("ROL YAPMA OYUNUNA HOS GELDINIZ \n");
printf("WASD TUSLARI ILE HARITADA ILERLEYINIZ \n");

for (int i=0; i < 50; i++ )
printf("*");
printf("\n");

```

Burada gördüğünüz gibi aynı kod blokunu iki kere kullanıyoruz. Bunu defalarca kullanmak istesek kopyala yapıştır ile programın dört bir yanına yapıştırmamız gerekecekti. Bunu kısaltma ve modül haline getirmek için şimdi fonksiyonları kullanacağız. Bizim yapacağımız belli bir görev vardı. O görev de ekrana belli bir sayıda yıldız (*) yazdırarak çizgi haline getirmekti. Biz bunu her yapmak istediğimizde ise yukarıdaki for döngüsünü kullanmak zorundaydık. Şimdi fonksiyon için gereken komutları bir arada toplayalım.

```

for (int i=0; i < 50; i++ )
printf("*");
printf("\n");

```

Bundan başka bir komut yazmak gerekli değil, değil mi? O halde bu komutu bir kenarda bekletiyoruz ve şimdi programa yeni bir fonksiyon prototipi yazıyoruz.

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
//Fonksiyon prototipleri

```



```
void cizgi_yaz (void);  
////////////////////
```

```
int main() {
```

Programın baş tarafına **void cizgi_yaz (void);** şeklinde fonksiyon prototipini yazdık. Fonksiyon prototipi aynı int x, float y gibi değişken tanımladığımız gibi fonksiyon tanımlamamızı sağlayan bir yapıdır. Biz burada daha fonksiyonun kendisini yazmadık. Derleyiciye böyle bir fonksiyon olduğunun haberini verdik. C++ dilinde fonksiyon prototiplerini yazmadan doğrudan fonksiyon blokunu yazarak da kullanabilirsek de C dilinde derleyicinin böyle bir fonksiyonu tanımladığımızdan haberi olması için fonksiyon prototiplerini yazıyoruz. Şimdi programın en alt kısmındaki boşluğa gelelim ve fonksiyonumuzu yazalım.

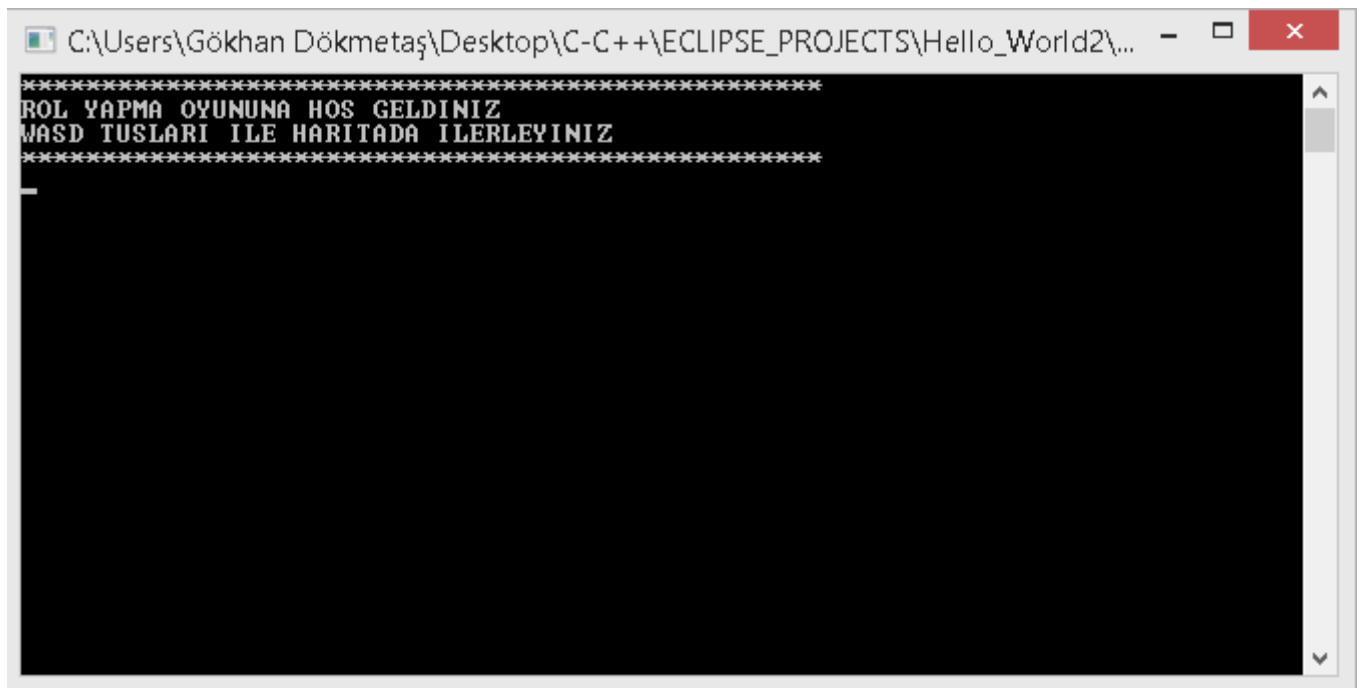
```
void cizgi_yaz (void)  
{  
    for (int i=0; i < 50; i++ )  
        printf("*");  
        printf("\n");  
}
```

Burada çizgi yazdırmak için gerekli kodları fonksiyonun içerisine yazdırarak bu kod blokunu bir modül haline getirdik. Artık printf demek gibi cizgi_yaz() dediğimizde ekrana çizgi yazacaktır. O halde şimdi ana programdaki döngüleri silip yerine cizgi_yaz() fonksiyonunu çağırıyoruz. Burada void'in boş değer olduğunu hatırlatalım.

```
int main() {  
  
    // KARŞILAMA YAZISI  
    cizgi_yaz();  
    printf("ROL YAPMA OYUNUNA HOS GELDINIZ \n");
```

```
printf("WASD TUSLARI ILE HARITADA ILERLEYINIZ \n");  
cizgi_yaz();  
// KARŞILAMA YAZISI BITIS
```

Burada artık kod kalabalığının kalktığını ve ne iş yaptığımızı rahatça görebiliyoruz. Oraya printf fonksiyonları ile beraber for döngüsünü koymak mı daha anlaşılırdı yoksa bir cizgi_yaz() adında fonksiyon koymak mı ? Artık kodu okuduğumuzda o fonksiyonun çizgi yazdırma işini yaptığını anlayabiliriz. Fonksiyonlara ad verirken yaptığı işlerin adını vermemiz bu noktada önemlidir. Programın çıktısına baktığımızda önceki ile arasında bir fark olmadığını göreceğiz.



```
C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Hello_World2\...  
*****ROL YAPMA OYUNUNA HOS GELDINIZ  
WASD TUSLARI ILE HARITADA ILERLEYINIZ*****  
_
```

İncelemeniz için kodun tamamını da verelim. İsterseniz çeşitli görevleri fonksiyonlara bölerek çalışma yapabilirsiniz.

```
#include <stdio.h>  
#include <stdlib.h>  
#include <conio.h>  
//Fonksiyon prototipleri
```

```

void cizgi_yaz (void);
////////////////////

int main() {

// KARŞILAMA YAZISI
cizgi_yaz();
printf("ROL YAPMA OYUNUNA HOS GELDINIZ \n");
printf("WASD TUSLARI ILE HARITADA ILERLEYINIZ \n");
cizgi_yaz();
// KARŞILAMA YAZISI BITIS
int x = 0, y = 0; // kordinatlar
while (1)
{
char yon;
yon = getch();
switch(yon)
{
case 'w':
    y++;
    break;
case 's':
    y--;
    break;
case 'a':
    x--;
    break;
case 'd':
    x++;

```

```

        break;
default:
printf("\n Lutfen Dogru Tusa Basiniz \n");
break;
}
printf(" X : %i   Y : %i   \n", x, y);
if( (x==15) && (y==2))
{
printf("\n Tebrikler!! \n Hazineyi Buldunuz!");
printf("\n GAME OVER!");
break;
}
if( (x==12) && (y==3))
{
printf("\n Tebrikler!! \n Ejderhayi Buldunuz Gecmis Olsun
:)))!");
printf("\n GAME OVER!");
break;
}

if ((y>50) || (x>50) || (y<-50) || (x<-50) )
    printf("\n Harita Sinirlarini Astiniz!");
} // sonsuz döngü sonu

system("PAUSE");
} // Main Sonu

```

```
void cizgi_yaz (void)
{
    for (int i=0; i < 50; i++ )
        printf("*");
        printf("\n");
}
```

Şimdi cizgi_yaz() fonksiyonu üzerinden bir fonksiyonu nasıl tanımladığımızı size açıklayalım. Öncelikle fonksiyon tanımlamadan önce bir fikir aşaması olur ve program yazmadan önce veya program esnasında fonksiyon oluşturma ihtiyacı duyarız. Bunun fonksiyon olması gerektiğini karar verdikten sonra fonksiyonun değer alıp almayacağını değer döndürüp döndürmeyeceğini hesaplarız. Sonrasında ise fonksiyona güzel bir isim bulmamız gereklidir. Yukarıda fonksiyon yapmamızın sebebi kodların tekrarlanması ve kod kalabalığıydı. Buna isim seçmenin ise değişken ismi kuralları olduğu gibi belli kurallar çerçevesinde bir sınırı yoktur. Şimdiye kadar size değişken ve fonksiyon adı seçme kurallarını anlatmadığımız için şimdi konuyu bölerek bunu anlatalım.

Değişken ve Fonksiyon Adı Seçme Kuralları

Şimdiye kadar yaptığımız örneklerde değişken ve fonksiyon adlarını kafamıza göre seçtiğimizi düşünüyorsanız yanılıyorsunuz. C dilinde değişkenlere ve fonksiyonlara istediğimiz adı verebilsek de bunun sıkı bir kuralı vardır. Kurallara aykırı bir ad vermemiz mümkün değildir. Üstelik kurallar çerçevesinde bile ad vermenin bir usulü olmaktadır. Usulsüz verilen bir ad okunabilirliği zorlaştırır ve programın kalitesini düşürür. Şimdi önemli kuralları sıralayalım.

- Değişken adları çok uzun olamaz. Değişken adları Visual C++'da 1 ile 255 karakter arasında olabilir. Başka geliştirme ortamlarında bu 1 ile 31 karakter arasında olabilir. O yüzden siz çok uzun değişken adı koymaktan kaçınmalısınız. Çünkü hem okunabilirliği, hem kolay program yazımını hem de taşınabilirliği etkilemektedir.
- Bütün değişken adları ya harf ile ya da alt tire (_) ile başlamalıdır. Rakam ile başlayan değişken adı kabul edilmez.
- Değişken adı koyulduğunda ilk karakter harf veya alt tire (_) olduktan sonra harf, alt tire ve rakam koyulabilir. Büyük veya küçük harf olabilir ve program büyük-küçük harf duyarlıdır. Başka sembol ve boşluk koyulamaz.
- C anahtar kelimeleri değişken adlarında kullanılamaz.

Şimdi size geçerli örnek değişken adlarını verelim.

sayi, Sayi, Sayi5, _sayi, _Sayi, SAYI_1, __sayi__1 , NotOrtalamasi, _not_ortalamasi, NOT_ORTALAMASI, NOTORTALAMASI

Şimdi ise bu değişken adları arasından güzel olanları gösterelim.

NotOrtalamasi, not_ortalamasi, _sayi_1, sayi_1

Güzel değişken adı kullanmak acemi programcıların hemen yapabileceği bir iş değildir. Zaman geçtikçe ve kendi programlama tarzınızı oturtunca kendiliğinden gelişen bir olaydır. Siz programlama yaparken başka örnek kodları, kütüphaneleri ve programlama arayüzlerini kullandığınız için ister istemez dünyadaki programcıları metodunu öğrenirsiniz ve buna uygun kod yazmaya başlarsınız.

Fonksiyon Prototipi Oluşturma

Yukarıdaki kurallara uygun olarak fonksiyon adını belirledik fakat bunu önce fonksiyon prototipi oluşturarak derleyiciye tanıtmamız gerekiyor.

Fonksiyonların değer alıp almaması ve değer döndürüp döndürmemesi de bu fonksiyon prototipini oluştururken belirlenen özelliklerdir. Şimdi bir fonksiyon prototipi yapısını inceleyelim.

```
float karekok (int sayi);
```

Burada programcı bir fonksiyon tanımladı. Bu fonksiyonun bütün özelliklerini burada görmemiz mümkündür. float dediğimizde fonksiyonun float tipinde değer döndürdüğünü anlıyoruz. karekok ise programcı tarafından verilmiş isimdir ve fonksiyonun yaptığı iş hakkında haber verir. (int sayi) diyerek parantez içine yazdığımız değer ise fonksiyonun aldığı argüman olup fonksiyonu çağırırken içine yazacağımız değer tipini bize göstermektedir. Yani bu fonksiyon int tipinde bir değer alıp işler ve float olarak bize geri getirir. Bu fonksiyon prototipi int main fonksiyonundan önce programa yazılır. int main fonksiyonundan sonra fonksiyonumuzu yazacaksak prototip oluşturmaya ihtiyacımız vardır. Kısacası fonksiyon prototipi şu şekil olmalıdır.

```
döndüdüğü_deger fonksiyon_adı (argüman1, argüman2, argüman3 ...)
```

Fonksiyonların tek bir değer döndürdüğünü fakat pek çok argüman aldığını unutmayalım. Şimdi gerçek karekok fonksiyonumuza bir bakalım.

```
float karekok (int sayi)
{
    komutlar;
    komutlar;
    komutlar;
    return karekok_degeri;
```

```
}
```

Görüldüğü gibi fonksiyon `int main()` fonksiyonundan ayrı bir kapsam içerisinde. `int main()` fonksiyonunda bu fonksiyon `karekok(20);` diye çağırıldığında fonksiyon `int sayi = 20` şeklinde kendi sayı değişkenine 20 değerini ekleyecektir. Sonrasında çeşitli işlemleri yaptıktan sonra herhangi bir değişkeni veya değeri geri döndürecektir. O yüzden biz programda `deger = karekok(20);` diye yazarak geri dönen değeri bir değişkene aktarırız. Fonksiyon çağırıldığında `main` fonksiyonunda işletilen program bırakılır ve buradaki kodlar işletilir. Burada iş bittikten sonra tekrar kalındığı yerden devam eder. **return** deyimi fonksiyonlar için döndürecekleri değeri belirlemektedir. `return` işletildiği zaman fonksiyondan operand olarak belirtilen değer ile beraber çıkarılır.

Artık fonksiyonların mantığını anlattığımıza göre örnekler üzerinden daha rahat inceleme fırsatı bulacağız. Bir sonraki yazıda hem değer alan ve hem de değer döndüren fonksiyonları yazacağız ve deneyeceğiz.

-25- Değer Alan Fonksiyonlar, Sabitler, Önek ve Sonekler

Önceki yazımızda fonksiyonlara giriş yaptık ve basit fonksiyonların nasıl kullanıldığını size anlattık. Şimdi ise fonksiyonları daha işlevsel hale getirmek adına fonksiyonlara değer aktaracağız ve değer alan fonksiyonların işleyişini anlatacağız. Önceki yıldız yazdırma örneğinde fonksiyon tek bir sayıda ve tek bir karakteri yazdırıyordu. Bu fonksiyonun ayarlanabilir olmadığı anlamına gelmektedir. Yani küçük bir değişiklik yapmak istiyorsak örneğin yıldız yerine “-” yazdırmak istiyorsak başka bir fonksiyon yazmamız gerekecektir. Bunu önlemek ve tek bir fonksiyonda başka başka işler yaptırmak için yani fonksiyonun esnekliğini artırmak için fonksiyonlara argüman değer aktarırız ve bu değerlere göre işlem yaparız.

İlk vereceğimiz örnekte argüman almayan bir fonksiyonun parametreleri ile nasıl oynayabileceğimizi anlatacağız. Bu pratik bir yöntem olmasa da alternatif bir çözüm olarak kullanılabilir. Biz de öğrenmeniz ve argümanların önemini kavramanız adına öncelikle bu örneği veriyoruz. Örneğimiz yıldız yazdırma fonksiyonu ile aynı görevi yapıyor fakat biz programlarken istediğimiz karakteri istediğimiz sayıda yazdırmayı global değişkenlerin değerleri ile oynayarak yapmaktayız.

```
#include <stdio.h>
#include <stdlib.h>
//////// FONKSİYON PROTOTİPLERİ //////////
void karakter_yaz(void);
//// GLOBAL DEĞİŞKENLER ////
int karakter_sayisi = 30;
```

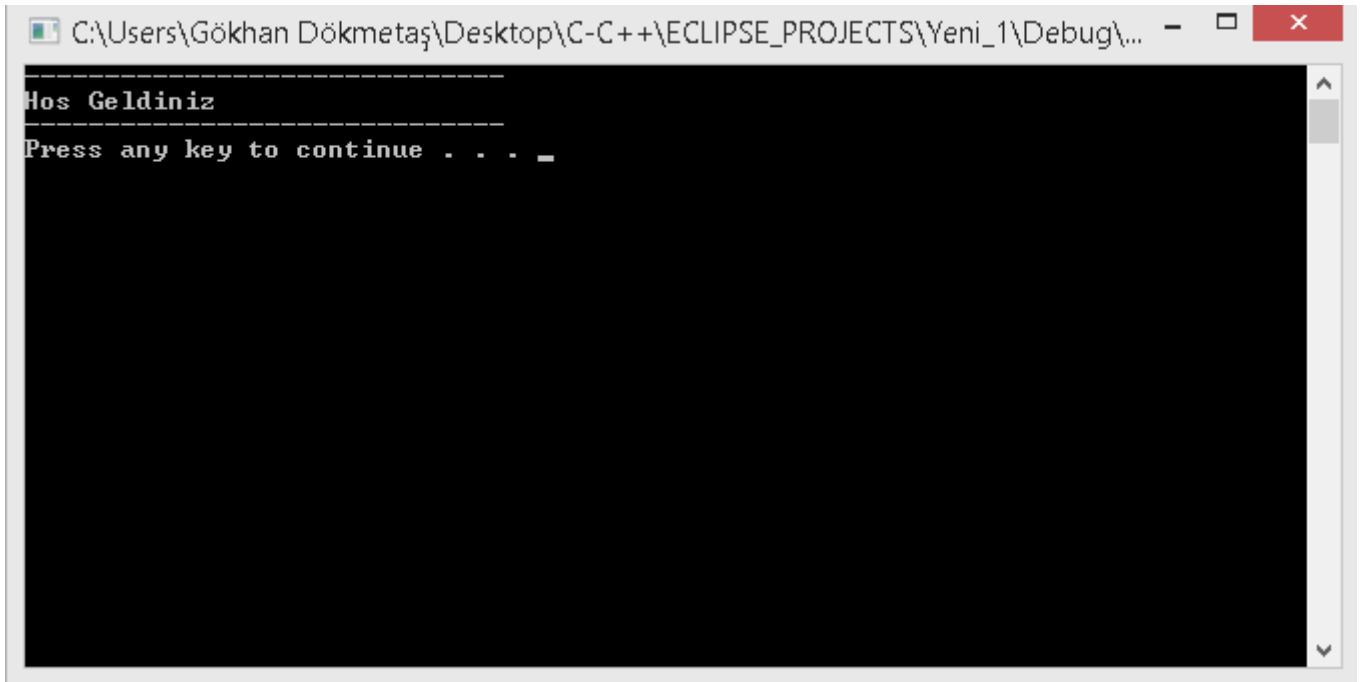
```
char karakter = '-';
int main() {

    karakter_yaz();
    puts("Hos Geldiniz");
    karakter_yaz();

    system("PAUSE");
}

void karakter_yaz (void)
{
    for (int i=0; i < karakter_sayisi; i++ )
        printf("%c",karakter);
        printf("\n");
}
```

Programı çalıştırdığımızda ekran görüntüsü şu şekilde olacaktır.

A screenshot of a Windows console window. The title bar shows the file path: C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\Yeni_1\Debug\... The console output is as follows:

```
-----  
Hos Geldiniz  
-----  
Press any key to continue . . . _
```

Burada karakterlerin kaç defa yazdırılacağı ve hangi karakter yazdırılacağı global değişkenlere atadığımız değerler yardımıyla belirlenmiştir. Global değişkenler C programında bir dosya içerisinde bütün fonksiyonlar tarafından erişilebilen ve bütün fonksiyonların değerini okuyup değiştirdiği değişkenlerdir. Normalde bir fonksiyon içerisinde tanımlanan değişken “kapsam” içerisinde kalmaktadır ve diğer fonksiyonlar tarafından erişilememektedir. Bu değişken tipine yerel değişken adı verilir. Bir fonksiyon içerisinde tanımlanan yerel değişken o fonksiyon kapsamından çıkıldığı zaman silinmektedir ve kısa bir süreliğine hafızada tutulmaktadır. Global değişken ise program başında tanımlanmakta ve program boyunca varlığını sürdürmektedir. O yüzden global değişkenleri çok fazla tanımlamak hafızada gereksiz yere yer kaplanmasına sebep olacak ve program güvenliğine olumsuz etki edecektir. Çünkü her fonksiyon tarafından erişilebilen bir değişkenin değerini kaza ile değiştirmemiz olasıdır. Kapsamlar değişkenlerin değerini güvenceye almaktadır. Aşağıdaki kod blokunda global değişkenlerin nasıl tanımlandığını görebilirsiniz.

```
int karakter_sayisi = 30;
char karakter = '-';
```

```
int main() {
```

Görüldüğü gibi main fonksiyonundan önce ve main fonksiyonunun dışında tanımlanır. main ana program fonksiyonu olsa da main programı içinde tanımlanan bir değişkene başka bir fonksiyonun erişme imkanı yoktur. Biz fonksiyonlara değer aktarmak istediğimizde bu yüzden argüman olarak değer aktarırız ve bu kapsam sınırlamasında kontrollü bir şekilde veri aktarma imkanı buluruz. Şimdi aynı fonksiyona değer aktarmayı sabitler ile yapalım.

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

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

```
//////// FONKSİYON PROTOTİPLERİ //////////
```

```
void karakter_yaz(char karakter, int sayi);
```

```
////// GLOBAL DEĞİŞKENLER //////////
```

```
int main() {
```

```
    karakter_yaz('*', 50);
```

```
    puts("Hos Geldiniz");
```

```
    karakter_yaz('*', 50);
```

```
    system("PAUSE");
```

```
}
```

```
void karakter_yaz (char karakter, int sayi)
```

```
{
```

```
    for (int i=0; i < sayi; i++ )
```

```
printf("%c", karakter);  
printf("\n");  
}
```

Burada fonksiyon prototipinden itibaren bir takım değişiklikleri yaptık. Önce fonksiyon prototipine bakalım ve ne değişiklik yaptığımızı görelim.

void karakter_yaz(char karakter, int sayi);

Burada fonksiyon parantezleri arasında iki değişken tipi tanımladık ve virgülle birbirinden ayırdık. Burada demek istediğimiz fonksiyonun ilk argümanı char tipinde, ikinci argümanı ise int tipinde değer olacağıdır. Buradaki değişken isimleri main fonksiyonunda da yer alabilir fakat kapsam dışında olduğu için yinelemesi sıkıntı olmayacaktır. karakter veya sayı diyebileceğimiz gibi istediğimiz ismi ekleyebiliriz ve virgül ekleyerek daha fazla argüman ekleyebiliriz.

karakter_yaz('*', 50);

Ana programda fonksiyonu çağırırken bu şekilde çağırdık. Fonksiyon argüman alıyorsa aldığı argüman kadar parantez içerlerine argümana uygun değer yazılması gereklidir. İlk argüman char karakter olduğu için karakter sabiti olan '*' değerini yazdık. Sonrasında int sayı dediği için 50 değerini yazdık. Artık bizim yazdığımız '*' değeri char karakter değişkenine, 50 değeri ise int sayı değişkenine aktarılacaktır.

Sabitler

Şimdiye kadar sabit değerleri kullansak da bunları özel olarak ele almadık. Sabit değer konusu çok kısa olduğu için konunun arasında size anlatmak istiyoruz. Biz bir değer belirtmek için değişken kullanmadığımız zaman sabit

değerleri kullanırız. Bu sabit değerler 50, 400 gibi onluk tabanda sayı değerleri olduğu gibi 'a' şeklinde karakter değeri de olabilir. Fark ederseniz bazı sabitleri ifade etmek için farklı kurallar vardır. Bu kuralları bilmeden bir değeri ifade ettiğimizde bazen derleyici hatası çoğu zaman da programlama hatası ile karşı karşıya kalırız. Örneğin tek bir karakteri ifade ettiğimiz tek tırnak ifadesiyle 'Merhaba Dünya' yazarsak derleyici buna hata vermese de değişkene tek bir karakter değerini aktaracaktır. O yüzden bu sabit değerleri kullanmayı iyi bilmemiz gereklidir.

Sabit değerleri ifade ederken örnek ve sonek adı verilen bileşenler kullanılmaktadır. Öncelikle size örnekleri anlatalım.

Önekler

Bizim dilimiz sondan eklemeli bir dil olduğu için örnekleri dilimizde yabancı ifadelerde kullanmaktayız. C programlama dilinde ise ön ekler normal dilde kelimelerin başına gelip anlamlarını değiştirmesi gibi değerlerin önüne gelerek bu değer tiplerini değiştirmektedir.

Onluk Taban : Desimal ifadelerde herhangi bir ön veya son eke gerek yoktur doğrudan sayıyı yazmakla ifade ederiz. Örneğin; 500, 1250, 20, -600.

Sekizlik Taban: Sekizlik tabandaki sayıları başına sıfır getirmekle ifade ederiz. Örneğin; 012, 05, 024.

Onaltılık Taban : Onaltılık tabandaki sayıları başına 0x ya da 0X getirmekle ifade ederiz. Hatta pek çok dijital elektronik ve mühendislik kitabında C diliyle alakası olmasa da onaltılık sayıların başına 0x getirilmekle ifade edildiğini görebilirsiniz. Biz de C ile alakasız yazılarımızda onaltılık sayıları böyle ifade ediyorduk. Örnekler; 0xFF, 0x12, 0xff55, 0XFF.

İkilik Taban : İkilik tabandaki sayıları ifade ederken başına 0b ya da 0B getiririz. Örnekler; 0b110010, 0B1111100.

Son Ekler

Önekler yukarıda görüldüğü gibi temel veri tiplerini tanımlamak için kullanılıyordu. Assembly dilinde de mevcut olan bu temel veri tipleri hemen hemen aynı önekler vasıtasıyla ifade edilmektedir. Sonekler ise C diline mahsus değişkenleri ifade etmek için kullanılmaktadır.

int : Integer tipindeki bir değeri ifade etmek için herhangi bir ek kullanılmaz. Doğrudan sayı yazılır. Örnekler; 500, 200.

unsigned int : İşaretsiz integer tipindeki değeri ifade etmek için u ya da U eki kullanılır. Örnekler; 500u, 6000U.

long int : long int yani uzun tamsayı tipindeki değeri ifade etmek için l ya da L eki kullanılır. Örnekler; 500l, 800L.

unsigned long int : İşaretsiz uzun tamsayı tipindeki değeri ifade etmek için ul ya da UL eki kullanılır. Örnekler; 5222UL, 200000ul.

long long int : çok uzun tam sayı olan long long int tipinde değeri ifade etmek için ll ya da LL eki kullanılır. Örnekler; 500000000ll, 20000000LL.

unsigned long long int : İşaretsiz çok uzun tam sayı değeri olan unsigned long long int tipinde değeri ifade etmek için ull ya da ULL eki kullanılır. Örnekler; 50000ULL, 6545622456ull.

float : float tipinde ondalıklı ifadeleri kullanırken nokta ile sayıyı yazdıktan sonra f ekini kullanmamız gereklidir. Aksi halde derleyici bunu double olarak anlayacaktır. Örnekler; 3.522f, 25.22F.

double : double tipinde ondalıklı ifadeleri tam sayı değerinde olsa bile nokta ile yazmamız gereklidir. Örnekler; 2.0, 25.0005, -0.55E-7.

Sonek kullanımında en çok sorunu float tipinde sabitleri yazarken yaşarsınız. O yüzden nokta koyduğunuza her zaman emin olmanız gereklidir. Tip dönüşümlerini ileride anlattığımızda daha ayrıntılı olarak açıklayacağız.

Karakter sabiti : Karakter değerini ifade etmek istediğimizde değeri tek tırnak arasına yazarız. Örnekler; 'a', 'B'.

Karakter dizisi : Karakter dizisi çift tırnak arasına yazılır. Örnekler; "Merhaba Dünya!", "Lutfen Deger Giriniz.".

Yazımız yeterince uzadı bir sonraki başlıkta değer alan fonksiyonları anlatmaya devam edeceğiz.

-26- Fonksiyondan Değer Döndürmek

Buraya kadar argüman alan fonksiyonlardan size bahsettik fakat değişkenleri argüman alan fonksiyondan bahsetmedik. Kısa bir örnekle bunu size göstereceğiz ve ardından fonksiyondan değer döndürerek fonksiyonları tam anlamıyla kullanmış olacağız.

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

//////// FONKSİYON PROTOTİPLERİ //////////
void karakter_yaz(char karakter, int sayi);

int main() {
    char c;
    int n;
    printf("Yazilmasini istediginiz karakteri giriniz:");
    scanf("%c", &c);
    printf("\nKac Defa Yazilacigini giriniz:");
    scanf("%i", &n);

    karakter_yaz(c, n);
    system("PAUSE");
}

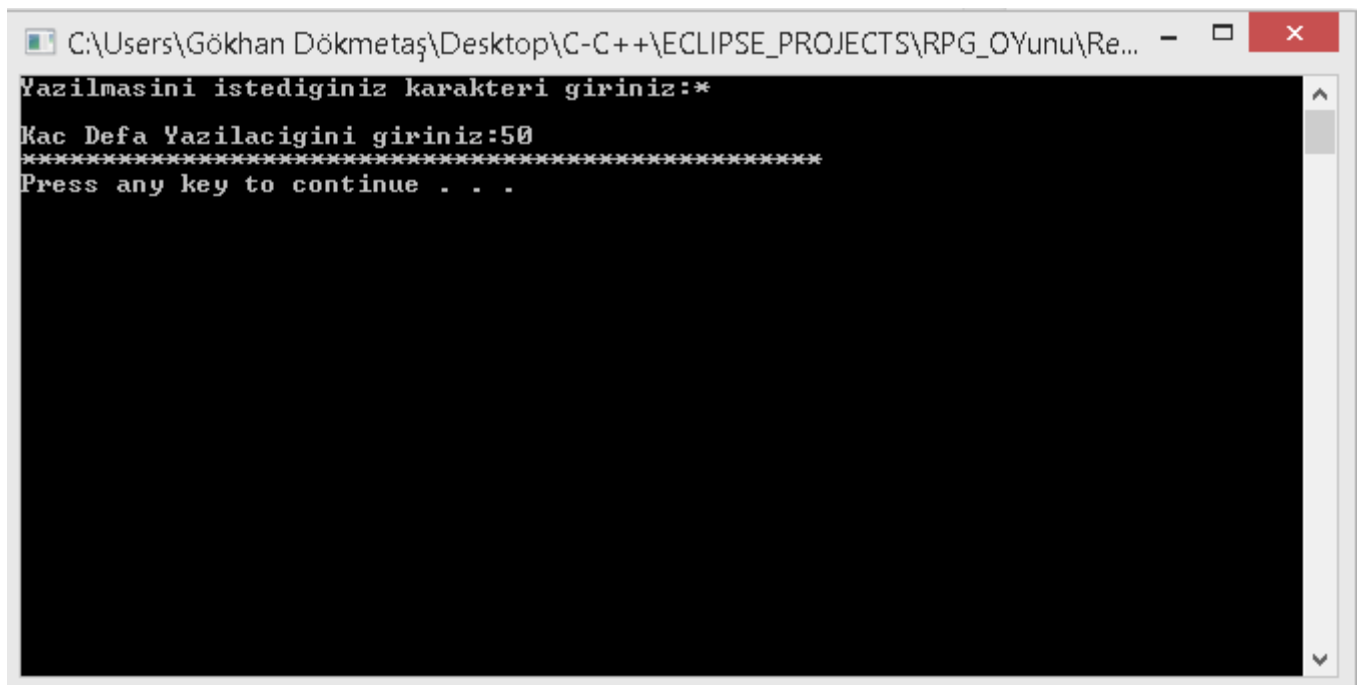
void karakter_yaz (char karakter, int sayi)
{
    for (int i=0; i < sayi; i++ )
```

```
printf("%c", karakter);  
printf("\n");  
}
```

Programı çalıştırdığımızda kullanıcıdan yazılması istenilen karakteri girmesi istenecektir. Bu durumda klavyedeki harf ve sembollerden birini girmemiz lazımdır. Bu sembol c değişkeninin içerisine atılacaktır. Sonrasında ise kaç defa bu harfin yazılacağı sorulmaktadır. Buraya girdiğimiz sayı değişkenin değerince önceden yazdığımız harf yazılacaktır.

karakter_yaz(c, n);

Bu komutla önceden kullandığımız fonksiyon üzerinde değişiklik yapmadan buna sabit değer atamak yerine değişken atadık. c değişkenini char tipinde ve n değişkeninin int tipinde olduğunu unutmayalım. Eğer birbirinden farklı tipte değişken atarsak hata verecektir. Programın çıktısı şu şekilde olabilir.



The screenshot shows a Windows command prompt window titled "C:\Users\Gökhan Dökmetaş\Desktop\C-C++\ECLIPSE_PROJECTS\RPG_OYunu\Re...". The prompt displays the following text: "Yazilmasini istediginiz karakteri giriniz:*. Kac Defa Yazilacigini giriniz:50". Below this, a line of asterisks is shown, followed by "Press any key to continue . . .". The window has a standard Windows title bar with minimize, maximize, and close buttons.

Değer döndüren fonksiyonlardan bahsetmeden önce C kütüphanesinde hazır olan bir değer döndüren fonksiyonu deneyelim ve ardından bunun benzerini

kendimiz yazalım. `pow()` fonksiyonu üs alma fonksiyonudur ve `math.h` kütüphane dosyasında bulunur. Eğer standart kütüphaneleri içeren bir derleyici kullanıyorsanız kullandığınız platform fark etmeksizin aynı fonksiyonları siz de kullanabilirsiniz. `math.h` başlık dosyasındaki diğer fonksiyonları konunun dağılmaması için buraya almayacağım. Öncelikle `pow()` fonksiyonunun referans yapısına bir bakalım.

```
double pow (double base, double exponent);
```

Burada `pow` fonksiyonunun iki adet `double` tipinde argüman aldığını ve yine `double` tipinde değer döndürdüğünü görüyoruz. Bu fonksiyonu kullanmadan önce referansından bu fonksiyon prototipine bakarız ve programımızda buna uygun değerleri yazarız. `double base`'den maksat taban, `double exponent` ise üs değeridir. Şimdi bu fonksiyonu kullandığımız bir programı yazıp deneyelim.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main() {
    double n;
    double p;
    printf("Ussunu Almak istediginiz karakteri giriniz:");
    scanf("%lf", &n);
    printf("\nKac Us Alinacagini giriniz:");
    scanf("%lf", &p);
    double sonuc = pow(n,p);
    printf("\n Sonuc: %.0lf \n", sonuc);
    system("PAUSE");
}
```

Öncelikle `#include <math.h>` diyerek matematik fonksiyonlarının olduğu kütüphane dosyasını çağırdık. Önceden aritmetik operatörleri anlatırken C dilinde matematik işlemlerinin sadece dört işlem ve mod almadan ibaret olduğunu görmüştünüz. Bu C diline mahsus bir özellik değildir aslında. Çoğu mikroişlemcinin komutlarına baktığımızda aritmetik işlemlerin dört işlemden ibaret olduğunu görürüz. Yani mikroişlemcilerde üs ya da karekök alacak bir komut yoktur. Böyle bir komutun olmaması bu işlemleri yapamayacağımız anlamına gelmez. Biz dört işlemi kullanarak programcılık yeteneği ile çok karmaşık matematik işlemlerini bile yaptırabiliriz. Burada da biz bir fonksiyon kullanarak üs alma işlemini yapıyoruz.

double n; ve double p; diyerek `pow()` fonksiyonuna gönderilecek iki argüman değişkeni tanımlamış olduk. Bunların `double` olmasının önemini önceden size anlattık. Fonksiyonun aldığı değerlerin tiplerine uygun değer atamamız gereklidir.

double sonuc = pow(n,p);

Burada öncelikle `pow` fonksiyonuna `n` ve `p` değişkenlerinin değeri aktarılır ve fonksiyon blokuna program akışı atlar. Burada `n` ve `p` değişkenleri `pow()` fonksiyonunun içinde farklı bir adda adlandırılabilir. Aradaki tek alaka bu değişkenlerinin değerinin fonksiyon değişkenine aktarılmasıdır. Fonksiyon argüman olarak aldığı değişkenler üzerinde doğrudan bir değişiklik yapamaz. `pow` fonksiyonu komutları işletip bize `float` tipinde değer verecektir. Bu değeri kullanabilmek için `double sonuc` adında bir değişken tanımladık. Bu değişkenin ilk değeri de `pow` fonksiyonundan dönen sonuç olacaktır.

Şimdi programın örnek bir çıktısını inceleyelim.

"C:\Users\Laptop\Desktop\ECLIPSE PROJECTS\eee\bin\Debug\eee.exe"

Ussunu Almak istediginiz karakteri giriniz:50

Kac Us Alinacagini giriniz:2

Sonuc: 2500

Press any key to continue . . .

Biz burada sonuç değişkenini double değil de integer olarak tanımlasaydık sonuç ekranında 2500 yerine 2499 yazacaktı. Doğru değişken tanımlamanın önemini bir daha görmüş oluyoruz.

Şimdi kendi pow() fonksiyonumuzu yazalım. Bu pow fonksiyonu double değil int tipinde değer alacak. Bu yönden tam sayılar için üs alma fonksiyonu işini yapacak.

```
#include <stdio.h>
#include <stdlib.h>
int us_al (int taban, int us);
int main() {
    int n;
    int p;
    printf("Ussunu Almak istediginiz karakteri giriniz:");
```

```

scanf("%i", &n);
printf("\nKac Us Alinacagini giriniz:");
scanf("%i", &p);
int sonuc = us_al(n,p);
    printf("\n Sonuc: %i \n", sonuc);
system("PAUSE");
}

int us_al (int taban, int us)
{
    int sonuc=1;
    while ( us != 0)
    {
        sonuc *= taban;
        us--;
    }
    return sonuc;
}

```

Burada double değerleri ile yapsaydık aynı pow() fonksiyonunun yaptığı işi gerçekleştirebilirdik. Yine de C dilinin standart kütüphane fonksiyonları yerine sizin kendi fonksiyonunuzu yazmanız tavsiye edilmez. Çünkü o kütüphanelere eklenen fonksiyonlar olabildiğince optimize edilmiş fonksiyonlardır ve sizin onlardan daha iyi fonksiyon yazmanız pek mümkün değildir. Biz burada gösterme adına pow fonksiyonunun taklitini yaptık ve nasıl değer döndürdüğünü gösterdik.

-27- Diğer Fonksiyon Konuları ve Recursive Fonksiyonlar

C dilinde fonksiyonlara dair anlatacağımız iki konu kaldı. Bunlar iç içe fonksiyonlar ve özyinelemeli (recursive) fonksiyonlardır. Eğer C++ dilini anlatsaydık fonksiyonlar hakkında daha çok bilgiyi anlatacaktık. Çünkü C++ dilinde fonksiyon konusuna ek özellikle getirilmiştir. Fakat C dilinde öğrendiğimiz bilgilerin neredeyse tamamını C++ dilinde de kullanabileceğimiz için biz hiç C++ konularına değinmeden C üzerinden konumuza devam edeceğiz. İleride C++, Visual C++ ve masaüstü yazılım geliştirme konularına geldiğimizde iyi ki önceden C öğrenmişim diyeceksiniz. Çünkü C++ kapsamlı, karmaşık ve yoğun bir dildir. C öğrenmekle sade bir dil üzerinden programlama mantığını öğrenirsiniz fakat ilk seferde C++ öğrenmekle bütün ayrıntıyı öğrenmek zorunda kalırsınız. Bu da kafanızı karıştırmaya yeterli olur.

İç İçe Fonksiyonlar

İç içe fonksiyonlar prensipte oldukça basittir. Fonksiyon çağrısı içerisinde fonksiyon çağırmak kadar basit olan bu yöntem ile kodlarınızı daha kısa yazma imkanı bulursunuz ve fonksiyondan dönen değerleri doğrudan argüman olarak aktarırsınız. Daha önceki yazıdaki örnekte yaptığımız üs alma fonksiyonunu şimdi printf içerisinde çağıralım ve bir satır koddan tasarruf edelim.

```
#include <stdio.h>
#include <stdlib.h>
int us_al (int taban, int us);
int main() {
    int n;
    int p;
    printf("Ussunu Almak istediginiz karakteri giriniz:");
    scanf("%i", &n);
```



```
printf("\nKac Us Alinacagini giriniz:");
scanf("%i", &p);
//int sonuc = us_al(n,p);
    printf("\n Sonuc: %i \n", us_al(n, p));
system("PAUSE");
}
```

```
int us_al (int taban, int us)
{
    int sonuc=1;
    while ( us != 0)
    {
        sonuc *= taban;
        us--;
    }
    return sonuc;
}
```

```
printf("\n Sonuc: %i \n", us_al(n, p));
```

Bu printf fonksiyonu işletilmeden önce parantez içindeki aritmetik ve mantık işlemlerinin öncelikle yapıldığını biliyorduk. Aynı zamanda parantez içerisinde çağırılan fonksiyonlar da önce çalıştırılmakta sonra da geri dönen değer argüman halinde printf fonksiyonuna gitmektedir. Bu C dilinin esnek özelliklerinden biridir. Eğer yazdığınız kodu bu şekilde anlamakta zorlanıyorsanız bunu kullanmayabilirsiniz. Daha yüksek seviye geliştirme ortamlarından farklı olarak C dilinde kod sadeleştikçe okunabilirliği

azalmaktadır. Okunabilirliğin artmasına yönelik yaptığımız çalışmalar kodu kalabalıklaştırmakta ve performansı düşürmektedir.

Özyinelemeli fonksiyonlar (Recursion)

Yukarıda fonksiyon çağırırken parantezler içerisinde farklı bir fonksiyonu çağırmıştık. Peki bu fonksiyon fonksiyon kendi içerisinde kendisini çağırırsa ne olur dersiniz? Muhtemelen program sürekli aynı fonksiyonu çağırır ve sonsuz döngüye girer veya derleyici hata verir, diye bir cevap vermeniz muhtemeldir. Fakat C dilinde bu esneklik vardır ve bir fonksiyon kendi içerisinde kendini çağırabilir. Elbette bu tehlikeli bir yöntemdir ve acemi programcıların kolay kolay bulaşmaması lazımdır. Fakat biz bir kod veya kütüphane incelediğimizde fonksiyonun kendisini çağırdığını gördüğümüz vakit “Bu da neyin nesi?” dememek için başlangıçta öğrenmemiz lazımdır. Bu yöntemi bazı matematik ve veri işleme fonksiyonlarında kullanmanız ise oldukça etkili olacaktır. Özyinelemeli fonksiyonlar ciddi derecede bellek tasarrufu sağlamaktadır.

Özyinelemeli fonksiyonları şu şekilde özetlemek mümkündür.

```
void oz_yineleme()  
{  
    komutlar;  
    komutlar;  
    komutlar;  
    oz_yineleme();  
    komutlar;  
}
```

```
int main()
{
    oz_yineleme();
}
```

Herhangi bir döngü veya karar mekanizmasına tabi tutmadan şartsız bir şekilde böyle bir programı çalıştırmak oz_yineleme() fonksiyonu çağırıldığı andan itibaren fonksiyon kendini fonksiyon blokunda sürekli çağırarak ve sonsuz bir döngü halinde blok başı ile kendi arasında olan komutları işletip duracaktır. Ama biz aynı komutu şu şekilde yazarsak durum nasıl olur bir bakalım.

```
void oz_yineleme()
{
    komutlar;
    komutlar;
    komutlar;
    if ( sart == TRUE )
        return 1;
    oz_yineleme();
}
```

```
int main()
{
    oz_yineleme();
}
```

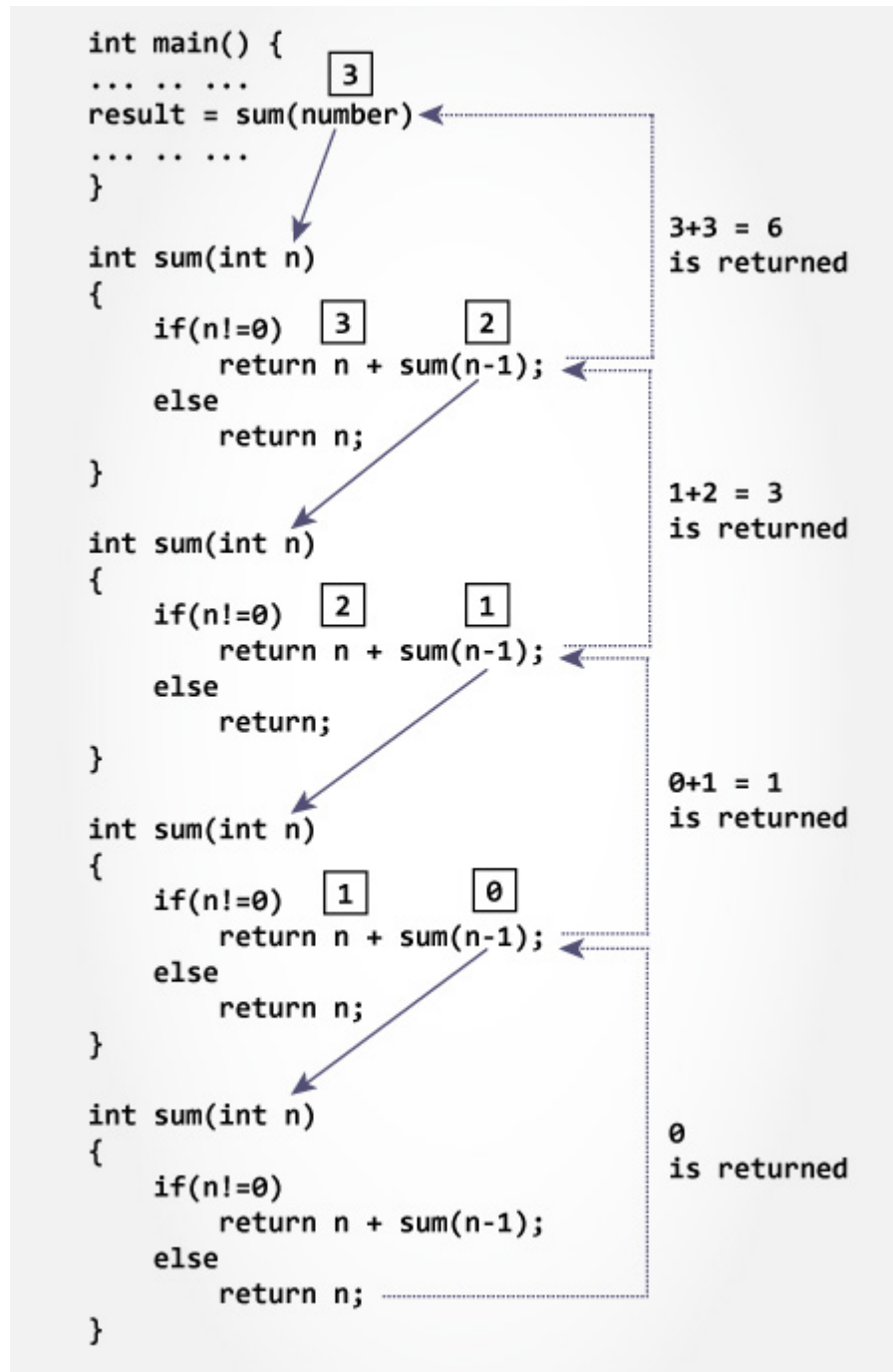
Burada oz_yineleme() fonksiyonu artık kendini sonsuz döngüye sokmuş olsa da biz buraya bir şart ekleyerek return komutu ile döngüden çıkma fırsatı

tanıdık. Eğer bu şart yapısı eksiksiz olmazsa ve bir noktada hata çıkabilirse program sonsuz döngüye girecektir. Biz burada önce programı sonsuz döngüye sokuyoruz ve sonra sonsuz döngüden çıkarma çözümünü arıyoruz. Şimdi örnek bir fonksiyonu inceleyerek ne kadar derine inebildiğini görelim.

```
int toplam(int sayi)
{
    if (sayi!=0)
        return sayi + toplam(sayi-1); // toplam() kendini
        çağırır
    else
        return sayi;
}
```

Burada ilk satırdan itibaren fonksiyonun basit bir fonksiyon gibi durduğunu görüyoruz. sayi adında bir değer alıyor ve adından da bu sayıya kadar olan sayıların toplamını verebileceğini tahmin edebiliyoruz. Defalarca toplama işlemi yapması için normalde döngüye ihtiyaç olsa da burada döngü yerine tek bir karar yapısı görüyoruz. Eğer sayı 0 değilse return komutu ile fonksiyondan çıkılıyor fakat return'dan önce bir dizi işlem yapılmış. Bu işlemler sayi + toplam(sayi-1) olunca kafamız oldukça karışabilir. sayi + 'ya kadar olan kısmı anlasak da sayıyla neyin toplanacağı konusunda kafamız karışıyor. Bu işlem yapılmadan ve return ile fonksiyondan çıkışmadan önce toplam(sayi-1) ile fonksiyon bu sefer argüman olarak sayi değişkeninin bir eksiğini alıyor ve fonksiyon içindeyken yine çağırılıyor. Bu sefer sayi değeri bir eksik olsa da karşımıza tekrar aynı if yapısı çıkıyor ve aynı şartı sağlarsa yine bir defa çağırılıyor. Bu sefer 2. fonksiyona göre sayi-1 olsa da birinci fonksiyona göre gönderilen argüman sayi-2 oluyor. Böylece alt alta sayı miktarınca fonksiyon çağırıldıktan sonra bütün sayı değerlerinin sayi ile toplamını elde ediyoruz.

Evet, bahsettiğimiz konu oldukça kafanızı karıştırırsa da bu normaldir. Benim de bunu anlamam uzun sürmüştür. Bunu anlayabilmek için C dili mantığını kafanıza oturtmanız ve pratik yapmanız gereklidir. Sadece bunun üzerinde değil fonksiyonlar üzerinde pratik yaptığınızda ve C dilinin esnekliğini anladığınızda bunu rahatça kavrarınız. Sizin için daha anlaşılır olması için bir resim vereceğim.



Resim: https://cdn.programiz.com/sites/tutorial2program/files/natural-numbers-sum-recursion_0.jpg

Karar yapılarını anlayabilmenin göstergesi kümelenmiş else if yapılarını etkin kullanabilmek olduğu gibi fonksiyonları tam olarak anlamanın göstergesi de özyinelemeli fonksiyonları anlamaktır. Biz pratikte çok fazla kullanmadığımız için yukarıda verdiğimiz örnek kod ile yetiniyoruz. Eğer bir kitap yazsaydım elbette daha ayrıntılı açıklardım fakat bu yazıların kitap kalitesinde olması için çabalasam da kitap yazacak kadar çok zaman bulamıyorum.

-28- Diziler

Biz günlük hayatımızda bu enzer nesneleri gruplara ayırırız ve bir arada toplarız. Örneğin yumurtalar kolide, meyve ve sebzeler kasada krakerler bir pakette toplanır. Bunun gibi öğrencileri de sınıflarda toplamaktayız. Daha önceden günlük hayatta karşılaştığımız durumları bilgisayar programına aktarma yolunu incelemiştik. Fakat şimdiye kadar böyle bir durumu bilgisayar programına nasıl aktaracağımızı anlatmadık. Bu yeni durumu yani benzer nesnelerin gruplaşmasını bilgisayar ortamına diziler vasıtasıyla aktarırız.

Şimdi bir sınıf düşünelim. Sınıfta 30 öğrenci var ve bu öğrencilerin ortak bir noktası var. Birincisi bu öğrenciler bir yerde topluluk oluşturmuş ve gruplaşmış. Buna biz günlük hayatta 12-B, 5-A sınıfı gibi adlar veriyoruz. Ayrıca sınıftaki öğrenciler aynı sınıfta olsa da numaralar ile birbirinden ayrılmış durumdadır. İstersek sınıfı tamamen ele alabiliriz istersek de numara numara öğrencileri çağırabiliriz. Daha önce öğrendiğimiz bilgilerle bir sınıf programı yazmak istesek şöyle olacaktır.

```
int ogrenci_1;  
int ogrenci_2;  
int ogrenci_3;  
int ogrenci_4;
```

Burada int sınıfından dört adet değişken tanımlasak ve öğrencileri böyle ifade etsek de bu öğrenciler arasında herhangi bir bağlantı yoktur. Tamamen bağımsız olarak ele alınabilir. Bağlantı sadece isim benzerliğinden ibaret olup bilgisayarın anlayacağı bir konu değildir. Bilgisayar bizim değişkenlere ne ad verdiğimiz umursamaz. Bu öğrenciler arasında bir bağlantı kurmak için bunları bir sıraya koymamız gereklidir. Bunun için ise şöyle bir komut yazarız.

```
int ogrenciler [4];
```

Burada yine int tipinde dört adet değişken tanımlasak da bunlar birbirinden bağımsız değil belli bir sıraya dizilmiş dört adet değişkendir. Birbiriyle alaka kurmaları ise hafızada adres olarak sıraya dizilmeleri kadar erişim operatörü vasıtasıyla da olmaktadır. Erişim operatörü dediğimiz [] operatörü arasına yazdığımız değer ile dizinin verisine erişmiş oluruz. Biz dizi tipinde değerleri MS Excel programında da sıkça kullanmaktayız. Oradaki tablo aslında iki boyutlu bir diziden farklı değildir. Bir tarafta harf ile adlandırılmış sütunlar ve diğer tarafta sayı ile adlandırılan satırlar bulunmaktadır. Biz de yukarıda tanımladığımız dizi değişkenini tablo halinde şu şekilde ifade edebiliriz.

ogrenciler[0]	ogrenciler[1]	ogrenciler[2]	ogrenciler[3]
Değer	Değer	Değer	Değer

Diziler birkaç elemandan on binlerce elemana kadar veriyi tutabilir. Dizi değişkenini tanımlarken öncelikle bir değişken tipini belirlememiz gereklidir. Dizinin elemanları farklı farklı tiplerde olamaz. Böyle bir durumda yapı adı verdiğimiz daha karmaşık veri yapıları kullanılır. İlerleyen konularda bundan bahsetsek de şimdilik tek tipte dizi değerlerini anlamamız yeterlidir. Biz dizi değişkeni tanımlarken şu söz dizimini takip ederiz.

```
değişken_tipi değişken_adı[boyut];
```



```
// Örnekler
```

```
int sayi[100];  
char karakter_dizisi[50];  
float ondalikli_sayilar[20];
```

Biz dizi tanımlarken öncelikle dizi tipini belirleriz. Bu tam sayı, karakter veya diğer değişken tiplerinde olabilir. Sonrasında dizisinin adını değişken adı kurallarına bağlı olarak belirleriz ve sonrasında dizinin boyutunu belirleriz. Dizi boyutunu indeks operatörü yani [] arasına sayısal olarak yazarız. Bu noktada dizinin boyutunu doğru olarak belirlemek önemlidir çünkü program dizi boyutu kadar hafızada yer ayırmaktadır. Örneğin `int sayi[1000];` dediğimizde program 4 baytlık int değişkeni tipinde 1000 adet yeri hafızada ayıracaktır bu da yaklaşık 4 kilobaytlık bir alan kaplayacaktır. Gömülü sistemlerde çalışırken bu dizi ve yapılarla çalışmak daha da zor olmaktadır çünkü çoğu aygıt 4 kilobaytlık bir RAM hafızasına bile sahip değildir.

Yukarıda verdiğimiz öğrencilerle alakalı diziye biraz daha geliştirelim ve bir örnek yapalım. Bu sayede dizi değişkenlerinin yapısını ve bunlara erişmenin en temel yolunu göreceksiniz. Öncelikle sınıfı biraz büyütelim ve 8 kişiden oluşan ve bir adı olan sınıf yapalım. Her dizi değişkenine de bir öğrenci numarasını ekleyelim. Burada her değişkenimiz bir öğrenci olacak. Şimdi programımızı deneyelim.

```
#include <stdio.h>  
#include <stdlib.h>  
int main() {  
int _1a_sinifi_ogrencileri[8];  
_1a_sinifi_ogrencileri[0] = 254;
```

```
_1a_sinifi_ogrencileri[1] = 951;
_1a_sinifi_ogrencileri[2] = 253;
_1a_sinifi_ogrencileri[3] = 425;
_1a_sinifi_ogrencileri[4] = 12;
_1a_sinifi_ogrencileri[5] = 84;
_1a_sinifi_ogrencileri[6] = 365;
_1a_sinifi_ogrencileri[7] = 145;

for (int i = 0 ; i < 8; i++)
{
    printf("Ogrenci : %i Numarasi: %i \n", i ,
_1a_sinifi_ogrencileri[i]);
}
system("PAUSE");
}
```

Bu program dizi değişkenlerinin en temel ve en sık kullanımını bize göstermektedir. Öncelikle **int _1a_sinifi_ogrencileri[8];** diyerek 8 adet int değişkenini içeren ve toplamda 8 adet int genişliğinde **_1a_sinifi_ogrencileri** adında bir dizi değişkeni tanımladık. Burada tanımlamada erişim operatörünün dizinin genişliğini belirttiğini unutmayalım.

_1a_sinifi_ogrencileri[0] = 254;

Bu komutta ilk öğrencinin numarasını yazmaktayız. Unutmamamız gereken nokta dizi üyelerinin numarasının hep sıfırdan itibaren başladığıdır. Yani bizim tanımladığımız değişkenin genişliği 8 ise bu 0-7 arasında olduğu anlamına gelmektedir. [8] şeklinde bir kullanım olamaz.

_1a_sinifi_ogrencileri[1] = 951;

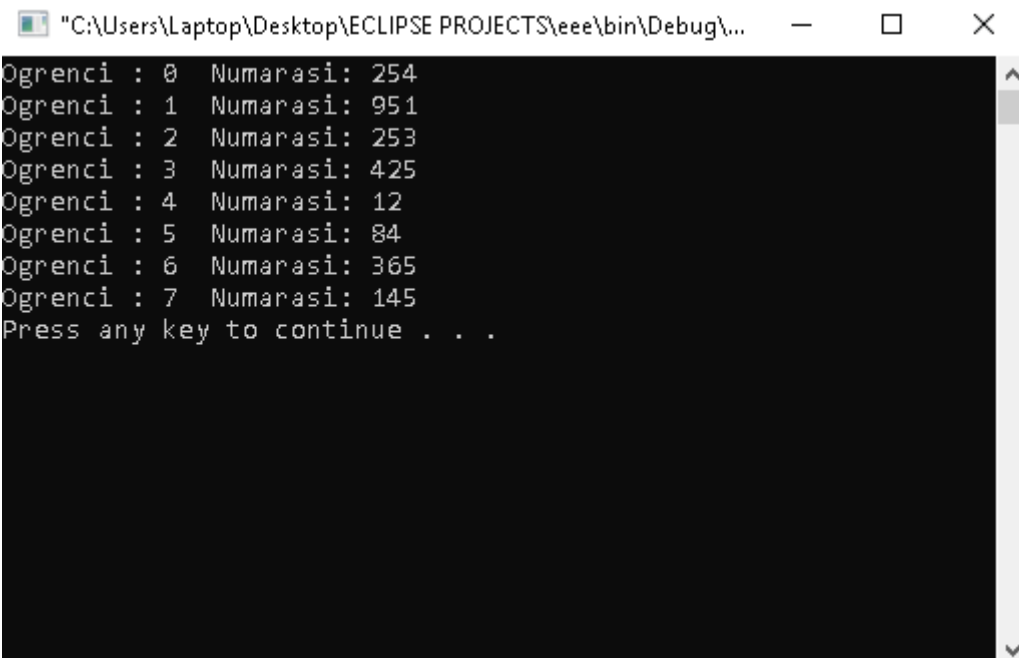
Burada hiçbirini eksik bırakmayarak bütün dizi üyelerine ilk değeri veriyoruz. Daha sonrasında bu değeri vermenin kısa yolunu da size göstereceğiz. Anlamanız için başlangıçta bu konulara değinmeden en sade yoldan değeri atamalarını yapıyoruz.

```
for (int i = 0 ; i < 8; i++)
```

Toplamda 8 kere çalıştırılacak bir döngü oluşturduk. Bu döngü değişkeni aynı dizi elemanlarının numaraları gibi 0'dan 8'e doğru gidiyor. Bu dizi elemanlarına sırayla erişmek için kullanılabilir.

```
printf("Ogrenci : %i Numarasi: %i \n", i , _1a_sinifi_ogrencileri[i]);
```

Bu komutta dizi elemanlarını sırayla yazdırmaktayız. Öncelikle öğrenci'nin sırası döngü değişkeni hem de dizi indeks değeri olarak yazdırılır. Sonrasında ise **_1a_sinifi_ogrencileri[i]** diyerek döngü elemanı erişim operatörünün içine yazılarak ilgili dizi üyesinin değerine erişilir ve bu ekranda yazdırılır. Programın çıktısı şu şekilde olacaktır.



```
Ogrenci : 0 Numarasi: 254
Ogrenci : 1 Numarasi: 951
Ogrenci : 2 Numarasi: 253
Ogrenci : 3 Numarasi: 425
Ogrenci : 4 Numarasi: 12
Ogrenci : 5 Numarasi: 84
Ogrenci : 6 Numarasi: 365
Ogrenci : 7 Numarasi: 145
Press any key to continue . . .
```

Dizileri tanımlarken aynı tipte ve birbiriyle alakalı elemanların bir araya getirileceğini unutmamak gerekir. Birbiriyle alakasız ve birbirinden bağımsız değerleri kaydederken değişkenleri kullanmamız lazımdır. Programcılıkta dizilerin öncülük ettiği “String” yani karakter dizileri programcılığın ana unsurlarından biridir. Karakter dizilerini C++, Java gibi dillerde String sınıfını kullanarak hazır fonksiyonlar üzerinden yapsak da C dilinde String sınıfı yoktur. O yüzden karakter dizilerini en temel yoldan üretmek gerekecektir. Ayrıca standart kütüphanede yer alan metin işleme fonksiyonları da vardır. Bunları daha sonraki konuda anlatacak olsak da bir sonraki yazımızda karakter dizilerinden başlayarak dizi konusunu daha ayrıntılı olarak ele alacağız.

Diziler -2-

Temel C programlamada diziler konumuza devam ediyoruz. Önceki yazımızda en temel dizi özelliklerinden size bahsetmiştik. Fakat diziler hem özellik bakımından hem kullanım alanı bakımından anlattığımız ile sınırlı değildir. Aslında işaretçiler ve yapılarda olduğu gibi dizileri de en basit haliyle kullanabildiğimiz gibi en karmaşık işlerde de kullanabiliriz. C, C++ gibi köklü programlama dillerinde hiçbir özelliğin giriş seviyesi veya basit olduğunu iddia edemeyiz. Bu özellikleri nasıl kullandığınız onun seviyesini belirlemektedir.

Dizileri anlatırken dizilere ilk değer atamayı size anlatmamıştık. İlk değeri atamak oldukça işlevsel bir özelliktir. Böylelikle satırlar boyu program yazmak zorunda kalmazsınız. Önceli programda dizi tanımlamasını ve dizi değişkenlerine ilk değeri atamayı şu şekilde yapmıştık.

```
int _1a_sinifi_ogrencileri[8];
_1a_sinifi_ogrencileri[0] = 254;
_1a_sinifi_ogrencileri[1] = 951;
_1a_sinifi_ogrencileri[2] = 253;
_1a_sinifi_ogrencileri[3] = 425;
_1a_sinifi_ogrencileri[4] = 12;
_1a_sinifi_ogrencileri[5] = 84;
_1a_sinifi_ogrencileri[6] = 365;
_1a_sinifi_ogrencileri[7] = 145;
```

Görüldüğü gibi oldukça kalabalık ve C'nin ruhuna uymayan yapıda bir program bloku değil mi? C dilini bu kadar övüyorsak C dili buna bir çözüm getirmek zorunda olması gerekir. Tahmin ettiğimiz gibi C dili buna bir çözüm getirmekte. Yukarıdaki kodun aynısını aşağıda şu şekilde yazabiliriz.

```
int _1a_sinifi_ogrencileri[8] = { 254, 951, 253, 425, 12, 84, 365, 145 };
```

Süslü parantez içinde sırayla atadığımız değerler sırayla dizi değişkenlerinin içerisine aktarılacaktır. Ayrıca ilk değer atama yöntemi ile bir dizi değişkeni tanımlıyorsak dizi boyutunu da yazmamıza gerek kalmayacaktır. Çünkü derleyici otomatik olarak biz kaç değeri içerisine yazdıysak dizi değişkenini o boyutta tanımlayacaktır. Yukarıda yazdığımızın aynısını şu şekilde de yazabiliriz.

```
int _1a_sinifi_ogrencileri[] = { 254, 951, 253, 425, 12, 84, 365, 145 };
```

Burada yazacağımız değerleri saymakla ve her yeni değer eklediğimizde bu boyutu artırmakla uğraşmadık. Artık derleyici otomatik olarak dizinin boyutunu hesaplıyor ve bizi bir yükten kurtarıyor. Eğer istersek yukarıdaki gibi bir tanımlama da yapabiliriz. Fakat böyle bir tanımlama daha kullanışlı olacaktır. Dizilere ilk değer atamayı şu şekilde özetleyebiliriz.

```
Değişken_Tipi Dizi_Adı [boyut(opsiyonel) {deger1, deger2, deger3 ...};
```

Şimdiye kadar yukarıda verdiğimiz sentaks yapısına benzer yapıları her zaman vermekteyiz. Bunu bir kağıda not alıp programlamada referans olarak kullanmanız çok önemlidir. Programlamada ciddi bir seviyeye gelene kadar sürekli bu sözdizimi referanslarına bakmanız gerekecektir. Hatta C dilinde çok ileri seviyeye gelmiş programcılar bile ellerinin altında referans kitaplarını bulundurmaktadır. En azından C dilinin tasarımcısı Dennis Ritchie'nin kitabını her C programcısı elinin altında bulundurmalıdır. Biz de ileri seviye konulara geldiğimizde bu kitaptan bahsedeceğiz. Şu an C dili hakkında yazılan bütün kitaplar o kitabı kaynak almaktadır.

Diziler üzerinde sıralama algoritmaları gibi işlemleri daha sonraya bırakalım ve çok boyutlu dizilere geçelim. Daha öncesinde dizilerin ne kadar işlevsel olduğunu anlatmıştık. Dizileri bu anlattığımızdan daha işlevsel yapmamız mümkündür. Bunun bir yolu da çok boyutlu dizilerdir. Örneğin Excel programında biz bir veri tabanı oluşturacağımız zaman iki boyutlu bir dizi kullanırız. Bu dizinin biri sütun verisini öteki ise satır verisini bulundurmaktadır. Hücre ise değeri içermektedir. Biz C dilinde de Excel tablosuna benzer bir dizi tanımlayabiliriz.

-30- İşaretçilere Giriş

C programlama hakkında şimdiye kadar basit konulardan bahsetsek de artık daha ayrıntılı konulara geçeceğimizi söyleyelim. Dikkat edin, “Zor” demiyorum ayrıntılı diyorum. C dilinin tüm özelliklerine baktığımızda zor bir tarafının olmadığını görürüz. Yalnız oldukça esnek dil olduğu için istersek çok basit programlar yazabilir istersek çok karmaşık ve zor programlar yazabiliriz. C dilinin sınırlarını zorlamak zordur fakat özelliklerini basit seviyede kullanmak kolaydır. Kısacası C dilinde ustalaşmanın zor ve zaman alıcı bir süreç olduğunu bilmeniz gerekli.

İşaretçilere Giriş

Bir de C dili derslerini yazarken bir “Merhaba Dünya!” uygulamasını yüzlerce kelimeyle anlattığım, uzattıkça uzattığımı söyleyenler oldu. Fark ederseniz ben burada tık alma niyetiyle basit bir blog yazısı yazmıyorum. Lojikprob’da tüm makaleleri aynı bir kitap yazar gibi yazmaktayım. O halde buraya yazacağıma neden kitap yazmadığımı sorabilirsiniz. Onu merak edenleri kitaplarla alakalı makalelerimi okumaya davet ediyorum. İsteseydim burada size bedavaya sunduğum makaleleri bir kitap haline getirirdim siz de 70-80 liraya alırsınız!

İşaretçiler konusuna gelirsek bunca yıldır işaretçilerin C dilinin en zor konusu olduğunu söyleyip dururlar. Aslında işaretçilerin kendisinin zor konu olduğunu bütün öğrenciler de öğretmenler de kabul etmektedir. Bunun sebebi çok yüksek seviyede çalışıp, hazır fonksiyon ve program çatıları üzerinde program yazanların alt seviye bir konu ile karşılaşınca işin temelini kavramadığı için çektikleri zorluk olmalıdır. C dili ile siz sistem programcılığı, gömülü sistemler, bilgisayar grafikleri, sürücü yazılımı gibi konularda yazılım yapıyorsunuz. Haliyle işin temelini bilmeniz ve donanımına yakın olmanız gerekli. Alt seviye programcılık ve onunla alakalı konularda donanımı bilmeden bir program yazmanız mümkün değildir.

Bana göre işaretçiler C dilinin en büyük gücünden birisidir. Bu sayede yapabileceklerinizin sınırı kalmamaktadır. Çünkü bütün bellek adreslerine

erişebildiğiniz gibi bu adresler üzerindeki değerleri de değiştirebilirsiniz. Elbette kullandığınız işletim sistemleri buna artık izin vermemekte fakat değer bir DOS işletim sistemine sahip bir bilgisayar kullansaydınız isterseniz ekran kartına, ses kartına veya bilgisayarın en kritik adreslerine doğrudan erişiminiz olurdu. Eğer yanlış bir program yazarsanız sistemi bile çökertebilirdiniz.

Günümüzde yüksek seviye programlama dilleri güvenlik endişelerinden dolayı ve çoğu zaman yüksek seviye programcıların gerek duymamasından işaretçilere sahip değildir. Artık yazılımın kütüphanelerden ileri program çatıları (framework) ile geliştirildiği bir dönemde çoğu yazılımcı işaretçileri pek kullanacak olmasa da gömülü sistemlerde C dili kullanılmaktadır ve örnek kodlarda işaretçileri bol bol görmemiz mümkündür.

İşaretçileri anlayabilmek için öncelikle bellek mimarisini ve adres kavramını bilmeniz gerekli. Sitemde elektronik ve bilgisayar bilimleri kategorisinde yazdığım belleklerle alakalı makaleleri okumanız bellek mimarisi konusunda size katkı sağlayacaktır.

İşaretçi Nedir?

İşaretçi (pointer) tam anlamıyla içerisinde bellek adresini bulunduran bir değişken demektir. İşaretçi değişkenin yanı sıra adres erişim operatörü ve değer erişim operatörü olmak üzere iki operatör de bulunmaktadır. Bildiğiniz üzere C dilinde değişkenler bir bellek adresi veya bellek bölümüne verilen adlardan oluşmaktaydı. Diziler de aslında bir bellek adresinden başlayan değişkenler sırasıydı. Normalde değişkenimizin bulunduğu adres 0xFD12 olsa da biz program yazarken buna *sayi* adını vermekteyiz. Doğrudan bellek adresleri yerine değişken tanımlayıp bunun üzerinde işlem yapmanın kolaylığı aynı internet sitelerine IP adresleri yerine site adlarından erişim sağlamak gibidir. Siz *lojikprob.com* adresine *86.125.223.61* adresini yazarak gitmek zorunda kalsaydınız ne zor olurdu değil mi? Aynı zamanda değişken tipleri ile de sınırlı olan bellek hücrelerinden bölümler gruplanır ve bu çoklu hücrelere

büyük değerler sığdırılır. Normalde bir bellek hücresi 8-bit veri bulundurabilir ve bu da 0-255 değerine denk gelir. Bu kadar küçük bir veriye hücrenin adını yazarak erişmemiz mümkündür. Örneğin 0x50 hücresine eriştiğimiz zaman değeri doğrudan okuyabiliriz.

Büyük verilerde ise çoklu hücreler kullanılmak zorunda kalındığı için erişimde bir sıkıntı yaşanacaktır. Örneğin 16 bitlik bir tam sayı değişkeni 0 ile 65535 arası bir değer alabilir. Bunun için iki adet hücreye ihtiyaç vardır. Bir hücre üst bitleri diğer hücre ise alt bitleri barındırmak zorundadır. Böyle bir sayı için örneğin üst bitte 0x50 adresine alt bitte ise 0x51 adresine erişmek zorundayız. C dili değişken tipleri ile bunu otomatik olarak ayarladığı için bunlara hiç kafa yormamıza gerek yoktur.

Assembly dilindeki gibi doğrudan bellek erişimi yerine C dilinde değişken adları ile bellekte belli hücreleri tanımlar ve onlar üzerinde işlem yaparız. Değişken tanımlarken herhangi bir adres değeri girmeyiz. Derleyici uygun adresi kendisi otomatik olarak verir. Bizim veriye erişimimiz burada sadece değişken adları ile olduğundan arada bir perde söz konusu olmaktadır. Bu da bazen kısıtlayıcı olup bizim yüksek performanslı ve esnek uygulama yapmamızın önüne geçmektedir.

Ayrıca Assembly dilinde olduğu gibi donanımın bütün adreslerine erişme imkanımız olmadığı için özel fonksiyon yazmaçları gibi donanımı kontrol ettiğimiz birimlerin adreslerine işaretçiler olmadan erişemeyiz. Örneğin 0x14 adresindeki bir veriyi değiştirmek istediğimiz zaman yine işaretçilerden faydalanmaktayız.

İşaretçiler bellek adreslerine erişmemizi sağladığı gibi bellek adreslerindeki değerlere de erişmemizi sağlamaktadır. Bu sayede Assembly diline ihtiyaç kalmadan donanım tamamen bizim kontrolümüz altında olabilmektedir.

İşaretçiler olmasaydı C dili yüksek seviye bir dil olarak yüksek seviye dilin getirdiği dezavantajlardan birine sahip olurdu. İşaretçiler olmasaydı C dili günümüzde belki

ölü diller arasında yer alırdı. Çünkü C dilinin kendine özel konumunun en önemli etkenlerinden biri işaretçilerdir. Bir sonraki başlıkta işaretçileri uygulamalı olarak sizlere anlatacağız.

-31- İşaretçiler

Önceki bölümde işaretçilerin bellek adreslerine erişmemizi ve bunların içindeki değerleri değiştirebilmemizi sağlayan bir özellik olduğundan bahsetmiştik. Yalnız işaretçiler uygulamada bu anlattığımız kadar basit bir iş için kullanılmamaktadır. Fonksiyonlarda referans ile aktarma yapabilir, fonksiyonlar arası fonksiyonları aktarabilir ve dinamik veri yapıları oluşturabiliriz. Bu oluşturduğumuz veri yapıları önceden anlattığımız değişkenler, diziler veya sonra göreceğimiz yapılar gibi sabit değildir. Dinamik veri yapıları program işletilirken büyüyüp küçülebilir. Ayrıca veri yapılarından bahsetmişken veri yapılarının bilgisayar bilimleri arasında önemli ve ayrı bir yeri olduğunu söyleyelim. Bağlı liste, sıra, yığın ve ağaç gibi veri yapıları sadece C dilinde değil bütün programlama dillerinde karşımıza çıkacak genel konulardandır.

İşaretçi Değişkeni Tanımlama

İşaretçileri kullanabilmek için öncelikler içerisinde “Adres verisi” barındıran bir işaretçi değişkeni tanımlamamız gereklidir. Bu işaretçi değişkeni normal değişkenlere benzeyip kendine ait bir adresi olsa da diğer değişkenler gibi bir sayı verisi barındırmamaktadır. Barındırabildiği tek değer bir adres verisi olmaktadır. Bir işaretçi değişkeninin adres barındırabilmesi için yine bir değişkene ait adresi barındırması gereklidir. Yani bir tarafta işaretçi değişkeni diğer tarafta ise işaretçi değişkeninin “işaret ettiği” gerçek değişken olmalıdır. İşaretçi değişkeni adından da anlaşılacağı gibi kendisine ulaşıldığında bizi işaret ettiği değişkene götürmelidir.

Aşağıdaki örnek söz diziminde bir işaretçi tanımlanmıştır. İşaretçiler de değişkenlerde olduğu gibi kullanmamız için öncelikle tanımlanmak zorundadır.

```
int *isaretci_degiskeni;
```

Burada “*” işareti tanımlama sırasında bu değişkenin işaretçi değişkeni olduğunu söylemektedir. Dikkat etmeniz gereken en önemli nokta “*” işaretinin program akışında farklı bir anlama geldiği, tanımlama sırasında ise farklı bir anlama geldiğidir. Program akışında bu işaret işaretçinin işaret ettiği değişkendeki değere erişmemizi sağlasa da tanımlama sırasında değişkenin işaretçi olduğunu derleyiciye söylemektedir. *int* isim* veya *int *isim* şeklinde tanımlamalar geçerli olsa da siz *int *isim* şeklinde yıldızın isme yanaşık olduğu şekli kullanmalısınız. Burada *int* tipinde bir tam sayı değişkenin adresini içinde barındıracak bir işaretçi tanımlanmaktadır. Aynı değişken tipleri olduğu gibi işaretçi tipleri de vardır ve aynı tipteki işaretçi aynı tipteki değişkeni işaret edebilir. Siz *int* tipindeki bir işaretçi ile *float* tipinde bir değişkeni ilişkilendiremezsiniz.

İşaretçiye Değer Atama

İşaretçiler işaret ettikleri bir değişkenin adres verisini bulundurmadan bir işe yaramazlar. Siz de bir işaretçi tanımladığınız zaman öncelikle bu işaretçiye bir değer atamanız gerekecektir. İşaretçiye atadığınız değer işaret edilecek değerın adresi olmalıdır. Başta söylediğimiz gibi işaretçiler adresten başka bir değer alamaz. Biz bir değişkenin adını yazdığımızda o değişkenin değerini yazmış gibi oluruz. Yani o değişkeni doğrudan referans etmiş oluruz. İşaretçiye bir değer atamamız gerektiğinde o değişkenin adresini atamamız gerektiğinden *addressof* (&) operatörünü kullanırız. Bu operatörü *scanf()* fonksiyonundan bilmeniz gerekir. *scanf()* fonksiyonunda da bu operatör ile değişkenlerin adresleri fonksiyona aktarılmaktadır. Böylelikle *scanf()* fonksiyonu doğrudan değişkenlere erişebilmiş olur ve değişkenler kopyalanarak boş yere hafıza işgal edilmez.

```
int degiskenimiz = 5;
```

```
int *degisken_isaretcisi;  
degisken_isaretcisi = &degiskenimiz;
```

Görüldüğü gibi öncelikle normal bir değişken tanımladıktan sonra bir de işaretçi tanımlıyoruz. Orada tanımladığımız işaretçi boş halde. Biz bunu önceki tanımladığımız değişkeni işaret etmek için kullanacağız. Bunun için önceki tanımladığımız *degiskenimiz* adındaki değişkenin adresine ihtiyacımız var. Bu adres değerini “&” operatörü ile alıyoruz ve *degisken_isaretcisi* adındaki işaretçi değişkenine aktarıyoruz. Bundan böyle *degisken_isaretcisi* işaretçi değişkeninin içerisi boş olmayacak ve *degiskenimiz* değişkeninin adresini içerecektir. Biz bu işaretçiye eriştiğimiz zaman bizi *degiskenimiz* değişkeninin değerine götürecektir. Eğer boş bir işaretçi tanımlarsak muhakkak 0 veya *NULL* değerini atamamız gereklidir. Programın ilerleyen kısımlarında farklı değerleri atayabilsek de ilk tanımlamada boş bırakmamak gereklidir.

Bunu böyle neden tanımladığımızı sorabilirsiniz. Normalde *degiskenimiz* diye tanımladığımız tam sayı değişkeninin değerine erişebiliyor ve üzerinde tam manasıyla bir işlem yapabiliyorduk. Burada bir işaretçi *degiskenimiz* adlı değişkenin adresini aldı ve istediğimiz vakit bu işaretçi vasıtasıyla de *degiskenimiz* değişkenine erişip değerini okuyacak veya değiştirebileceğiz. Bunu normalde değişkeni yazarak da yapabilirsek de C dilinde “kapsam” adı verilen bir değer erişim kısıtlamasının olduğunu unutmayın. Bir değişken global olmadığı müddetçe ancak ilgili kapsamda erişilebilir. Bu bir fonksiyon bloku veya döngü bloku olabilir. Üst kapsamdaki değişken alt kapsamlar tarafından erişilebilse de alt kapsamdaki değişken üst kapsamlar tarafından veya aynı statüde farklı kapsamlar tarafından erişilemez. Kapsamların { ve } işaretleri arasındaki bölge olduğunu biliyoruz. Bu değişkenin adresini bilmekle farklı kapsamlar tarafından da erişme imkanına sahip oluruz. Çünkü doğrudan adrese ulaşıyoruz!. Eğer değişkenimiz *auto* yani geçici değişken ise kapsam

dışında yok edildiğini unutmayın fakat *static* tipinde bir değişken bu şekilde erişilebilir. Ama işin aslına bakarsanız işaretçilerin değişkenlerle de pek fazla işi yoktur.

İşaretçinin İşaret Ettiği Değişkenin Değerine Erişme (Indirection)

Dolaylı erişim adını verebileceğimiz *indirection* kavramı programlama dillerinden öte mikroişlemci mimarisinin ana yapılarından biridir. Mikroişlemci komutlarına baktığımızda dolaylı erişim ve dolaylı atlama gibi komutların olduğunu görürüz. Program akışı bizim yazdığımız bir sabit adres değerine göre değil bir yazmaçta bulunan değişken adres değerine göre yönlendirilebilir. Bu dolaylı erişim her seviyede programa büyük bir esneklik katmaktadır. Yukarıda bir değişken tanımladık, sonrasında ise bir işaretçi değişkeni tanımladık, en sonunda & operatörü ile değişkenin adresini işaretçi değişkenine yükledik. Artık son yapacağımız iş ise değişkene “işaretçi üzerinden” erişebilmektir. Bunun için “*” operatörünü kullanırız fakat yukarıda söylediğimiz gibi bu operatör ile işaretçi tanımlama operatörü birbirinden farklıdır. Burada bu operatöre dolaylı erişim operatörü (*Indirection Operator*) adı verilmektedir. Şimdi *degiskenimiz* değişkeninin değerine erişmek için bir kod yazalım.

```
printf("%i", *degisken_isaretcisi);
```

Burada eğer doğrudan *degisken_isaretcisi* işaretçi değişkenini yazdırmak isteseydik bize adres değerini verecekti. Elbette bunu *printf()* fonksiyonu ile *%i* formatında yazdırmamız mümkün olmadığından işaretçi formatında yani *%p* formatında yazdırabilecektik. Ama burada işaretçi değişkeni işaret ettiği değişkenin değerini yazdırmakta. Bizim normal değişkenimiz ise *int* tipinde olduğu için format olarak tam sayı formatını seçtik. Ekran görüntüsü “5”

değerini gösterecektir. “*” operatörü burada adres değerinde bulunan değeri getirme görevini yapmaktadır.

İşaretçi Operatörleri Özeti

Bu işaretçi operatörlerinin ne işe yaradığını tam olarak kavramadıkça işaretçileri anlamamız ve kodları incelemeniz beklenemez. Matematik operatörlerini (+, – gibi) kolayca anlayabilsek de bunlar yeni bir konuya ait ve alakasız gibi görünen operatörler olduğu için kafamızı karıştıracaktır. Özellikle değişken tanımlama ve değer erişme sırasında kullandığımız “*” işaretinin kullanılması kafanızı karıştıracaktır. Tanımlama sırasında bu işaret operatör olarak değil belirtme işareti olarak kullanılmaktadır. Yani bu tanımlanan değişkenin işaretçi olduğunu söylemektedir. Ötekinde ise operatör olarak değer erişim operatörü olarak görev yapmaktadır. *pointer_int*, *pointer_float* demek yerine *int**, *float** demişler. Bunun işaretçi operatörleri ile ilgisi yoktur.

İşaretçi operatörleri ise program akışında yer alan & ve * operatörleridir. & operatörü adres operatörü olup bir değer adresini geri döndürmektedir. Normal bir şekilde bir değere eriştiğimiz zaman bunun değerini alırız. İşaretçiye normal bir şekilde eriştiğimiz zaman ise adres değeri alırız. Yani & operatörü işaretçi olmayanlar ile beraber kullanılmaktadır. İşaretçiler zaten normal olarak adres değerini vermektedir. “*” operatörü ise dolaylı erişim operatörüdür. Bildiğiniz üzere işaretçi değişkenleri normal erişim sırasında adres değerini bize vermekte. O halde işaret ettikleri adresin değerini elde etmemiz için özel bir operatör kullanmamız gerekli. Bunu & operatörünün tersi olarak da düşünebilirsiniz. “*” operatörü ile işaretçilerin adres değerinde yer alan veri değerini elde ederiz. Bunu da normal bir değişken ile kullanmamızın bir anlamı olmayacaktır çünkü normal değişken bize adres değil değer verisini vermektedir.

Şimdi bütün bu anlattıklarımızın örnek programda nasıl çalıştığını görelim.

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

int main(int argc, char *argv[]) {
    int degisken = 5;
    int* isaretci;
    isaretci = &degisken;

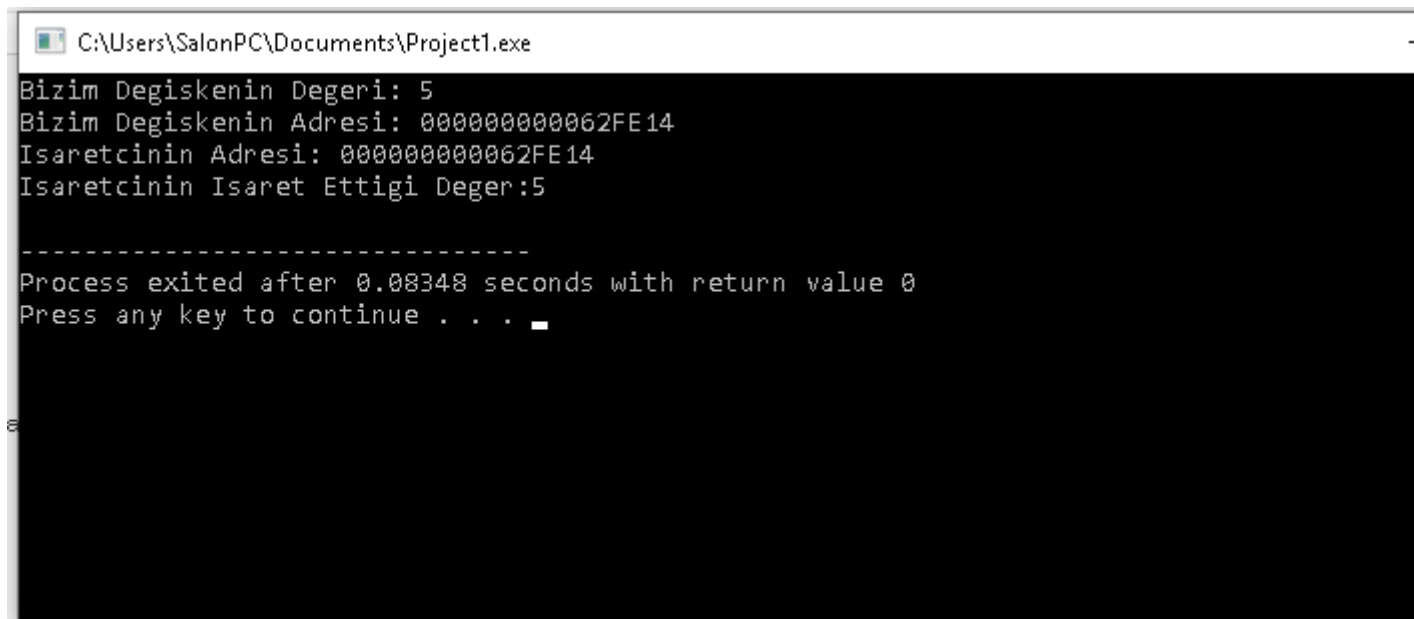
    printf("Bizim Degiskenin Degeri: %i \n", degisken);
    printf("Bizim Degiskenin Adresi: %p \n", &degisken); //
Burada işaretçi yok sadece adres var.
    printf("Isaretcinin Degeri: %p \n", isaretci);
    printf("Isaretcinin Isaret Ettigi Deger:%i \n",
*isaretci);

    return 0;
}
```

Burada yukarıda anlattıklarımızın uygulamasını görebilirsiniz. Sizin de bir bakışta programı anlayacağınızı umuyorum. Öncelikle yukarıdaki kod parçasına benzer bir şekilde bir değişken ve bir işaretçi değişkeni tanımladık. Sonrasında ise tanımladığımız değişken ile işaretçi değişkenini ilişkilendirdik. Sonrasında ise bütün bu bahsettiğimiz verileri *printf()* fonksiyonu ile uygun formatlarda yazdırdık. Tam sayı için *%i* ve işaretçi (adres) tipi için *%p* format belirtecini kullandığınızı biliyorsunuz. Burada öncelikle bizim bildiğimiz üzere değişkenin değerini yazdırdık ve sonrasında *°isken* ile adres operatörünün döndürdüğü adres değerini yazdırdık. Sonrasında ise *isaretci*

değişkenini doğrudan yazdık çünkü biz o işaretçi değişkenini yazdığımız zaman içindeki adres değeri oraya gidecek. O halde format olarak %p yazmamız gerekti. Sonrasında ise işaretçinin adresinde bulunan veriyi çıkarmak için * operatörünü *isaretcı* işaretçi değişkeniyle kullandık ve bu veriyi %i formatında yazdırdık. Çünkü **isaretcı* kısmı işletildiği zaman ortada 5 sayısından başka bir şey kalmayacaktır.

Şimdi programın çıktısına bakalım ve nasıl olduğunu görelim.



```
C:\Users\SalonPC\Documents\Project1.exe
Bizim Degiskenin Degeri: 5
Bizim Degiskenin Adresi: 000000000062FE14
Isaretcinin Adresi: 000000000062FE14
Isaretcinin Isaret Ettigi Deger:5

-----
Process exited after 0.08348 seconds with return value 0
Press any key to continue . . .
```

Düzeltilme: Yukarıda işaretçinin adresi diye belirttiğim kısım “İşaretçinin içinde bulundurduğu adres verisi” demektir. İşaretçi değişkeninin kendi adresi aynı değişkenlerin adresi gibidir ve belleğin başka bir kısmında yer almaktadır.

Burada gördüğünüz üzere değişkenin değeri ile işaretçinin işaret ettiği değer birbirinin aynı olmakta. Değişkenin adresi ile de işaretçinin adresinin birbirinin aynı olduğunu görüyoruz. Bu aynı kimlik bilgilerinden T.C kimlik numarasını bulmak ve T.C kimlik numarasından ise kimlik bilgilerine ulaşmak gibi bir durum. Biz burada int tipinde sıradan bir değeri kullandığımız için çok da bir şey beklemememiz lazım. Çünkü işaretçilerin asıl kullanım alanları dizilerdir.

Diziler bildiğiniz üzere belli bir sayıda değişkenin birbiri ardınca dizilmesinden oluşur. Yani dizi elemanları `sayi[0]`, `sayi[1]`, `sayi[2]` gibi sıralanırken bellekte `0x50`, `0x51`, `0x52` diye sıralanır. Bu durumda işaretçi önceki ve sonraki bellek adreslerinde dizinin elemanlarının olduğunu bilecektir. Tek bir bellek adresiyle yapabileceklerimiz çok fazla değildir.

Burada adresin oldukça uzun bir değer olduğunu görmekteyiz. 64-bitlik sistemde adres değerleri de bu kadar uzun olmaktadır. Eğer siz gömülü sistemler üzerinde çalışacaksanız bu adres değerlerinin oldukça kısaldığını görebilirsiniz. Görüldüğü gibi işaretçilerin temel yapısı oldukça basittir. Bunu zorlaştıran bunların nerelerde ve ne için kullanıldığıdır.

-32- Çift İşaretçiler

Önceki başlıkta işaretçileri en basit yoldan anlatmaya çalıştık. Bütün işaretçi eğitimlerinde işaretçiler kutu şeması ile anlatılmaya çalışılır. Biz de bu yöntemi faydalı gördüğümüz için kendimiz böyle bir şema hazırladık. Bu şemalarla

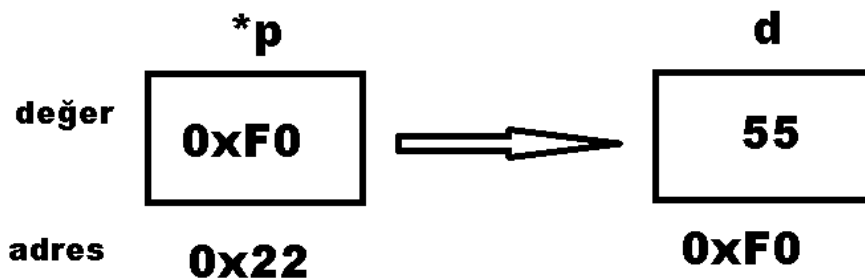
öncelikle basit bir işaretçiyi sonrasında ise işaretçiyi gösteren işaretçiyi anlatacağız.

Basit Bir İşaretçi

Bizim bir tam sayı değişkenimiz olsun ve tekrardan onu bir işaretçi vasıtasıyla işaretleyelim. Bunun için şöyle bir kod yazacağız.

```
int d = 55;  
int *p;  
p = &d;
```

Öncelikle d tamsayı değişkenini tanımlayıp buna 55 değerini aktarıyoruz. Sonrasında p adında bir işaretçi değişkeni tanımlayıp d değişkeninin adresini buna aktarıyoruz. Kodlar işletildikten sonra hafızadaki görünümü şu şekilde olacaktır.



Burada d değişkeninin değeri kutu içerisinde gösterilmiştir. Kutuyu bir hafıza hücresi olarak düşünebilirsiniz. Her kutunun değeri olduğu gibi adres değeri de vardır. Bu değer ile adres değeri birbiriyle alakasızdır. Kısacası değer kutunun içerisinde yer alır ve adres çağırılarak veri yolundan okunur. İşaretçi değişkeni ise diğer değişkenlerden farklı olarak içinde adres değerini bulundurabilir. Bu adres değeri de bir değişkenin, fonksiyonun veya dizinin

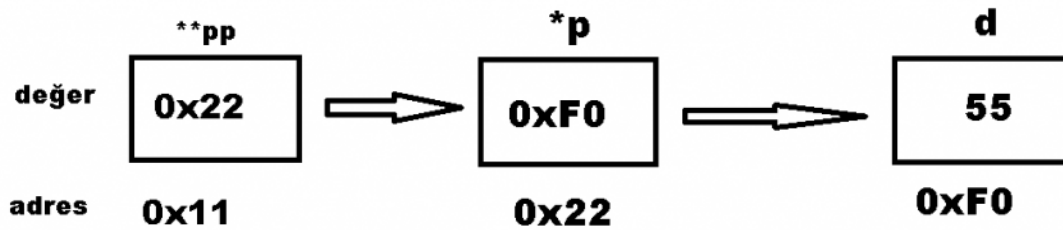
adres değeri olabilir. Bu durumda &d dediğimiz zaman 0xF0 değeri geri dönmektedir ve bu da p işaretçisine aktarılmaktadır. Artık p işaretçisinin içine baktığımızda bize d değişkeninin adresini “göstermektedir.” Böylelikle bizi d değişkenine yönlendirecek bir kılavuzumuz olmuş olur.

Çift İşaretçiler

Şimdi işi biraz daha karmaşık hale getirelim ve işaretçinin işaretçisi olur mu buna bakalım. Yukarıdaki gibi bir kod yazacağız fakat bir değişkenin ve işaretçinin yanı sıra işaretçiyi gösteren çift işaretçi yer alacak. Bu çift işaretçi meselesi uygulamada kafanızı karıştırabilse de bunu en basit şekliyle anlatmaya çalışacağız. Şimdi kodu inceleyelim.

```
int d = 55;
int *p;
p = &d;
int **pp; // çift işaretçi
pp = &p; // işaretçinin adresi
```

Burada **pp diyerek işaretçiyi gösteren bir işaretçi tanımladık. İşaretçiyi gösteren işaretçiler diğer işaretçilerden farklı olarak “”** belirtecini alırlar. Sonrasında ise işaretçinin adresini yine addressof (&) operatörü ile çift işaretçiye aktardık. Dikkat edin, işaretçinin bulundurduğu adres değerini değil işaretçinin kendi adresini aktardık. Çünkü işaretçiler adres verisi depolayan birer değişkendi. Bu değişkenler de hafızada bir yer işgal edecek ve bu yerin de muhakkak bir adresi olacaktır.



Burada ****pp** diye tanımladığımız **pp** çift işaretçinin de kendi adresi bulunmaktadır. Ama değer olarak normal bir değişkenin değil bir işaretçi değişkeninin adresini bulundurmaktadır. İşaretçi değişkeni ise bizi bir normal değişkene götürmektedir. Burada bütün işaretçilerin ve değişkenlerin birer değeri olduğu gibi adres değerlerinin olduğunu da unutmayalım. Değer yani veri programlamada bizi ilgilendiren kısım olduğu için sürekli içli dışlı oluruz. Adres ise donanım ve makine dili ile alakalı kısım olduğu için gözümüzün önüne gelmez. Aslında bizden saklanmaktadır. Bunu daha öncesinde size söylemiştik. Bizden saklanan bu adresi ise bazen bizim bilmemiz gerekebilir. İşte burada işaretçileri kullanmaktayız.

Şimdi bu çift işaretçinin nasıl çalıştığını görmek için bir program yazalım ve deneyelim.

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

int main(int argc, char *argv[]) {
    int d = 55;
    int *p;
    p = &d;
    int **pp; // çift işaretçi
    pp = &p; // işaretçinin adresi
```

```

    printf("Bizim Degiskenin Degeri: %i \n", d);
    printf("Bizim Degiskenin Adresi: %p \n", &d); // Burada
işaretçi yok sadece adres var.
    printf("Isaretcinin Adresi: %p \n", p);
    printf("Isaretcinin Isaret Ettigi Deger:%i \n", *p);
    printf("Cift Isaretcinin Isaret Ettigi Deger:%i\n",
**pp);
    printf("Cift Isaretcinin Kendi Degeri: %p \n", pp);
    printf("Cift Isaretcinin Adresi %p", &pp);
    return 0;
}

```

Burada önceki kısmı daha önce anlattığımız için sadece çift işaretçi ile alakalı olan kısımları size açıklayacağım. *pp diyerek çift işaretçinin işaret ettiği değeri gösteriyoruz. Bu işaret ettiği değer işaret ettiği işaretçinin işaret ettiği değer olmaktadır. Yani bizi zincirin en sonundaki işaret edilen tam sayı değişkeninin değerine götürmektedir. Bu değer ise 55 sayıdır. Sonrasında ise çift işaretçinin kendi değerini %p formatı ile doğrudan pp yazarak görmektedir. Bu da p işaretçisinin adresinden başka bir şey değildir. Sonrasında ise pp işaretçisinin kendi adresini ekrana yazdırdık. Bu değer pp işaretçisinin kendi adres değeri olup geri kalan değişkenler ve işaretçilerle alakalı değildir. Bilgi vermek için size gösterdik.

C dilinde &* ve *& operatörlerini beraber yazmanız aynı çarpma ve bölme işlemlerinin birbirini götürmesi gibi birbirini sıfırlayacaktır.

Şimdi programın çıktısına göz atalım ve 64 bitlik bir sistemde nasıl görüldüğünü görelim.

```
C:\Users\SalonPC\Documents\Project1.exe
Bizim Degiskenin Degeri: 55
Bizim Degiskenin Adresi: 000000000062FE1C
Isaretcinin Adresi: 000000000062FE1C
Isaretcinin Isaret Ettiği Deger:55
Cift Isaretcinin Isaret Ettiği Deger:55
Cift Isaretcinin Kendi Degeri: 000000000062FE10
Cift Isaretcinin Adresi 000000000062FE08
-----
Process exited after 0.02611 seconds with return value 0
Press any key to continue . . .
```

Unutmayın çift işaretçiye tek “*” operatörü koyduğunuz zaman sizi işaret ettiği işaretçiye götürür. Eğer çift yıldız operatörü “**” koyarsanız işaret ettiği işaretçinin işaret ettiği değere götürür. Burada çift işaretçinin gösterdiği işaretçinin gösterdiği değere götürmüştür.

İşaretçiler daha önce bahsettiğimiz gibi bellek adreslerinin tamamına erişmemizi sağlayan bir özelliktir. Fakat günümüzde kullandığımız Windows gibi işletim sistemlerinde her program (görev) için belli bir bellek bölümlendirmesi bulunmaktadır. Bir program kendine ayrılmış bellek bölümünde çalışabilir fakat belleğin tamamında çalışamaz. Bu yüzden biz belleğin önemli noktalarına veya donanım ile alakalı özelliklere erişememekteyiz. Eğer ben bunu MS-DOS gibi bir işletim sisteminde yapsaydım işaretçilerle çok ilginç gösteriler sunabilirdim. Fakat bir programcı olarak bilmeniz gerekir ki artık işletim sistemleri donanım ile aranızda büyük bir bariyer olmaktadır. Siz asla doğrudan ses kartına veya ekran kartına erişemezsiniz. İşletim sistemi bile donanıma erişmek için sürücülerini kullanır. Siz ise işletim sisteminin sağlamış olduğu kütüphaneleri kullanırsınız. Örneğin

grafik için DirectX veya OpenGL ya da Windows elementleri ve özellikleri için Windows API bunların başlıcalarıdır.

-33- İşaretçileri Referans Olarak Aktarmak

Buraya kadar işaretçileri sadece değişkenlerle kullandık ve aslında çok da bir şey yapmadık. Sadece size işaretçi mantığını ve işaretçinin ne olduğunu size göstermeye çalıştık. Şimdi ise işaretçilerin asıl kullanım alanlarına geçeceğiz. Basit bir konu olduğunu düşündüğüm için ilk olarak işaretçileri referans olarak aktarmaktan bahsedeceğim. Bu işaretçilerin fonksiyonlar ile kullanımına bir örnek olacaktır. C++ dilinde referans argüman adını verdiğimiz değişkenlerin adreslerinin aktarılması ve kapsam dışındaki değişkenler üzerinde işlem yapılması C dilinde yoktur. Fakat bu bizim C dilinde işaretçileri fonksiyonlarla beraber kullanamayacağımız anlamına gelmez. C++ dilinde fonksiyonlar daha özelliklidir ve işlevseldir. Çok daha sonra C++ dilini anlattığımız zamanlar bunları size açıklayacağız ve karşılaştırmasını yapacağız.

Şimdi bir program yazalım ve bu programın fonksiyonu argüman olarak işaretçi alsın. Biz ana programda bir değişkeni aktardığımız zaman fonksiyona değişkenin kopyası yerine adresi gitsin ve fonksiyon “doğrudan” değişken üzerinde değişiklik yapsın. Bunun için şöyle bir program yazalım.

```
#include <stdio.h>
#include <stdlib.h>
void kare_al( int *kareptr);
int main(int argc, char *argv[]) {
    int sayi;
    printf("Karesini Almak Istediginiz Sayi:");
    scanf("%i",&sayi);
    kare_al(&sayi);
    printf("\nSayinin Karesi: %i", sayi);

    return 0;
}
```

```
void kare_al( int *kareptr)
{
    *kareptr = *kareptr * *kareptr;
}
```

Bu program önceki anlattığımız programlar kadar basit değil. Çünkü işaretçiler aslında basit olsa da oldukça esnek bir kullanıma sahip olduklarından bazı yerlerde oldukça zorlayıcı olabilir. Yine de bu programı anladığınız zaman oldukça basit olacağını göreceksiniz. İyi anlamanız için bunu satır satır açıklayalım.

void kare_al(int *kareptr); Burada kare_al adında bir fonksiyon oluşturuyoruz. Bu fonksiyonun void değerini döndürdüğünü unutmayın. Yani fonksiyon değer döndürmüyor. Değer döndürmediği halde yaptığı işlemi nasıl öğreneceğiz dersiniz bu fonksiyon doğrudan main kapsamı içindeki değer üzerinde işlem yapıp o değeri değiştirecek. Yani fonksiyona ne değer gidecek ne de ondan değer gelecek. Aradaki yavaşlatıcı transfer işlemlerinden kurtulmuş olacağız ve bu da programımızın performansını artıracak. Sonrasında fonksiyonun aldığı değere baktığımızda int *kareptr olarak bir işaretçi değerini aldığını görmekteyiz. Yani bu fonksiyon değer olacak değişkenin kopyasını değil bir adres değerini almakta. Biz de değişkenin adresini yolladığımızda değişkenin adres değeri fonksiyona gidecek.

kare_al(&sayi); Burada addressof (&) operatörü ile sıradan bir değişkenin adresi fonksiyona argüman olarak aktarılmıştır. Normalde fonksiyonlara değişkenlerin kendisi aktarılrsa da aslında fonksiyona giden değer değişkenin değerinin kopyası olmaktadır. Yani program akışı fonksiyona gittiği zaman fonksiyon parantezlerine yazdığımız değişkenlerle hiçbir bağımız olmamaktadır. Sadece o değerlerin kopyaları elimizde olmakta ve onlar

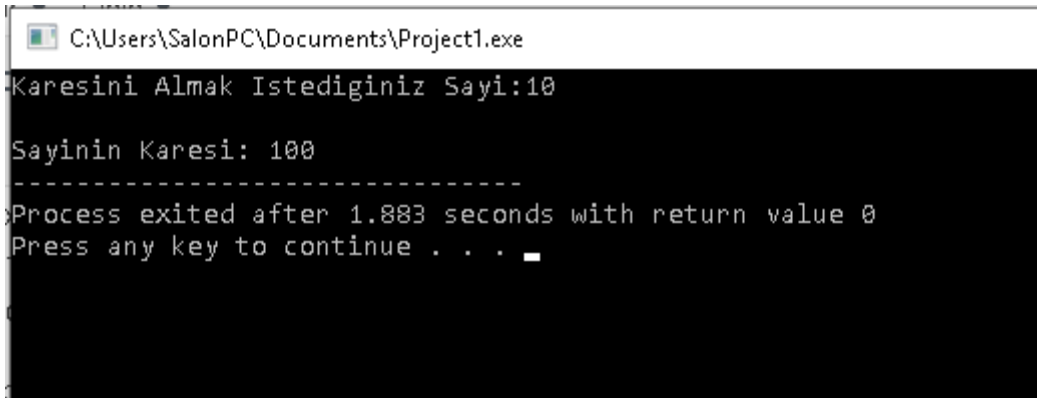
üzerinde işlem yapmaktayız. Fonksiyonların sadece bir değer döndürebildiğini unutmayın. Fonksiyona giden değerler değişkenlerin kopyası ve fonksiyondan çıkan bir değer de bir değişkene aktarılabilirdiği için fonksiyonların esnekliği kısıtlanmış olmaktadır. Burada fonksiyonlara değişkenlerin kendisini aktarabildiğimiz gibi fonksiyonlar birden fazla değer üzerinde işlem yapıp kayıt etme imkanına sahiptir. Bu satırda sadece &sayi diye sayi değişkeninin adresini yollamaktayız.

***kareptr = *kareptr * *kareptr;** Burada dikkat ederseniz fonksiyon bloku içerisinde fonksiyon aldığı değer üzerinde işlem yapıyor ve return komutu ile herhangi bir değer geri döndürmüyor. Buna rağmen fonksiyon doğrudan sayi değişkeni üzerinde kalıcı olarak değişiklik yapmakta. Dikkat etmeni gereken bir nokta ise “*” operatörü ile işaretçinin işaret ettiği değişkenin değerine erişmektir. Eğer böyle olmasaydı ve kareptr işaretçisini düz halde yazsaydık adresleri birbirine çarpmanın hiçbir anlamı olmayacaktı! Daha sonra dizilerle beraber kullandığımız işaretçilerde aritmetik işlemlerin bir anlamı olsa da burada herhangi bir anlamı yoktur. O yüzden biz sayi değerine erişip sonra tekrar sayi değerine eriştikten sonra bu ikisini yine (*) operatörü ile birbiriyle çarpılmaktayız. Burada işaretçi erişim operatörü ile çarpma işleminin operatörünün birbirinin aynısı olması biraz kafa karıştırıcıdır. Sonrasında ise atama operatörü ile *kareptr yani sayi değişkeninin değerine değerimizi atıyoruz.

Tekrar söyleyelim. Bizim burada sayi değişkenine kare_al fonksiyonu içerisinden erişme imkanımız yoktur. C dilinde kapsamlar veri erişiminde sınırlayıcı özelliğe sahip olurlar. Burada ya global değişken tanımlayacağız bütün fonksiyonlar tarafından erişilebilecek ya da bu şekilde işaretçiler vasıtasıyla farklı bir kapsamdaki değişkenin kendisine ulaşacağız. Global değişkenleri kullanmanın hem RAM belleği aşırı derecede işgal edeceğini

hem de güvenlik sorunlarına yol açabileceğini şimdiden belirtelim. Global değişkenler eğer çok fazla olursa program yazarken de hata yapma oranımızı artıracaktır. Bu şekilde daha performanslı ve ekonomik bir yöntemi görmüş olduk.

Programımız çalıştığında şu şekilde bir ekran görüntüsü verebilir.



```
C:\Users\SalonPC\Documents\Project1.exe
Karesini Almak Istediginiz Sayi:10
Sayinin Karesi: 100
-----
Process exited after 1.883 seconds with return value 0
Press any key to continue . . . _
```

Gömülü sistemlerde çalışacaksanız kütüphanelerdeki fonksiyonların (Mesela STM32 HAL) değişkenlerin kendisini değil de adresini aldığını göreceksiniz. Bunun neden böyle olduğunu bu dersimizde anlayabilirsiniz. Performans açısından değişkenlerin kopyalarını aktarmak yerine adreslerini aktarmak her zaman daha iyidir. Eğer bir de yapı değişkenleri üzerinde çalışıyorsanız belki onlarca farklı değişkeni aktarmanız gerekebilir. Elinizde kısıtlı bir donanım varsa gereksiz yere kopyasının RAM belleği işgal etmesini istemeyebilirsiniz.

-34- İşaretçi Aritmetiği

Buraya kadar işaretçi değişkenlerine sadece bir adres değeri atadık ve o değer öylece kaldı. Fakat bunlar bir değişken olduğuna göre içindeki değerler ile aritmetik işlemler de yapılabilirmeli değil mi? Bu konuda işaretçiler karşımıza değişkenler kadar esnek halde çıkmamaktadır. Bunun sebebi ise oldukça

basittir. Bir adres verisini çarpmak, bölmek veya mod almanın ne gibi bir anlamı olabilir? Değişken aritmetiği zaten diziler üzerinde çalışırken işimize yaramaktadır. Eğer bir dizi üzerinde çalışıyorsak dizinin ilk elemanını işaret eden işaretçinin değerini bir artırırsak artık o dizinin ikinci elemanını işaret edecektir. Ama sıradan değişkenlerde bu bizi rastgele bir değere götürecektir. O yüzden işaretçi aritmetiğini kullanacağımız yerler sınırlı olduğu gibi sınırlı yerlerde de dikkatli kullanmamız gerekecektir.

Özetlemek gerekirse biz işaretçilerle şu işlemleri yapabiliriz:

- **Artırma veya Azaltma**
- **Toplama veya Çıkarma**
- **Karşılaştırma**
- **Atama**

Dizi ve Fonksiyonlar

Örnekleri vermeden önce size diziler ve fonksiyonlara ait bir özelliği hatırlatmak istiyorum. Bildiğiniz üzere dizi ve fonksiyonları tanımlarken aynı bir değişken tanımlar gibi bir ad veriyor ve dizilere elemanları ve fonksiyonlara da kodları yerleştiriyorduk. Bu verdiğimiz ad bize ne değerini geri döndürebilir? Bu verdiğimiz adlar aslında birer işaretçi olup dizide dizinin ilk elemanının adresini fonksiyonda ise fonksiyon blokunun başladığı hafıza adresini bize verecektir. Yani fonksiyon çağırıldığında mikroişlemciye hafızada yer alan şu adrese git diye bir komut verilmektedir. Dizide ise dizinin ilk elemanının adresi verilmekte ve hafızada sırayla okunmaktadır.

Şimdi bir fonksiyonun adresini ekrana nasıl yazdırdığımızı görelim.

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

```

void kare_al( int *kareptr);
int main(int argc, char *argv[]) {
    printf("kare_al fonksiyonunun adresi: %p",kare_al);

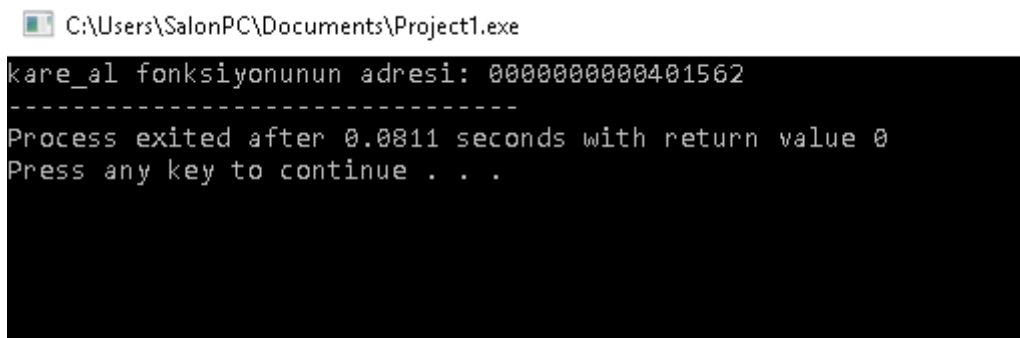
    return 0;
}

void kare_al( int *kareptr)
{

}

```

Gördüğünüz gibi yine printf ile ve %p formatında fonksiyonun adresini ekrana yazdırıyoruz. Burada kare_al diyerek kare_al fonksiyonunun adresini yazdırmış olduk. Bu biraz tuhaf gibi görünse de C dili buna imkan vermekte fonksiyonların adresini ekrana yazdırabilmektedir. Ekran görüntüsü şu şekilde olabilir.



```

C:\Users\SalonPC\Documents\Project1.exe
kare_al fonksiyonunun adresi: 0000000000401562
-----
Process exited after 0.0811 seconds with return value 0
Press any key to continue . . .

```

Fonksiyonun adresini öğrenmenin bize ne katacağı fonksiyon işaretçisinin konusu olduğundan ben asıl anlatmak istediğim konu olan dizilere geçmek istiyorum. Çünkü bundan sonra yapacağımız işlemlerde diziler bol bol yer almak zorundadır. Diziler daha önce bahsettiğim gibi aslında birer işaretçidir ve adları bize adres değerini geri döndürmektedir. Dizinin ilk elemanının adres

değerini öğrendikten sonra o adres değeri üzerinde aritmetik işlem yapıp dizinin diğer elemanlarına alt seviyeden erişme imkanımız olacaktır. Bu sayede büyük boyutlu dizilerde daha hızlı çalışma imkanımız olur. Şimdi bir dizi tanımlayalım ve onun adresini öğrenelim.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    int dizi [5] = {10, 11, 12, 13, 14};
    printf("dizi dizisinin adresi: %p",dizi);
    printf("\ndizi dizisinin ilk elemani: %i", *dizi);

    return 0;
}
```

Burada dizi adında 5 elemanlık bir dizi tanımladık ve dizi elemanlarına ilk değerlerini verdik. İlk değerlerini vermemiz sonrasında üzerinde işlem yaptığımızda bize yardımcı olacak. Sonrasında yukarıda fonksiyonun adresini öğrendiğimiz gibi %p biçimlendiricisiyle dizi dizi değişkenini yazdırdık. Dikkat ediniz, burada addressof (&) operatörü kullanılmayıp aynı işaretçilerde olduğu gibi düz bir şekilde yazıldı. Normal durumda değişkenler değeri dizi ve fonksiyonlar ise adresi vermektedir. Yalnız biz *dizi şeklinde yani erişim operatörüyle kullanırsak bu sefer de ilk elemanın değerini bize verecektir. O halde biz erişim operatörüyle de bütün dizi elemanlarına erişip bunları değiştirme imkanına sahip oluruz. Programın çıktısı şu şekilde olabilir.


```
C:\Users\SalonPC\Documents\Project1.exe
dizi dizisinin adresi: 000000000062FE00
dizi dizisinin ilk elemanı: 10
-----
Process exited after 0.04307 seconds with return value 0
Press any key to continue . . .
```

İşaretçilerle Aritmetik İşlemler ve sizeof() Operatörü

Dizi ve fonksiyonların özel konumunu öğrendiğinize göre artık diziler üzerinde işaretçi aritmetiğini kullanarak nasıl işlem yapabileceğimizi göreceğiz. Burada kullanacağımız operatörler karşılaştırma operatörleri ile beraber +, -, ++, — ve = operatörleridir. Diziler üzerinde işlem yapmadan önce de sizeof() operatörünün ne işe yaradığını görmeyi istiyorum. Çünkü işaretçilerle dizi üzerinde çalışırken dizinin boyutunu öğrenmeniz gereklidir. En basit olarak bir dizi içerisindeki bütün küçük harfleri büyük harfe çeviren bir for döngüsü oluşturduğumuzu varsayalım. Bunun için döngünün kaç kere tekrarlayacağını makine ancak dizinin boyutunu bilmekle öğrenebilir. Burada sabit boyutlu bir dizi için çok zor bir durum olmasa da dinamik dizilerde zorunlu olmaktadır. Elbette daha dinamik dizileri ve hafızada yer ayırmayı anlatacak değiliz. Fakat sizeof() operatörünü kullanmaya alışmanız sizin için daha pratik olacaktır. Şimdi sizeof() operatörüyle belli başlı değişkenlerin boyutunu ve yukarıda verdiğimiz dizinin boyutunu görelim.C

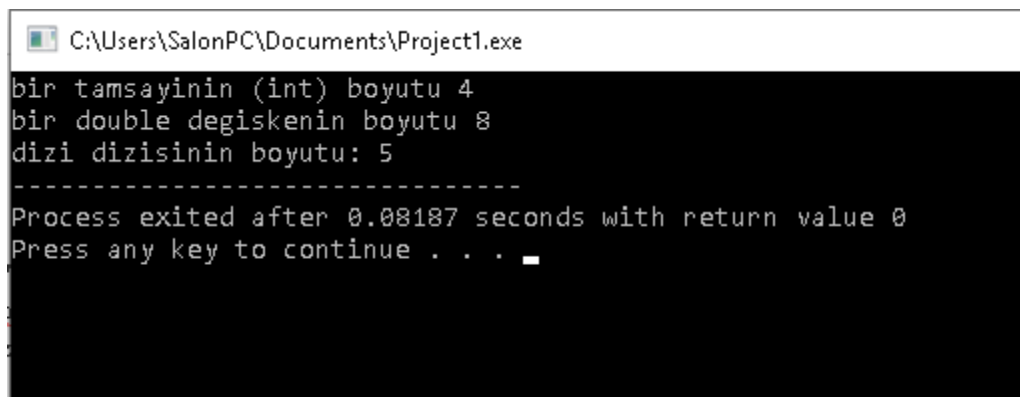
```
#include <stdio.h>
#include <stdlib.h>
void kare_al( int *kareptr);
int main(int argc, char *argv[]) {
    int dizi [5] = {10, 11, 12, 13, 14};
```

```

    printf("bir tamsayinin (int) boyutu %i \n", sizeof(int));
    printf("bir double degiskenin boyutu %i \n",
sizeof(double));
    printf("dizi dizisinin boyutu: %i", sizeof(dizi) /
sizeof(dizi[0]) );
    return 0;
}

```

Burada biz `sizeof(int);` yazdığımızda `sizeof` fonksiyonu tam sayı yani integer tipinde verinin büyüklüğünü bize geri döndürecektir. Bir integer de 32 bitlik bir derleyicide 4 baytlık bir yer kaplamaktadır. Aynı şekilde double ise 8 baytlık bir yer kaplar. Biz buraya `sizeof(int)` şeklinde yazabiliriz. Biraz kural dışı gibi görünse de aslında tamamen doğrudur. Bir dizinin boyutunu yani kaç elemandan oluştuğunu öğrenmenin yolu ise o dizinin toplam boyutunu yani toplamda kaç bayt olduğunu hesaplayıp sonrasında ise bunu bir elemana bölmektir. Bu sayede dizide kaç eleman olduğunu bulabiliriz. Aynı şekilde dizinin bir elemanının boyutunu ise `sizeof(dizi[0])` ile bulabiliriz. Çünkü dizinin ilk elemanı ile son elemanı her zaman aynı tiptir. Programın ekran çıktısı şu şekilde olacaktır.



```

C:\Users\SalonPC\Documents\Project1.exe
bir tamsayinin (int) boyutu 4
bir double degiskenin boyutu 8
dizi dizisinin boyutu: 5
-----
Process exited after 0.08187 seconds with return value 0
Press any key to continue . . . 

```

Şimdi ise işaretçilerle nasıl aritmetik işlem yaptığımıza bakalım. Biz yukarıda dizinin ilk elemanının değerine erişmiştik. Acaba sadece aritmetik işlemlerle farklı elemanlara da erişebilir miyiz bir buna bakalım.

```

#include <stdio.h>
#include <stdlib.h>
void kare_al( int *kareptr);
int main(int argc, char *argv[]) {
    int dizi [5] = {10, 11, 12, 13, 14};
    int *diziptr = dizi;
    printf("Dizinin ilk elemani: %i", *diziptr);
    printf("\nIsaretcinin adres: %p", diziptr);
    diziptr++;
    printf("\nSonraki Eleman: %i", *diziptr);
    printf("\nIsaretcinin Adresi: %p", diziptr);
    diziptr+=2;
    printf("\nİki adım sonraki eleman: %i", *diziptr);
    return 0;
}

```

Burada `int diziptr = dizi;` şeklinde bir değer atama yolunu tercih ettik.

İşaretçilere böylece ilk değeri kolayca atayabilirsiniz. Sonrasında ise her zaman olduğu gibi diziptr ile işaretçinin işaret ettiği değeri ekrana yazdırdık. Sonrasında diziptr++; işlemini yaptık. Bu işlem doğrudan diziptr işaretçisinin değerini etkilemektedir. Yani içinde bulundurduğu adres değeri 0x50 ise 0x51 olmaktadır veya işaretçinin boyutu ne kadarsa adres değeri ona göre artmaktadır. Burada 32 bitlik bir derleyicide int tipindeki bir işaretçinin boyutu 4 baytlık olduğu için her artırımda dörder dörder artmaktadır. Kısacası biz işaretçiyi bir artırdığımız zaman dizinin bir sonraki elemanını işaretler hale gelecektir. Eğer işaretçiyi iki artırırsak bu sefer de adres değeri bu durumda 8 artacak ve iki adım atlayıp o sıradaki elemanı gösterecektir. Biz hangi elemanda olduğumuzu ise programda sıfırıncı eleman ile sizeof() operatörü ile

bulduğumuz dizi boyutu sayesinde buluruz. Programın ekran çıktısı şu şekilde olacaktır.

```
C:\Users\SalonPC\Documents\Project1.exe
Dizinin ilk elemanı: 10
İşaretçinin adres: 000000000062FE00
Sonraki Eleman: 11
İşaretçinin Adresi: 000000000062FE04
İki adım sonraki eleman: 13
-----
Process exited after 0.02672 seconds with return value 0
Press any key to continue . . .
```

Şimdi bir sorun düşünelim ve buna çözüm getirmeye çalışalım. Bizim bir işaretçimiz var ve bu bir dizideki bir elemanın adres değerini barındırmakta. Fakat biz bu elemanın dizinin kaçınıcı elemanı olduğunu bilmiyoruz. Bunu öğrenmek için bir program yazalım.

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

int main(int argc, char *argv[]) {
    int dizi [5] = {10, 11, 12, 13, 14};
    int *diziptr;
    int rastgele_sayi = (rand()) % 5;
    diziptr = &dizi[rastgele_sayi];
    // diziptr artık benim de bilmediğim bir üyeyi gösteriyor!

    int sayi = (int)(diziptr - dizi);
    printf("%i", sayi);
    return 0;
}
```

Bu programda `rand()` fonksiyonu ile ürettiğimiz sayının 5'e bölümünden kalanını elde ettik. Bu bize 0 ile 4 arasında bir sayı verecek ve işaretçimize bu ürettiğimiz rastgele sayıyı atayacağız. Bu sayının rastgele olmasını programın doğruluğunu göstermek için tercih ettik. Bu bir kontrolümüz dışında değere sahip olabilen bir değişken de olabilir. Kısacası bizim diziptr işaretçimiz `dizi[0]`, `dizi[1]`, `dizi[2]`, `dizi[3]` ve `dizi[4]` elemanlarından herhangi birinin adresini alabilir.

Sonrasında ise yaptığımız elimizde bulunan dizi elemanının adresini dizinin adresinden çıkarmak oldu. Çünkü dizi elemanlarının adresi dizinin adresine eşit veya ondan fazladır. Başta söylediğimiz gibi dizinin ilk adresi dizinin 0 numaralı elemanını işaret eder. O halde kalan bize dizinin kaçınıcı elemanı olduğunu gösterecektir. Kalan sayıyı (`int`) ile integer formatına çevirip ekrana yazdırdığımızda program doğru bir şekilde bize `rastgele_sayi` ile seçtiğimiz bilinmeyen dizi konumunu bize gösterecektir. Bunu yapabilmeyi işaretçileri birbirinden çıkarabilmeye borçluyuz.

Buraya kadar anlattıklarımızın şimdilik yeterli olacağını düşünüyorum. İleride uygulamalarla beraber karşımıza işaretçi aritmetiği ve karşılaştırması çıkacaktır.

-35- İşaretçiler ve Diziler

İşaretçilerin en önemli kullanım alanlarından biri dizilerdir. Diziler derken artık sadece sayıdan ibaret olan dizileri değil bizim “yazı” olarak ifade ettiğimiz karakter dizilerini de hesaba katmamız gerekir. Karakter dizileri bilgisayar programlarında en sık kullanılan veri tiplerindendir. Daha sonra ayrıntılı olarak ele alacak olsak da karakter dizileri C dilinde String veya başka bir sınıfta ayrı bir veri tipi olarak yer almaz ve saf haliyle bizim karşımıza çıkar. Veri tipleri de hakeza öyledir siz yapı tipleri ile bunu sıfırdan tanımlarsınız. C dilinin en güzel yanlarından biri eğer harici bir kütüphane veya program çatısı kullanmıyorsanız her şeyi sıfırdan yapar olmanızdır.

Bütün bu veri işlemenin temellerine indiğimizde ise işaretçilerin ve dizilerin beraber kullanımını görürüz. Daha önceki başlıkta ifade ettiğimiz gibi işaretçiler üzerinde aritmetik işlem yaparak farklı dizi elemanlarına erişebiliyorduk. Şimdi konumuzu daha ilerleyerek devam edecek olsak da öncelikle işaretçi tiplerinin neler olabileceğini sizlere hatırlatmak istiyoruz.

İşaretçi Tipleri

Biz bir işaretçi tanımladığımızda normal değişkenden farklı olarak değişken adının başına “*” işareti getirilmektedir. Bu sayede `int *p;` yazdığımız zaman tam sayı tipinde bir işaretçi tanımlamış oluruz. Bu işaretçi sadece tam sayı tipindeki değişkenleri işaretleyebilir. O halde her değişken tipi için ayrı işaretçi olması gerektiğini düşünebilirsiniz. Bu konuda haklısınız. Bazı işaretçiler şöyle olabilir.

```
int *ip;
float *fp;
double *dp;
char *cp;
long *lp;
```

İşaretçiler sadece bununla sınırlı kalmayıp aynı zamanda işaretçi dizileri veya işaretçi yapıları da olabilir. Örneğin şu şekilde bir işaretçi dizisi tanımlayabiliriz.

```
int *ptr [3];
```

Şimdilik yapı tiplerini görmesek de istersek yapı tipinde veya typedef olarak kullanıcının adlandırdığı yapı tipinde bir işaretçi tanımlamamız mümkündür. Bu işaretçi kullanıcının belirlediği yapıyı işaret eder. Veri yapılarını görürken bunun çok önemli olduğunu anlayacaksınız.

```
struct yapi *ptr_yapi;
```

Şu an yapılara geçmediğimiz için sadece örneğini vermekle yetiniyorum. Görüldüğü gibi hangi değişkeni veya yapıyı tanımlarsak tanımlayalım aynı zamanda bunun işaretçisini tanımlamamız da mümkündür.

İşaretçiler ve Diziler

İşaretçiler özellikle karakter dizileri üzerinde işlem yaparken bize büyük kolaylık sağlamaktadır. Tek tek karakterler üzerinde işlem yapılacaksa hem fonksiyona dizi işaretçi olarak aktarılmakta hem de dizinin değerlerinin kopyası üzerinden değil de adresleri üzerinden işlem yapılmaktadır. Bu da programı daha kısa ve performanslı hale getirecektir. Sadece öğreneceğimiz birkaç şey ile bizim elimizde oldukça pratik bir araç olacak ve programlamada kolaylık sağlayacaktır. Bunun nasıl olacağını size bir örnek ile gösterelim. Örneğin elimizde büyük ve küçük harflerle karışık bir yazı var ve biz bunu daha okunaklı büyük harflere dönüştürmek istiyoruz. Bunun için şöyle bir program kullanacağız.

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

void buyuk_harf( char *string );
int main(int argc, char *argv[]) {
    char string[] = "BuRaDa ApacHi GiBi YaziOz MoRuQ";
    printf("Cevirmeden Onceki String: %s \n", string);
    buyuk_harf(string);
    printf("Cevirdikten Sonra String: %s \n", string);

    return 0;
}
```



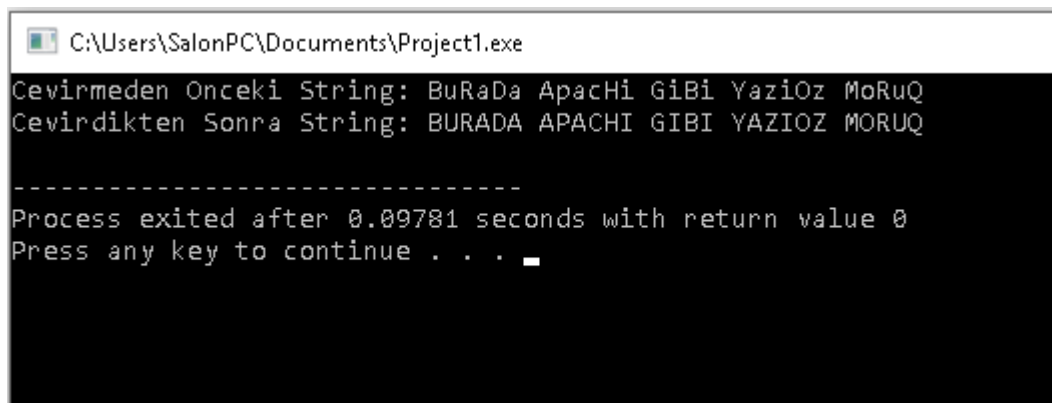
```

}

void buyuk_harf (char *string)
{
    while (*string != '\0')
    {
        *string = toupper( *string );
        ++string;
    }
}

```

Programı çalıştırdığımızda ekran görüntüsü şu şekilde olacaktır.



```

C:\Users\SalonPC\Documents\Project1.exe
Cevirmeden Onceki String: BuRaDa Apachi GiBi YazioZ MoRuQ
Cevirdikten Sonra String: BURADA APACHI GIBI YAZIOZ MORUQ

-----
Process exited after 0.09781 seconds with return value 0
Press any key to continue . . . _

```

Şimdi programın ne işe yaradığını ve nasıl çalıştığını bütün ayrıntısıyla inceleyelim. Bu programın tek amacı string adındaki karakter dizisine girilen değer ne olursa olsun bunun bütün harflerini büyük harflere çevirmektir. İstersek tamamı küçük harf istersek de yukarıdaki örnek gibi birbirine karışmış harflerden oluşabilir. Bu program bu sayede 10 yaşındaki bir gencin yazısını caps lock'u açık unutan bir ihtiyarın yazısına çevirebilir. Şimdi satır satır programın önemli kısımlarını açıklayarak bunu nasıl yaptığını sizlere anlatalım.

#include <ctype.h> C standart kütüphanesinde ctype.h adında bir başlık dosyası bulunmaktadır. Bu başlık dosyasında karakterlerle alakalı özel fonksiyonlar yer alır. Buradaki bütün fonksiyonlar karakterlerle (char) alakalı fonksiyonlardır. Örneğin karakterin sayı olup olmadığı, büyük olup olmadığı, küçük olup olmadığı büyükse küçültme veya küçükse büyültme gibi metin işlemle alakalı fonksiyonlar yer almaktadır. Bu fonksiyonlar teker teker karakter bazlı olmakta. Yani bunları bir cümle gibi bir karakter dizisine uygulamak için bir algoritma geliştirmemiz gerekmekte. Biz de burad toupper() yani büyük harf yap fonksiyonunu kullanabilmek için ctype.h dosyasını programa dahil ettik.

void buyuk_harf(char *string);

Normalde buna baktığınızda sadece bir char tipine ait bir *string adında işaretçi görürsünüz değil mi? Ama C dilinde karakter işaretçilerinin farklı bir yeri bulunmakta ve char *string bir karakter dizisi işaretçisi olmaktadır. char *string ile karakter dizisi olarak alınan argümanın birinci elemanına işarete bulunulur. Biz istersek C dilinde şu şekilde bir karakter dizisi tanımlaması da yapabiliriz.

```
char *string = "string";
```

Bu tanımlama normal dizilere göre derleyici tarafından farklı olarak ele alınmakta ve daha hızlı olmaktadır. Sabir değerleri bu şekilde kullanmak daha performanslıdır. Yeni standartlarla beraber bunun başına const getirilmesi gerektiğini unutmayın.

buyuk_harf(string); Burada string diyerek karakter dizisini fonksiyona göndersek de bunun adres mi yoksa dizinin kopyası mı olacağını fonksiyonun

aldığı argüman tipi belirlemektedir. Eğer `char * string` yerine `char string []` diye bir değer alsaydı bu sefer dizinin değerlerini argüman olarak alacaktı. Ama `char *string` diyerek dizinin adres değerini argüman olarak almaktadır.

while (*string != '\0') Burada `*string` ile dizinin değerlerine erişilmiş ve `'\0'` karakteri denetlenmiştir. C dilinde biz koymasak bile derleyici otomatik olarak dizinin sonuna `\0` karakterini koymaktadır. Bu karakter dizinin bittiğini bildirmektedir. Aksi halde dizinin bittiğinden haberimiz olmaz. Bu özel karakter aynı bize yazının bittiğini gösteren nokta işareti gibi görev yapmaktadır. Karakter dizileri üzerinde çalışırken yazacağımız her programda hemen hemen `\0` karakterinin denetimini yapmak zorunda kalırız. Önceden de söylemiştik, C dilinde her şeyi siz yapmak zorunda kalıyorsunuz.

***string = toupper(*string);** Burad `toupper()` fonksiyonundan dönen değer ilgili dizi elemanına `*` erişim operatörü vasıtasıyla yükleniyor. Aslında burada yapılan işlem oldukça basit. Dizin ilgili elemanı işaretçi erişim operatörü ile okunur ve bu `toupper()` fonksiyonu ile büyük harfe çevrildikten sonra tekrar aynı elemana yerleştirilir.

++string; Görüldüğü gibi işaretçi aritmetiği burada da karşımıza çıkmakta. Aslında programın en önemli özelliklerinden biri olarak işaretçilerin bir dizi üzerinde işlem yapabilmesini sağlamakta. Eğer bu olmasaydı tek bir karakter üzerinde işlem yapmaktan öte geçemezdik. Her döngü sırasında sıra ile birer birer artırılarak bütün dizi üzerinde işlem yapmamızı sağlıyor. Her bir artırıldığında bir sonraki dizi elemanını göstermekte bu işaretçi.

Bir sonraki yazımızda konumuza devam edeceğiz. Sıkılıyor musunuz bilmiyorum ama ben yazarken eğleniyorum. Böyle kafa yoracağınız zor

konular işi daha eğlenceli kılıyor. Programcılıkta bana kalırsa en sıkıcı şey tek düzelik ve sıradanlıktır.

-36- İşaretçi ve Değer Sabitleri

Değişkenleri size anlattığımız zaman const anahtar kelimesi ile bir değişkeni bir değişken sabiti haline getirebileceğimizi söylemiştik. Aynı özelliği işaretçiler için de kullanmamız mümkündür. Yalnız işaretçi boyutundan baktığımız zaman işaretçilerin kendisi olduğu gibi işaret ettiği eleman da söz konusudur. O halde karşımıza dört ayrı ihtimal çıkmaktadır.

- **Sabit olmayan işaretçi ve sabit olmayan değişken**
- **Sabit işaretçi ve sabit olmayan değer**
- **Sabit olmayan işaretçi ve sabit değer**
- **Sabit işaretçi ve sabit değer**

Biz daha önceki başlıklarda sabit olmayan işaretçi ve sabit olmayan değerler ile program yazmıştık. O yüzden bunu atlayarak bu konuda sadece sabit

değişken ve işaretçileri kullanarak program yazacağız. Tahmin edeceğiniz üzere sabit işaretçi tek bir değer alır ve program boyunca o değer asla değişmez. İşaretçi olduğu için tek bir adres değeri alır ve program boyunca o adres değerini gösterir. Sabit olmayan işaretçi sabit bir değeri işaret edebildiği gibi başka bir değeri de gösterebilir. Yalnız sabit değere geldiği zaman “*” operatörü ile bu sabit değişkenin değerine erişebilse de bu değer üzerinde herhangi bir değişiklik yapamaz. Sabit işaretçi ve sabit değer beraber kullanıldığı zaman ne işaretçinin adresi üzerinde ne de değişken üzerinde bir işlem yapılabilir.

Şimdi değişken bir işaretçi ile sabit bir veri örneğini yapalım.

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

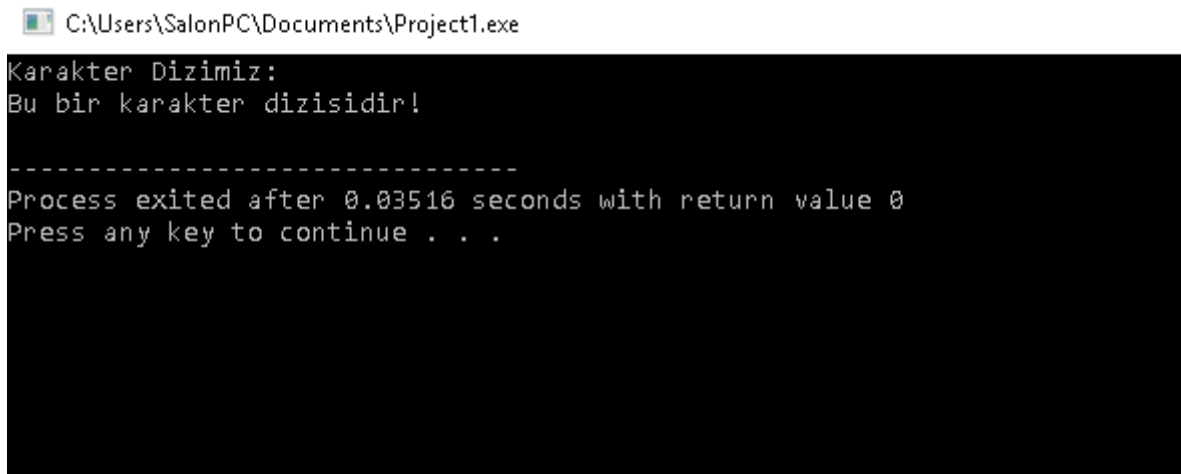
void karakterYazdir (const char *stringP);
int main(int argc, char *argv[]) {
    char string[] = "Bu bir karakter dizisidir!";
    puts("Karakter Dizimiz:");
    karakterYazdir ( string );

    return 0;
}

void karakterYazdir (const char *stringP)
{
    while (*stringP != '\0')
    {
```

```
printf("%c", *stringP);  
stringP++;  
}  
printf("\n");  
}
```

Programı çalıştırdığımızda ekran görüntüsü şu şekilde olacaktır.



```
C:\Users\SalonPC\Documents\Project1.exe  
Karakter Dizimiz:  
Bu bir karakter dizisidir!  
-----  
Process exited after 0.03516 seconds with return value 0  
Press any key to continue . . .
```

Burada sabit veri olarak belirttiğimiz karakter dizisi `main()` fonksiyonu içerisinde sabit değildir. Ama ne zaman yazdırma fonksiyonuna gelirse o zaman `void karakterYazdir (const char *stringP)` diyerek sabit hale gelmektedir. Burada sabit verinin olması işaretçinin asla veri üzerinde değişiklik yapamayacağı anlamına gelir. Bu verinin değiştirilmeden güvenle ekrana yazılmasını sağlar. Aynı zamanda bizim burada işaretçimiz sabit olmadığı için `stringP++;` komutu ile işaretçinin değerleri ile oynayabilir ve bütün diziyi yazdırabiliriz.

Sabit İşaretçi ve Değişken Değer

Sabit işaretçi ve değişken işaret edilen değer örneğinde her zaman işaretçi aynı hafıza bölümünü göstermektedir. Aynı hafıza bölümünü göstermesine karşın içindeki değeri istediği gibi değiştirebilir. Buna en iyi örnek olarak dizi

adlarını verebiliriz. Dizi adları değişmeyen birer işaretçidir ve her zaman dizinin ilk elemanını işaret etmektedir. Biz sabit bir işaretçi tanımlıyorsak muhakkak değerini ilk tanımlamada vermek zorundayız. Örnek bir sabit işaretçi tanımlaması şu şekilde olabilir.

```
int * const ptr = &degisken;
```

Sabit İşaretçi ve Sabit Değer

Bu örnekte en az erişim imkanı verilmiş olur. Sabit işaretçi aynı hafıza birimini işaret ettiği gibi sabit veri de program akışı boyunca değiştirilemez. Dizi değişkenleri fonksiyonlara argüman olarak aktarılırken bu halde aktarılır. Yani aktarılan diziden sadece okuma yapılır ve herhangi bir değişiklik yapılamaz. Sabit işaretçi ve sabit değeri şu şekilde tanımlayabiliriz.

```
const int * const ptr = &degisken;
```

Buraya kadar işaretçilere dair fonksiyon işaretçilerinden başka konumuz kalmadı. Bir sonraki konuda onu ele aldıktan sonra işaretçiler konusunu bitireceğiz. Gördüğünüz gibi hiç de zor değilmiş, değil mi?

-37- Fonksiyon İşaretçileri

Daha öncesinde sizlere diziler ve fonksiyon adlarının aslında birer adres verisi bulunduran bir işaretçi olduğunu söylemiştik. Yani adres operatörü ile (&) dizinin adını yazdığımız zaman bize dizinin adresini vermektedir. Aynı işlemi fonksiyon adına uyguladığımızda ise fonksiyonun hafızadaki adresini almaktaydık. Bu konu hakkında yaptığımız uygulamada hep beraber bunu görmüştük.

İşaretçiler hakkında son konumuz ise bu sebepten dolayı fonksiyon işaretçileri olmaktadır. Fonksiyon işaretçisi fonksiyonun hafızadaki adresini bize vermektedir. Aynı dizilerde olduğu gibi fonksiyonun başlangıç adresini &fonksiyonadi şeklinde bir operatör kullanımı ile alabilme imkanımız vardır.

Neden normal fonksiyon çağırma işlemi yerine fonksiyon işaretçilerini kullanmamız gerektiğini sorabilirsiniz. Normalde biz fonksiyonu çağırırken fonksiyonadi(); şeklinde biz söz dizimi kullanarak kolaylıkla işi halletmekteyiz. Bunun sebebi bazı durumlarda fonksiyon işaretçilerinin bize büyük bir esneklik sağlamasıdır. Örneğin bir diziyi sıralama algoritmasında sıralamanın artan veya azalan olacağını fonksiyon işaretçileri ile yapabiliriz.

Fonksiyon işaretçilerinin söz dizimi şu şekildedir.


```
void (*pf) (int);
```

Buradan bakıldığında biraz kafa karıştırıcı gibi görülebilir. Alışıldık C sözdiziminden uzakta görünen bir yapı ile karşı karşıyayız. Burada öncelikle *pf tanımlamasından başlayarak içten dışa doğru inceleyelim. *pf bir fonksiyonu işaret eden işaretçidir, void fonksiyonun döndürdüğü değer tipi ve int ise fonksiyonun argüman tipidir.

Şimdi fonksiyon işaretçisine işaretçileri ekleyelim ve tekrar okuyalım.


```
char* (*pf) (int*);
```

Burada *pf yine fonksiyon işaretçisidir*, *char* fonksiyonun döndürdüğü değer tipi, *int** ise argüman tipidir. Şimdi fonksiyon işaretçilerinin kullanıldığı örnek bir kod yazıp kullanımına bakalım.

```
#include <stdio.h>
void fonksiyonumuz(int deger)
{
    printf("Fonksiyon Cagirildi Arguman: %d\n", deger);
}

main()
{
    void (*pf)(int);
    pf = &fonksiyonumuz;
    printf("Simdi Fonksiyonu Cagiriyoruz. \n");
    (pf)(5);
    printf("Fonksiyondan donuldu");
}
```

Burada görüldüğü gibi bir fonksiyon işaretçisi tanımladıktan sonra ilginç bir şekilde (pf) (5); şeklinde fonksiyon çağırıyoruz! C dilinde size en tuhaf görünen biçimlerden biri olacağı garantisini verebilirim. Bu sayede işaretçiler vasıtasıyla çağırdığımız fonksiyon argüman olarak 5 değerini almakta ve ekrana yazdırmakta. Programın ekran çıktısı şu şekilde olacaktır.

 C:\Users\SalonPC\Documents\Project13.exe

```
Simdi Fonksiyonu Cagiriyoruz.  
Fonksiyon Cagirildi Arguman: 5  
Fonksiyondan donuldu  
-----  
Process exited after 0.08419 seconds with return value 20  
Press any key to continue . . .
```

Aşağıda verdiğim kaynaklarda fonksiyon işaretçileri ile alakalı örnekler yer almaktadır. Konunun devamını merak ederseniz aşağıdaki örneklere göz atabilirsiniz.

Kaynaklar,

https://www.learn-c.org/en/Function_Pointers

<https://www.geeksforgeeks.org/function-pointer-in-c/>

Deitel, C How To Program 7th. ed, 2012, sf 338 vd.

-38- Karakterler ve Karakter Dizilerine Giriş

Bilgisayarların en büyük görevlerinden biri de bizim günlük hayatta kullandığımız yazıları okuyup, işleyip, kaydedebilmesidir. Eğer bilgisayarlar metin üzerinde işlem yapmasaydı hesap makinesinden çok da farklı olmazdı. Bu site sayfasında okuduğunuz yazılar, klavyeden sohbet uygulamasında yazdığınız mesajlar, telefonda attığınız SMS mesajı gibi bütün yazılar bilgisayarlarda karakter yani harf harf olarak ele alınır ve bu harflerin sırasına ise karakter dizisi (string) adı verilir.

Bilgisayar bilimlerinden ve binary kodlamadan bahsettiğimiz yazılarda bilgisayarların sadece ikilik sistemde bir (1) veya sıfır (0) değerleriyle işlem yaptığından bahsetmiştik. Bu ikilik tabandaki değerler sayı değeri olduğu için bunları onluk veya on altılık tabana çevirebiliriz. Fakat bizim yazıda kullandığımız harfler, semboller ve rakamlar bilgisayar sistemlerinde bir anlam ifade etmemektedir. O yüzden kodlama adını verdiğimiz ve sadece bizim anladığımız özel bir kodlama sistemi mevcuttur. Bu kodlama ile belli bir sayısal değere karşılık belli bir harf değeri temsil edilir. Örneğin bilgisayar sisteminde 'A' karakterinin karşılığı 65 değeridir. Program yazdırma formatına göre bu 65 değerini sayı olarak da ekrana yazdırabilir metin olarak da. Ekranda görülen A değeri bizim için bir anlam ifade etse de bilgisayar için anlam ifade eden 65 değeridir. Daha ayrıntılı bilgi için ASCII tablosuna bakmanızı rica ederim.

C dilinde karakter verisinin 8-bitlik bir deęer bulundurduęunu söylemiřtik. Aslında char adı koyulan veri tipi bir baytlık tamsayı deęiřkeninden ok farklı bir řey deęildir. İstersek char deęiřkenine sayı deęeri koyar ve sayı deęerleri zerinden iřlem yaparız. Ama adından da anlařılacaęı zere char deęiřken tipinin asıl amacı 0-127 arası ana ASCII karakterleri ya da unsigned char ile 0-255 arası geniřletilmiř ASCII karakterleri iinde bulundurmadır. Her char deęiřkeni sadece bir karakteri iinde bulundurabilir. rneęin “STM32” yazdırmak iin 5 ayrı char deęiřkenine ihtiyaımız vardır. Bunu istersek ayrı ayrı deęiřkenlerle yazdırabiliriz ki ancak bizim iin bir anlam ifade edebilir. İstersek de char dizi[6] diyerek bir char tipinde dizi tanımlarız ve sıra ile yazdırırız. Dizi olarak yazdırmak hem daha pratik hem de bilgisayar iin daha anlamlı olacaktır. Birbirinden baęımsız deęiřkenler ancak bizim iin bir anlam ifade edebilir. Bilgisayar bunların arasındaki baęlantıyı kavrayamaz.

Dięer yksek seviye programlama dillerinde karakter dizileri iin String adında sınıf yapıları bulunabilir ve daha yksek seviye iřlemler yapılabilir. Ama C dilinde String adında bir yapı yoktur!. Evet, C dilinde siz karakter dizilerini sıfırdan yapmak zorundasınız. Fakat dilde gml olarak karakter dizileri iin zel ayrıcalıklar vardır. Mesela karakter dizilerine deęer atarken iki tırnak arasına deęer girmek veya karakter dizilerinin otomatik olarak \0 bitirme karakteri ile bitmesi gibi bazı ayrıcalıklar dıřında C dilinin standart ktphanelerinde karakter dizileri ve karakterler iin zel fonksiyonlar bulunmaktadır. Bu fonksiyonları kullanmak metin iřlemenin olmazsa olmazıdır.

İsterseniz C dilindeki standart ktphane fonksiyonlarına benzer fonksiyonları siz de yazabilirsiniz. Ama bu standart ktphane fonksiyonları zenle hazırlandıęı iin performans ve kararlılık bakımından st seviyededir. ok zorunda kalmadıķça hazır fonksiyonları kullanmalısınız.

C dilinde karakter dizileri için üst seviye özellikler olmadığından çok uğraştırıcı ve teferruatlı gibi görünebilir. Fakat burada üstü kapalı hiçbir yer yoktur. Perde arkasında yürüyen hiçbir işlem olmadığı için mantığını kavramanız oldukça kolay olacaktır. Bir sonraki yazımızda örneklerle konumuza devam edeceğiz. Tekrar belirtelim, bu konu zor değil fakat teferruatlı. O yüzden zor gelmese de sıkıcı gelebilir.

-39- Karakter Dizileri (String)

Bu başlıkta C dilinde karakter dizilerine giriş yapacağız ve size uygulamalı olarak göstereceğiz. Karakter dizileri daha önce söylediğimiz gibi karakter tipindeki bir dizi yapısından oluşmaktaydı. Bunun mantığını anlamanız aslında

çok kolaydır. Daha önce söylediğimiz gibi üstü kapalı hiçbir yer yoktur ve oldukça basittir. Mantığını kavramanız kolay olsa da sonrasında karşımıza pek çok metin işlem fonksiyonu çıkacaktır. Bu fonksiyonlar karakter dizileri üzerinde işlem yapmamız için gereklidir. Hepsini ezberlemenize gerek olmasa da elimizin altındaki fonksiyonların ne yaptığını ve ihtiyacınız olduğunda nerede bulacağınızı bilmeniz gerekir. Öncelikle karakter dizilerinin mantığını anlatmakla konuya devam edelim.

Nedir Bu Karakter Dizileri?

Karakter dizileri (string) belli sayıda bir karakter verisinin sıralanmış haline denir. C dilinde bu diziler vasıtasıyla tanımlanır ve veri tipi char olmak zorundadır. Bazı dillerde String sınıfı olup karakter dizileri aynı değişkenler gibi kullanılmakta hatta “+” operatörü ile karakter dizileri birbirine eklenmektedir. C dilinde ise bu tarz yerleşik özelliklerin yerine standart kütüphanede gelen karakter dizileri ve karakterlerle alakalı fonksiyonlar kullanılır. Dil yapısına yerleşik bir karakter dizisi yoktur.

Bilmeniz gereken en önemli noktalardan biri ise C dilinde bütün karakter dizilerinin “\0” karakteri ile bittiğidir. Bu karakter ekranda yazdırılmaz fakat program tarafından okunur ve karakter dizisinin bittiği böylece anlaşılır. Yazdığınız programlarda sürekli “\0” karakterini denetleyen bir şart ifadesi kullanmak zorunda kalırsınız. Çünkü C dilinde dizinin bittiği ancak böyle anlaşılmaktadır. “\0” karakterini biz kendimiz koymayız derleyici otomatik olarak bunun karakter dizisi olduğunu anlar ve koyar. Örnek bir karakter dizisi tanımlaması şu şekillerde olabilir.

```
char kart[] = "Arduino";
```

```
const char *kartptr = "Arduino";
```

Görüldüğü gibi tanımladığımız bir dizi veya işaretçi olmasına rağmen buna “Arduino” şeklinde bir karakter dizisi değerini ekleyebiliyoruz. Normalde dizide her bir değere ayrı ayrı değer vermemiz gerekirken burada bir istisna olarak bu şekilde yazabiliyoruz. Eğer böyle olmasaydı karakter dizilerine veri atamak aşırı derecede zor olurdu. Aynı şekilde bir işaretçiye de normalde ancak adres değeri atayabilirken char olması durumunda “Arduino” şeklinde karakter dizisi değeri aktarabiliriz. Elbette bu değer aktarma işleminin görünüşte olduğunu unutmayın. İşin aslında hafızada “Arduino” diye bir karakter dizisi sabiti kayıt edilir ve işaretçi bunun ilk elemanını (A harfini) işaret eden bir değere sahip olur. “Arduino” değerini yazdırmak istediğimizde A, r, d, u, i, n, o diye harfler sırayla yazdırılmaya devam eder ta ki ‘\0’ karakterine gelene kadar. Eğer ‘\0’ karakteri okunmazsa program nerede duracağını bilemez. Bunlar istisnadır ve “\0” bitiş karakterinin yanında diziler hakkında bilmeniz gereken en önemli bilgilerdendir.

Eğer biz isteseydik normal bir dizi tanımlar gibi de karakter dizisini tanımlayabilirdik. Bu C dilinin alışlageldik yapısına uygun olsa da oldukça kullanışsız olacaktır. Yine de böyle bir yaklaşım bunun mantığını kavramanız için oldukça önemlidir. Şimdi yukarıda `char kart[] = "Arduino";` diye tanımladığımız karakter dizisinin aslında ne olduğunu görelim.

```
char kart [] = { 'A', 'r', 'd', 'u', 'i', 'n', 'o', '\0' };
```

Eğer biz bilindik bir şekilde böyle tanımlamak istesek kendimiz ‘\0’ karakterini koymak zorunda kalırız. Çünkü derleyici bunun karakter dizisi olduğunu anlamaz ve bütün denetimi bizim elimize bırakır. Yukarıda daha pratik bir şekilde tanımladığımız karakter dizisi aslında bu şekildedir. Her harf birer char değeri olarak karşımıza çıkar ve sonundaki bitirme karakteri ‘\0’ ise yine özel bir komut sembolü olarak ASCII tablosunda yer almakta. Bütün bilgisayarlar

'\0' komutunu okuduğunda karakter dizisinin bittiğini anlayacaktır. Biz normalde program yazarken böyle bir karakter dizisi tanımlaması kullanmayız. Yukarıda verdiğim iki örnekten birini kullanırsınız. Eğer değişken karakter dizisi olacaksa *char isim[]* şeklinde eğer sabit bir karakter dizisi olacaksa *const char *ptr* şeklinde tanımlayabilirsiniz. C dilinde iki tırnak işareti ("") arasına yazılan harf, rakam ve sembollerin de karakter dizisi olarak değerlendirildiğini biliyoruz. Biz bunu hiç '\0' karakterini düşünmeden tanımlasak da program işleyişinde '\0' karakteri sıklıkla karşımıza çıkacaktır.

Size karakter dizilerini tanımlamakla alakalı son bir ipucunu vereceğim. Yukarıda verdiğimiz iki tanımlamada da karakter dizileri ilk değerini almakta ve buna göre tanımlamada boyutlanmaktaydı. Fakat bir program başında boş bir karakter dizisi tanımlayabilir program akışında kullanıcıdan veya çevreden alacağımız veri ile bunu doldurmak isteyebiliriz. Örneğin bir mikrodenetleyici seri iletişim vasıtasıyla aldığı baytları (haliyle karakterleri) bir tampon belleğe kaydetme ihtiyacı duyabilir. Bu tampon bellek ise normalde boş olacak fakat gelen veriye göre sırayla dolması gerekecek. Bu durumda bir makul bir boyutta boş bir karakter dizisi tanımlayabiliriz. Bunu da şöyle yaparız. C++

```
char buffer [50] = "";  
// ya da  
char buffer [50] = {'\0'};
```

Burada statik veri tahsisinde (allocation) geçerli olan tanımlamaları verdik. Dinamik veri ileri bir konu olduğu için gerektiği zaman bundan bahsedeceğiz. Buraya kadar anladıysanız bundan sonrası C standart kütüphanelerini kullanarak ve yazdığımız algoritmalarla metin işlem uygulamaları yapmaktır.

-40- ctype.h Başlık Dosyası ile Karakter İşlemleri

Artık karakter dizilerini anlattığımıza göre bunlar üzerinde nasıl çalışabileceğimizi göreceğiz. Bunun için bol bol kütüphane fonksiyonu

kullanmamız gerekecektir. Hatta bazı kütüphane fonksiyonları bizim için hayati bir öneme sahiptir ve onlar olmadan bazı noktalarda tıkanıp kalabiliriz. Örneğin bir veri iletişimde tam sayı tipindeki bir değeri karakter dizisine çevirmemiz gerekebilir veya karakter dizisini üzerinde aritmetik işlem yapabilmeniz için tam sayı değere çevirmeniz gerekebilir. Unutmayın, “10” değeri ile 10 değeri aynı değildir siz örneğin bir seri iletişim uygulamasında bunu “10” diye alsanız da üzerinde işlem yapmak için tam sayıya çevirmeniz gerekecektir.

Bunun için C dilinde standart kütüphanenin içerisinde karakterler ve karakter dizileriyle alakalı fonksiyonlar gelmektedir. Karakter dizileri üzerinde çalışırken içli dışlı olacağımız kütüphane dosyalarını baştan belirtelim:

- **stdio.h**
- **stdlib.h**
- **ctype.h**
- **string.h**

Bizim bundan sonra anlatacağımız konular bu başlık dosyalarında yer alan fonksiyonları size öğretme yönünde olacak. Fakat siz program yazarken referanstan sürekli bu başlık dosyalarına bakmalı ve içindeki fonksiyonları sentaksa uygun olarak yazdığınıza emin olmalısınız. Kimse ezberden program yazmıyor. Bu tarz kütüphane dosyaları kullanılırken muhakkak kütüphanenin referans kılavuzuna bakmamız gerekiyor. Sizin burada öğrenmeniz gereken hangi iş için hangi fonksiyonun kullanılacağı, bu fonksiyonların neler yapabildiği, elimizde ne tarz araçlar olduğudur. Çünkü bu fonksiyonlardan bazılarını ayda belki yılda bir kere kullanmanız gerekebilir. Bu durumda işiniz düştüğünde bunun nerede olduğunu bilmeniz sizin için yeterli olacaktır.

Karakter İşlem Kütüphanesi (ctype.h)

C dilinin standart kütüphanesinde ctype.h adında bir başlık dosyası yer almaktadır. Bu başlık dosyasının özelliği bünyesinde sadece karakter işlem fonksiyonlarını bulundurmasıdır. Sadece karakterler üzerinde işlem yapabilseniz de karakter dizilerinin karakterlerden meydana geldiğini unutmayınız. Yani bir döngü yardımıyla bütün bir karakter dizisi üzerinde teker teker işlem yapabiliriz. Bu fonksiyonlar dilin çekirdek yapısına ait fonksiyonlar değildir. O yüzden standart kütüphane içerisinde zikredilmektedir. C dili kendi haliyle tamamen çıplak bir dildir ve herhangi bir ek fonksiyon, kütüphane veya sınıf içermez. Ekrana yazı yazdırmak için bile bir kütüphane fonksiyonu kullandığımızı daha önceden anlatmıştık. ctype kütüphanesini kullanabilmek için öncelikle #include ile ana programa dahil etmemiz gerekir.

```
#include <ctype.h>
```

Bundan sonrası kütüphane fonksiyonlarına bakmak ve ihtiyaç halinde bu fonksiyonları kullanmaktır. Fonksiyonları akılcıca kullanmak ise tamamen size bağlıdır. Hangi fonksiyonu nerede, hangi yöntem için kullanacağınızı referansa bakarak öğrenseniz de bunu doğru yerde kullanabilmek yine size bağlıdır. Şimdi ctype.h başlık dosyasında yer alan fonksiyonların işlevlerine kısaca bakalım. Bunun için ben şu bağlantıda yer alan referans sayfasını açıyorum ve fonksiyonlara bakıyorum. Kimse ezberden bunları yazamaz!.

<http://www.cplusplus.com/reference/cctype/>

ctype.h fonksiyonları

ctype.h kütüphane dosyasında yer alan kütüphane fonksiyonları şu şekildedir.

isalnum	Karakterin alfanümerik olup olmadığını denetler.
isalpha	Karakterin harf olup olmadığını denetler.
isblank	Karakterin boşluk olup olmadığını denetler.
iscntrl	Karakterin kontrol karakteri olup olmadığını denetler.
isdigit	Karakterin rakam olup olmadığını denetler.
isgraph	Karakterin grafik temsili olup olmadığını denetler.
islower	Karakterin küçük olup olmadığını denetler.
isprint	Karakterin yazdırılabilir olup olmadığını denetler.
ispunct	Karakterin noktalama işareti olup olmadığını denetler.
isspace	Karakterin beyaz boşluk olup olmadığını denetler.

isupper	Karakterin büyük olup olmadığını denetler
isxdigit	Karakterin on altılık bir rakam olup olmadığını denetler.
tolower	Karakter büyükse küçük yapar.
toupper	Karakter küçükse büyük yapar.

Görüldüğü gibi fonksiyonların işlevi gayet basittir. Bu fonksiyonları metin işlemede gerektiği zaman kullanmanız gerekecektir. Örnek vermek adına basit bir örnek yazıyorum.

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

main()
{
    char c = 'a';
    if (islower(c))
        puts("c adli degisken kucuk harftir");
    else
```

```
puts("c adli degisken buyuk harftir");  
}
```

Program çalışınca ekran görüntüsü şu şekilde olacaktır.



Siz de lazım olduğu zaman ctype.h kütüphane dosyasındaki fonksiyonları referansa bakarak kullanabilirsiniz. Ben C referansı yazmadığım ve temel C programlama eğitimi verdiğim için referans sayfalarını nasıl okuyacağınızı öğretmek istiyorum. Bunu C standartından da okuyabilirsiniz. Standartlar programlama dilleri için birinci elden kaynaktır. Biz daha pratik olan referanslardan ctype.h içindeki isalnum fonksiyonuna bakalım.

isalnum

Defined in header <ctype.h>

```
int isalnum( int ch );
```

PROTOTİP

AÇIKLAMA

Checks if the given character is an alphanumeric character as classified by the following characters are alphanumeric:

- digits (0123456789)
- uppercase letters (ABCDEFGHIJKLMNOPQRSTUVWXYZ)
- lowercase letters (abcdefghijklmnopqrstuvwxyz)

The behavior is undefined if the value of ch is not representable as unsigned char.

Parameters

PARAMETRELER

ch - character to classify

DÖNDÜRDÜĞÜ DEĞER

Return value

Non-zero value if the character is an alphanumeric character, 0 otherwise.

Her dil referansında veya kütüphane kılavuzunda belli başlı öğelerin olduğunu görürüz. Bunlardan ilki fonksiyon prototipi adını verdiğimiz fonksiyonun iskeletidir. Bu prototip ile fonksiyonu nasıl kullanacağınız belli olur. Fonksiyon adı ne işe yaradığına dair bir ipucu olsa da bu prototipte yer alan argümanların adı da yine aldığı değerlerin ne olduğuna dair bir ipucu olacaktır. Fakat bu prototipe bakarak fonksiyonun ne işe yaradığını öğrenemezsiniz. Burada öğreneceğiniz şey fonksiyonunun döndürdüğü değer, aldığı argüman sayısı, sırası ve tipi ve fonksiyon adıdır. Bu sayede düzgün bir sözdizimi ile fonksiyonu kullanabilirsiniz.

Açıklama kısmında ise (bazı kaynaklarda description diye başlık konur.) fonksiyonun ne işe yaradığı açıklanır. Kaynaklar programcılara yönelik yazıldığından uzun uzadıya fonksiyon mantığı, nasıl kullanacağınız, örnek kod gibi şeylerle açıklanmaz. Genelde fonksiyonun ne işe yaradığı bir veya iki cümle ile açıklanır ve geçilir. Eğitim içeriklerinde bu daha ayrıntılı açıklanmaktadır.

Parametreler ise fonksiyonun aldığı argümanların açıklamasıdır. Fonksiyonun açıklaması ne işe yaradığı yönünde parametreler de argümanların ne olması gerektiği yönündedir. Bu açıklama sayesinde fonksiyonun kaynak kodunu alıp incelemek zorunda kalmazsınız.

Return value ise fonksiyonun döndürdüğü değerdir ve dokümentasyonda en son olarak bu açıklanır. Eğer gerek duyulursa bazı kaynaklarda beraberinde örnek kodla gelebilir. Ama genelde örnek kod olmadan bunları okuyup fonksiyonu sorunsuzca kullanabilmeniz gerekir.

Unutmayın, Bunları okuyup fonksiyonu kullanamıyorsanız programcı olamazsınız!!!

-41- string.h Başlık Dosyası ile Karakter Dizisi İşlemleri

Daha öncesinde size ctype.h başlık dosyasını anlatmış fakat karakter dizileri için stdio.h, stdlib.h ve string.h başlık dosyalarının da kullanıldığını söylemiştik. Şimdiden belirtelim ki bütün fonksiyonlar için örnek uygulama vermeyeceğim. Sadece önemli gördüklerim hakkında uygulama göstereceğim ve geri kalanı denemek üzere size bırakacağım. Bu kütüphane fonksiyonları dilin teferruatı olduğu için uzun uzadıya anlatmaya gerek yoktur. Çünkü programcılıkta karşınıza belki yüzlerce kütüphane dosyası ve binlerce fonksiyon çıkacaktır. Karşınıza çıkabilecek bütün fonksiyonları anlatan bir eğitim veya öğretecek bir öğretmen bulma imkanınız yoktur. Dilin temelini bir eğitim ile alsanız dahi karşınıza çıkacak kütüphanelerin fonksiyonlarını kullanmayı sürekli öğrenmeniz gerekecektir. Ben de bunu göz önünde bulundurarak dilin temelini uzun uzadıya anlatsam da ayrıntısı olan bu tarz fonksiyonları yorulmamak adına kısa kesiyorum. Bu asla bilgimi sakladığım anlamına gelmesin.

string.h dosyası

Karakter dizisi kütüphanesi (<string.h>) karakter dizileriyle alakalı pek çok kullanışlı fonksiyonu bünyesinde bulundurur. Bunlardan başlıcaları karakter dizisi kopyalama, karakter dizisi birleştirme, karakter dizilerini karşılaştırma gibi fonksiyonlardır. Daha önce söylediğimiz gibi C dilinde karakter dizileri üzerinde operatörler ile işlem yapamadığımız için bu tarz fonksiyonlar vasıtasıyla bu işlemleri yapmaktayız. Şimdi string.h kütüphanesinin

fonksiyonlarının ne işe yaradığını baştan sona verelim ve sonrasında örneklerle geçelim.

Fonksiyon Adı	İşlev
memcpy	Bir hafıza blokunu kopyalar.
memmove	Bir hafıza blokunu taşır.
strcpy	Bir karakter dizisini kopyalar.
strncpy	Karakter dizisindeki karakterleri kopyalar.
strcat	Karakter dizilerini birleştirir
strncat	Karakter dizisindeki karakterleri birleştirir.
memcmp	İki hafıza blokunu karşılaştırır
strcmp	İki karakter dizisini karşılaştırır.
strcoll	İki karakter dizisini C yerelini kullanarak karşılaştırır. (locale)
strncmp	İki karakter dizisinin karakterlerini karşılaştırır.
strxfrm	C yereli kullanılarak (locale) karakter dizisi dönüştürülür.
memchr	Bir hafıza blokundaki bir karakter tespit edilir.

strchr	Karakter dizisindeki belirtilen karakterin ilk konumu tespit edilir.
strcspn	Karakter dizisinde belli bir karakter aranır ve ilk konumu tespit edilir.
strpbrk	Karakter dizisi içindeki karakterler tespit edilir.
strrchr	Karakter dizisindeki belli bir karakterin en son yer aldığı konum tespit edilir.
strspn	Karakter dizisinde belli bir karakter dizisi aranır ve ilk konumu tespit edilir.
strstr	Karakter dizisi içinde alt karakter dizisi aranır.
strtok	Bir karakter dizisi parçalara bölünür.
memsek	Bir hafıza blokunu doldurur.
strerror	Hata mesajını içeren karakter dizisinin işaretçisi tespit edilir.
strlen	Karakter dizisi uzunluğu tespit edilir

NULL (makro)	BOŞ İŞARETÇİ
size_t (TİP)	sizeof operatörünün döndürdüğü değer tipi

Bir karakter dizisinin uzunluğunu bulma

Elimizde bir karakter dizisi var fakat bunun uzunluğunu bilmiyoruz. Uzunluğu dizinin kaç karakter içerdiği demektir ve bazen bunu öğrenmemiz önemli olabilir. Bunun için bir algoritma kurar ve dizi işaretçisinden (sıfırıncı eleman) itibaren bitirme karakterini ('\0') bulana kadar saydırabiliriz. Fakat daha kolayı strlen() fonksiyonunu kullanmaktır. Bu fonksiyon bize dizinin uzunluğunu hemen verecektir.

Fonksiyonu kullanmadan önce hemen fonksiyonun prototipine bakalım. Ne kadar ne işe yaradığını bilsek de ezbere kullanmak doğru olmaz.

```
size_t strlen ( const char * str );
```

Görüldüğü gibi strlen fonksiyonu tahmin ettiğimiz gibi bir karakter dizisi değeri olsa da karakter dizisinin boyutunu size_t biçiminde geri döndürmekte. Bu değeri geri dönen değeri (unsigned) çeviricisi ile işaretsiz tamsayıya çevirmemiz gerekir. Ya da %u yerine %zu yazmamız gerekiyor. Bunu dikkate alarak bir program yazalım.

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

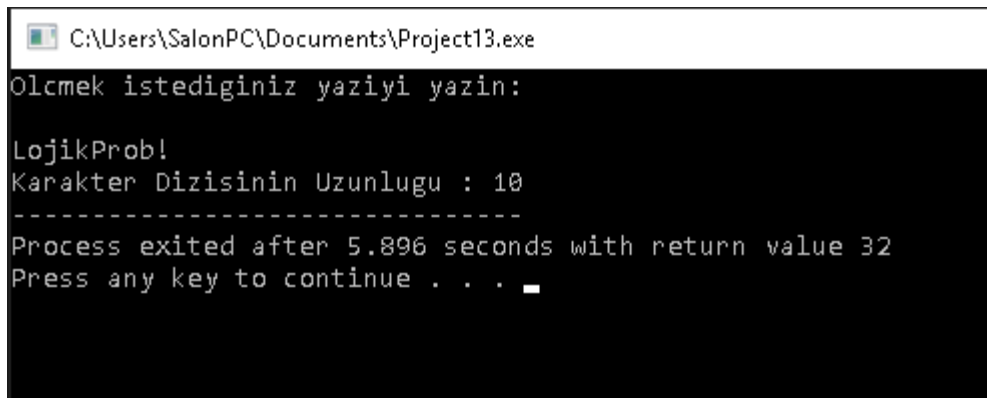
main()
```

```

{
    char karakter_dizisi[100] = "";
    puts("Olcmek istediginiz yaziyi yazin:\n");
    scanf("%s", karakter_dizisi);
    printf("Karakter Dizisinin Uzunlugu : %u",
        (unsigned)strlen(karakter_dizisi));
}

```

Programı çalıştırdığımızda ekran görüntüsü şu şekilde olacaktır.



```

C:\Users\SalonPC\Documents\Project13.exe
Olcmek istediginiz yaziyi yazin:
LojikProbl
Karakter Dizisinin Uzunlugu : 10
-----
Process exited after 5.896 seconds with return value 32
Press any key to continue . . .

```

Görüldüğü gibi girdiğimiz metnin kaç karakterden oluştuğu burada yazdırılmakta. Yalnız strlen() fonksiyonunun bir eksikliği vardır o da boşluk bıraktığımız metinlerde boşluktan sonrasını okuyamamasıdır. Daha önceden sizeof operatörü ile de dizi boyutunun bulunduğu bahsetmiştir. strlen burada bitirme karakterini saymadan dizi uzunluğunu bize vermektedir. sizeof ise bitirme karakterini ('\0') saymaktadır. Yukarıdaki programı eğer sizeof ile yapmış olsaydık 11 sonucunu elde ederdik.

Karakter dizilerini birleştirme

İki karakter dizisini birleştirmek için “+” işaretini kullanamayacağınızı biliyorsunuz değil mi? Bu bazı dillerde mümkün olsa da (mesela C++) C dilinde operatörleri aşırı yükleme denilen bir kavram yoktur. O yüzden dilin

temel öğeleri ile halledilmeyen bütün işleri kısa yoldan fonksiyonlarla halletmemiz lazımdır. Bunun için strcat() fonksiyonunu kullanacağız. Öncelikle fonksiyonun ne olduğunu ve nasıl kullanıldığını öğrenmek için referanstan fonksiyon prototipine bakıyoruz.

```
char * strcat ( char * destination, const char * source );
```

Burada görüldüğü gibi bir kaynak bir de hedef olarak iki argüman almakta. strcat bir değer döndürse de bu referansta destination değeri olduğu belirtilmekte. Yani bu fonksiyon sonuc = a + b gibi bir işlem yapmak yerine a+=b gibi bir işlem yapmakta. Yani bizim destination olarak belirttiğimiz eklenecek kısım kalıcı değişikliğe uğramakta. Bunu düşünerek şöyle bir program yazalım.

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

main()
{
    char s1 [] = "AVR";
    char s2[] = "Mikrodenetleyici";
    printf("Ilk Karakter Dizisi : %s \n", s1);
    printf("Ikinci karakter dizis: %s \n",s2);
    strcat(s1, s2);
    printf("Birlestirilmis: %s",s1);
}
```

Program çalıştığında ekran görüntüsü şöyle olacaktır.

C:\Users\SalonPC\Documents\Project13.exe

```
Ilk Karakter Dizisi : AVR
ikinci karakter dizisi: Mikrodnetleyici
Birlestirilmis: AVR Mikrodnetleyici
-----
Process exited after 0.02899 seconds with return value 35
Press any key to continue . . .
```

Peki biz strcat() fonksiyonunu kullanıyoruz ama arka planda neler işliyor diye sorabilirsiniz. Eğer böyle soruyorsanız iyi bir öğrencisiniz demektir! Arduino kitlesinden gördüğüm üzere pek çok Arduino kullanıcısı digitalWrite(), digitalRead() gibi fonksiyonlar ile program yazarken bu fonksiyonların arka planında neler olduğunu bir kere bile merak edip öğrenmek istememiştir. Merak edip işin derinini öğrenenler ise başarıya ulaşmıştır. Diğerleri ise yerinde saymaya devam etmektedir. O yüzden işin derinini merak edip öğrenmek isteyen hiçbir zaman zarar etmez. Bu soruya cevap vermek istersek strcat fonksiyonu bizim kullandığımız derleyicilerde kaynak kod olarak yer almamaktadır. Kendisi standart kütüphane içerisinde olduğundan derlenmiş halde gelir ve linker (bağlayıcı) vasıtasıyla bizim yazdığımız koda bağlanır. Böylelikle en performanslı şekilde kullanılmış olur. Fakat biz istersek internette çeşitli C derleyicilerinin kaynak kodlarını bulabiliriz. Her derleyicide bu fonksiyon birebir aynı değildir. Hatta yazıldığı sisteme göre bu kütüphane fonksiyonları değişiklik göstermektedir. Örneğin avr-gcc (AVR için C derleyicisi) derleyicisinde printf fonksiyonunu kullanmayı isteyeceğinizi hiç düşünmüyorum. Çünkü AVR mikrodnetleyicilerde çıktı alabileceğiniz bir ekran yoktur. Benim bulduğum strcat için kaynak kodları şöyledir.

```
char *
STRCAT (char *dest, const char *src)
{
```

```
strcpy (dest + strlen (dest), src);  
return dest;  
}
```

Kaynak : <https://code.woboq.org/userspace/glibc/string/strcat.c.html>

Burada strcat fonksiyonu strcpy fonksiyonu kullanılarak oluşturulmuş. Aslında burada iş yapan fonksiyon strcpy fonksiyonuyken bu fonksiyon strcat bloku içerisinde standarda göre uyarlanmıştır. Şimdi ise gerçek bir fonksiyona bakalım.

```
char *  
strcat(char *dest, const char *src)  
{  
    size_t i,j;  
    for (i = 0; dest[i] != '\0'; i++)  
        ;  
    for (j = 0; src[j] != '\0'; j++)  
        dest[i+j] = src[j];  
    dest[i+j] = '\0';  
    return dest;  
}
```

Kaynak: https://en.wikibooks.org/wiki/C_Programming/string.h/strcat

Burada strcat fonksiyonunun bir döngü vasıtasıyla yapıldığını görmekteyiz. Elbette ki bütün bu kaynak kodlar derleyiciden derleyiciye değişmektedir. GCC derleyicisinde farklı Microsoft'un derleyicisinde farklı olabilir. Önemli nokta hepsinin standarda uymak zorunda olmasıdır. Eğer kendinizi derleyici yazarlardan daha iyi görüyorsanız bu fonksiyonları kullanmak yerine kendiniz de yazabilirsiniz.

String.h dosyasındaki fonksiyonlara ait kullanım, açıklama ve örnek kodları aşağıdaki referans sayfasından bulabilirsiniz.

<http://www.cplusplus.com/reference/cstring/>

-42- stdlib.h ve stdio.h Başlık Dosyaları ile Karakter Dizisi İşlemleri

Karakter dizileri ile ilgili size anlatacağımız en son konu ise stdlib.h ve stdio.h kütüphane başlık dosyalarında yer alan karakter dizileri ile alakalı fonksiyonlardır. Bu fonksiyonlar arasında sıkça kullanacağınız ve önemli fonksiyonlar da yer almaktadır. Özellikle tamsayı, float veya diğer tipteki değişkenleri karakter dizisine çevirme veya karakter dizilerinin içinde yer alan tam sayı, ondalıklı sayı gibi değerleri aslına çevirme konusunda oldukça kullanışlı fonksiyonlar yer almaktadır. Karakter dizisi konusunda belki de size en çok lazım olacak fonksiyonlar bunlardır.

stdio.h başlık dosyası ve karakter dizisi işlemleri

stdio.h dosyası giriş ve çıkış işlemlerini yerine getiren fonksiyonları barındıran bir kütüphane dosyasıdır. Hepinizin şimdi bileceği gibi printf() ve scanf() fonksiyonları bu dosyaya dahildir. C dilinde kütüphane dosyaları kullanılmadan ekrana yazı bile yazdırılamayacağını söylemiştik. Nitekim bazı sistemlerde ekran olmayabilir ve ekrana yazı yazdırmaya hiç ihtiyaç duyulmayabilir. Bu durumda bunu dile eklemenin ne gereği olur? Python, BASIC gibi dillerde dile gömülü halde Print fonksiyonu olsa da bunun aslında gerekli olmadığını bilmeniz gerekli.

stdio.h başlık dosyasının karakter dizileri ile ilgili fonksiyonları şunlardır

sprintf	Karakter dizisine formatlı bir şekilde yazdırır.
----------------	---

sscanf	Karakter dizisinden formatlı bir şekilde okuma yapar.
vsprintf	Değişken argüman listesinden okuduğu veriyi karakter dizisine yazdırır.
vsscanf	Karakter dizisinden okuduğu veriyi değişken argüman listesine yazdırır.
fgets	Dosya akışından karakter dizisini okur.
fgetc	Dosya akışından karakteri okur.
fputc	Dosya akışına karakter yazar.
fputs	Dosya akışına karakter dizisi yazar.
getc	Dosya akışından karakter dizisi okur
getchar	stdin'den karakter okur
gets	stdin'den karakter dizisi okur
putc	Akışa karakter yazar.
putchar	stdout'a karakter yazar.
puts	stdout'a karakter dizisi yazar

Burada karakter dizileri ile alakalı fonksiyonlar olsa da pek çoğunun giriş ve çıkış fonksiyonu olduğunu ve karakter dizilerini argüman olarak kullandıklarını

görmekteyiz. Burada önemli olan giriş ve çıkış fonksiyonlarını görsek de sizin için çok önemli olan yeni bir fonksiyondan bahsetmek istiyorum. O da `sprintf()` fonksiyonudur.

sprintf fonksiyonu

Siz bilgisayar üzerinde C programı yazarken işiniz kolaydır. Çünkü `printf` fonksiyonu ile istediğiniz değeri istediğiniz formatta kolaylıkla yazdırabilirsiniz. Elbette bu ASCII karakterleri ile sınırlı olsa da bazen sırf ASCII karakterleri kullanılarak oyun bile yapılabilir. Biz gömülü sistemler üzerinde program yazarsak bunda `printf` fonksiyonunun kullanılmadığını görürüz. Çünkü gömülü sistemlerde çıkış alacağımız bir terminal veya ekran bulunmamaktadır. Biz genellikle sisteme bağladığımız bir parçaya bu değeri karakter dizisi formatında göndermek zorunda kalırız. Örneğin bir karakter LCD ekranı buradaki konsol ekranı gibi çıkış birimi olarak kullanabiliriz. Bunun için paralel veya seri iletişim yolundan karakter dizisindeki karakterleri (baytları) sırasıyla göndermemiz gerekebilir. Bunun için programdaki bütün tam sayı, ondalıklı sayı gibi değişken tiplerini karakter dizisine çevirip sonra bu diziyi yollamamız gerekir. Aslında bizim kullandığımız `printf()` fonksiyonu da aynı görevi yapmaktadır fakat çıkışı çıkış akışına otomatik olarak yönlendirmektedir. Geri kalanı işletim sistemi üstlenmektedir. Gömülü sistemlerde genelde işletim sistemi olmadığı için çıkışı aldığımız karakter dizisini kendimiz yönlendirmek zorunda kalırız.

Örneğin PI sayısını `sprintf` ile yazdırmamız gerekirse şöyle bir komut yazarız.

```
char tampon[100];  
float pi = 3.14159F;  
sprintf( tampon, "%f", pi );
```

```
lcd_yazdir(tampon); // Özel kütüphane fonksiyonu

// Kütüphanedeki fonksiyon
void lcd_yazdir(char *string) {
    for (char *it = string; *it; it++) {
        lcd_write(*it); // DONANIMSAL KOMUTLARI ve VERİYİ YAZDIRAN
        FONKSIYON
    }
}
```

Görüldüğü gibi bazen sistemlerde bir şey yazdırmak için printf() fonksiyonu yazmamız yeterli olmuyor. Bunun neden anlattığımı soracak olursanız siz C dilini ya sistem programcılığında ya da gömülü sistemlerde kullanmak üzere öğreniyorsunuz. C dili sadece bilgisayardaki konsol ekranına bir çift yazı yazdırmak için kullanılmaz. Pek çok fonksiyonu bazen kendiniz yazmanız gereklidir. Bunun içerisine donanım sürücülerini de dahil edebiliriz.

Ayrıca bazen gömülü sistemler için C derleyicilerinde standart kütüphane fonksiyonları kısıtlanmış halde olabilir. AVR-GCC’de ben bunu ondalıklı sayı dönüşümünde yaşadım. Aslına bakarsanız yukarıda verdiğim örnek kod normal haliyle sadece soru işareti karakterini bize verecekti. Bu durumda bağlayıcı (linker) ayarlarını değiştirerek ondalıklı dönüşümünü etkinleştirmemiz gerekli.

stdlib.h başlık dosyası ve karakter dizisi işlemleri

stdlib.h dosyasında standart genel araçlar yer almaktadır. Rastgele sayı üretmek, dinamik bellek tahsisi, çevre, arama ve sıralama, tamsayı aritmetiği gibi fonksiyonların yanı sıra karakter dizisi çevirimi fonksiyonları da yer almaktadır. Bazen kullandığınız donanım ve derleyiciye bağlı olarak standart

dışı kütüphane fonksiyonları da bu tarz başlık dosyalarının içerisinde yer alabilir. Bunu derleyicinin belgelerinden okuyup öğrenmeniz gerekli. Örneğin AVR-GCC için `dtostre()` ve `dtostrf()` fonksiyonları ondalıklı sayıyı karakter dizisine dönüştürmekte. Siz standarta bağlı kalmak istiyorsanız `sprintf()` fonksiyonu ile formatlı bir şekilde dönüştürmelisiniz. `stdlib.h` başlık dosyasının karakter dizileri ile alakalı fonksiyonları şunlardır.

atof	Karakter dizisini double değerine dönüştürür.
atoi	Karakter dizisini integer değerine dönüştürür.
atol	Karakter dizisini long integer değerine dönüştürür.
atoll	Karakter dizisini long long integer değerine dönüştürür.
strtod	Karakter dizisini double değerine dönüştürür.
strof	Karakter dizisini float değerine dönüştürür.
strtol	Karakter dizisini long integer değerine dönüştürür.
strtold	Karakter dizisini long double değerine dönüştürür.
strtoll	Karakter dizisini long long integer değerine dönüştürür.
strtoul	Karakter dizisini unsigned long integer değerine dönüştürür.

strtoull	Karakter dizisini unsigned long long integer değerine dönüştürür.
-----------------	--

Bu fonksiyonları biz nerede kullanacağız diye soracak olursanız bunları kendimize ait veri transferi işlemlerini yaptığımız zaman kullanacağız cevabını veririz. sscanf() fonksiyonunu kullanmak da bir alternatif olsa da burada alınan karakter dizisi verisinin C diline göre formatlanmış olması şarttır. Biz farklı bir formatta veya formatsız bir şekilde bu veriyi alıp dönüşümü yapmamız gerekiyorsa bu fonksiyonları kullanabiliriz. Örneğin bilgisayarın seri portundan bir veri alırsak bu bize “50” şeklinde gelecektir ve bir formatı olmayacaktır. Bunu tam sayı olarak 50’ye çevirmek istiyorsak atoi fonksiyonunu kullanırız.

Bu noktaya kadar karakter dizileri hakkında anlatacağımız çok bir şey kalmadı. Bir sonraki konuda görüşmek üzere.

-43- Formatlı Giriş ve Çıkış

Formatlı giriş ve çıkış C dilinin en kolay konularından biri olmasına rağmen yeni başlayanlar için biraz kafa karıştırıcı gelebilir. Biz aslında “Merhaba Dünya!” uygulamasından beri formatlı giriş ve çıkış işlemlerini kullanmaktayız. Formatlı giriş ve çıkış hiç yabana atılacak bir özellik değildir ve C dilinin en

kullanışlı özelliklerinden biridir. Ama yeni başlayanlar bunun ayrıntısından dolayı biraz karmaşık bulmakta ve çok da fazla beğenememektedir. Bazı dillerde de bu giriş ve çıkış işlemi basitleştirilmiştir. Örneğin C++ dilinde bile `cout` ve `cin` komutları ile format girmeden giriş ve çıkış yapabilmekteyiz. Ama bir kere formatlı giriş ve çıkış işlemlerinin gücünü ve kullanışlılığını gördükten sonra onu seveceğinizi umuyoruz.

Akışlar

C dilinde bütün giriş ve çıkış işlemleri akışlar (stream) ile yapılmaktadır. Bu akışlar bir sıralanmış baytlardan oluşmakta ve fonksiyona göre yönlendirilmesi yapılmaktadır. Örneğin girişte klavyeden girilen karakterlerin sırası, dosyadan okunan bayt sırası veya ağ bağlantısından gelen baytlar akışlara örnektir. Program çalışmaya başladığında üç adet akış otomatik olarak programa bağlanmaktadır. Bunlardan biri standart giriş akışı olup klavyeye bağlıdır (*stdin*), diğeri standart çıkış akışı olup monitöre (konsol ekranına) bağlıdır (*stdout*) ve sonuncusu ise standart hata akışı olup hata mesajlarını içermektedir (*stderr*). İşletim sistemi bu akışları başka aygıtlara yönlendirme yeteneğine sahiptir.

printf fonksiyonu

Baştan beri gördüğümüz `printf()` fonksiyonu formatlı çıkış elde etmemizi sağlayan fonksiyondur. Yukarıda belirtildiği üzere standart çıkışa (*stdout*) bağlıdır ve format belirteçleri ile aldığı argümanları karakter dizisi olarak yazdırmaktadır. `printf()` fonksiyonunun iç kısmına baktığımızda oldukça karmaşık bir format çözümleme ile en sonunda işletim sistemine karakter dizilerinin gönderildiğini görürüz. `printf()` fonksiyonu formatlı çıkış almak için tek fonksiyon değildir. `sprintf()` ve `fprintf()` gibi aynı işleve sahip fakat farklı

noktalarda çıkış aldığımız fonksiyonlar da mevcuttur. `sprintf()` önceden söylediğimiz gibi karakter dizisine aynı `printf()` fonksiyonu gibi formatlı bir çıkış vermekte, `fprintf()` ise ileride göreceğimiz dosyalara formatlı bir çıkış vermektedir.

Şimdiye kadar `printf()` fonksiyonu için farklı format belirteçlerini kullandık. Bunları da değişken tiplerini verdiğimiz sıra tablo içerisinde size vermiştik. Örneğin tamsayı yazdıracağınız zaman `%i`, float değer yazdıracağınız zaman `%f` (aslında burada float double'a çevirilir.), işaretçi yazdıracağınız zaman `%p`, karakter dizisi yazdıracağınız zaman da `%s` gibi belirteçleri kullanmaktaydık. Bazen `printf()` fonksiyonu bu format belirteçleri ile tip dönüşümü de yapabilmektedir.

Fakat `printf()` fonksiyonu bu kadar basit bir fonksiyon olmakla sınırlı kalmamaktadır. Belirteçlerin yanında bayraklar, genişlik, ondalık hane gibi özellikler de yer almaktadır. Aynı zamanda kaçış sekansları adı verdiğimiz (escape sequence) karakterlerle özel komutlar işletilmektedir. Bu sayede örneğin ondalıklı sayıları yuvarlatabilir, çıkışı sağa veya sola hizalayabilir veya ondalıklı değerleri bilimsel formatta gösterebiliriz.

Öncelikle `printf()` fonksiyonunun prototipini sizlere gösterelim.

```
int printf ( const char * format, ... );
```

Biz `printf()` fonksiyonundan dönen değeri şimdiye kadar okumasak da aslında `printf()` fonksiyonunun `int` tipinde bir sayı değerini geri döndürdüğünü görmekteyiz. Bu geri dönen değer yazdırma başarılı ise kaç karakterin yazdırıldığına dair bir pozitif değer olmaktadır. Eğer yazdırma başarısız ise negatif değer dönmektedir. Bu değer hata denetleme için kullanılabilir.

Burada *const char * format* tipini hepimiz biliyoruz. Bu bir karakter dizisi. `printf()` fonksiyonunu kullanırken ilk argümanın her zaman bir karakter dizisi sabiti olarak çift tırnak arasında yazıldığını gördük. “Merhaba %s” gibi yazılan değer hiç bir zaman değişmiyor. O yüzden *const char *format* olarak değer almakta. Burada asıl mesele ikinci argümandır. Gördüğünüz gibi orada herhangi bir değer yazmamakda ve sadece üç nokta (...) yer almakta. Bu aldığı argümanların herhangi bir sınırı olmadığını göstermektedir. Değişken argümanlar (variable arguments) adı verilen bu özellik C dilinde mevcuttur ve biz de fonksiyonları bu şekilde yapabiliriz. İlerleyen konularda anlatacağımız için şimdiden kafanızı karıştırmak istemiyoruz.

Şimdi `printf()` fonksiyonun aldığı format belirteçlerini bir tablo halinde size verelim.

Belirteç	Çıkış	Örnek
d ya da i	İşaretili tam sayı (signed int)	300
u	İşaretsiz tam sayı (unsigned int)	5000
o	İşaretsiz oktal	640
x	İşaretsiz onaltılık tamsayı (küçük)	ff0
X	İşaretsiz onaltılık tamsayı(büyük)	FF0
f	Onluk ondalıklı sayı (float, küçük)	320.15

F	Onluk ondalıklı sayı (float, büyük)	300.25
e	Bilimsel notasyon, küçük	3.12e+2
E	Bilimsel notasyon, büyük	3.12E+2
g	%e ya da %f'nin en kısa temsili	3.14
G	%E ya da %F'nin en kısa temsili	3.14
a	Onaltılık ondalıklı sayı, küçük	-0xc.90FEA-2
A	Onaltılık ondalıklı sayı, büyük	-0XC.90FEA-2
c	Karakter	c
s	Karakter dizisi	Arduinocu
p	İşaretçi Adresi	0FFA005B
n	Birşey yazdırılmaz.	
%	Peşine gelen % işareti ekranda yazdırılır.	%%

Burada kadar anlamadığımız bir nokta yok. Zaten derslerin başından veri bunları kullanıyoruz. Kullanmadığımız bazı format belirteçleri yer alsa da bunları siz rahatlıkla ihtiyaç duyduğunuz zaman kullanabilirsiniz.

Formatlamada belli bir söz dizimi takip etmek zorunda olduğunuzu unutmayın. Eğer sıraya uygun formatı yazmazsanız doğru formatlanmayacaktır. Bunun söz dizimi şu şekildedir.

%[bayraklar][genişlik][.hane][uzunluk]belirteç

Burada % işaretini ve belirteçleri biliyoruz fakat karşımıza bayraklar, genişlik, hane ve uzunluk gibi değerler çıkmakta. Şimdiye kadar bunlardan size bahsetmedik ve bunları hiç kullanmadık. Artık sıra printf() fonksiyonunu tam anlamıyla öğrenmeye geldi.

Bayraklar

Bayraklar formatlı giriş ve çıkışta önemli bir yere sahiptir. Verinin nasıl hizalanacağı, nasıl işaretleneceği, nasıl belitileceği gibi konuları bayraklar vasıtasıyla belirtiriz. Bayrakları şu şekilde özetleyebiliriz

Bayrak	Açıklama
-	Belirtilen alan genişliği içerisinde veriyi sola hizalama. Normalde sağa hizalanmaktadır.
+	Yazdırılan verinin başına pozitifse (+) negatifse (-) işaretini koymaya zorlar. Normalde sadece negatif sayılarda (-) işaretini görürüz.
(boşluk)	Eğer herhangi bir işaret yazılmazsa değerden önce boş bir karakter eklenir.
#	o, x ve X belirteçleri ile kullanılır. Bu oktal ve onaltılık sayıları belirtmek için önek koyar. Örneğin hex sayı için 0xFF gibi. Ondalıklı sayılarda ise düz sayı olsa bile noktasını belirtir. Örneğin 32.00 gibi.

0	Sol taraftaki boşluklar sıfır ile doldurulur boşluk yerine.
---	--

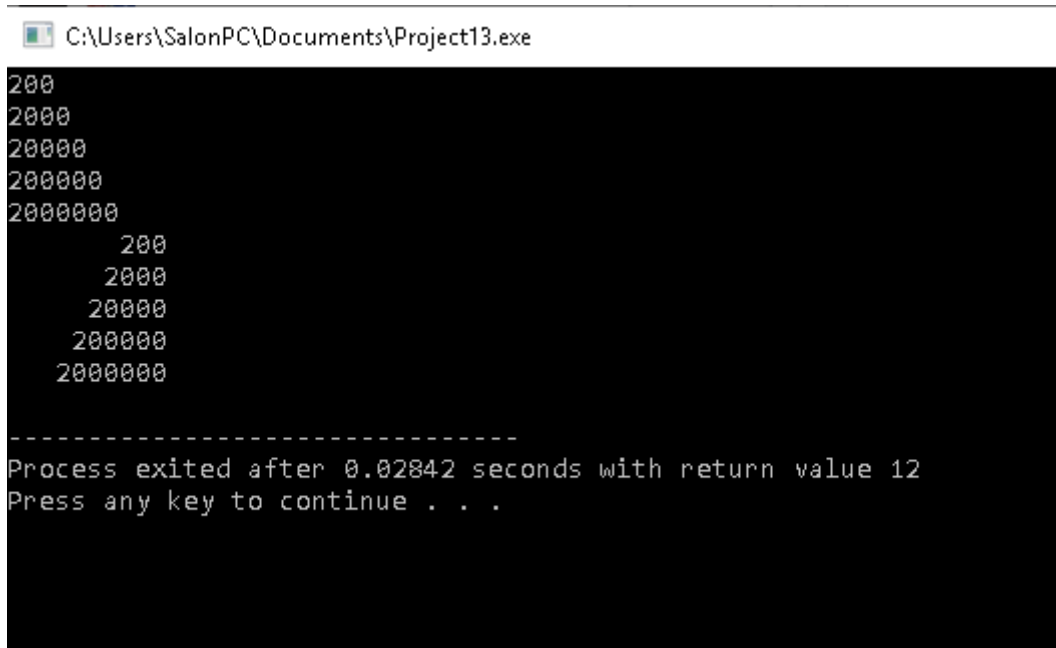
Şimdi “-” bayrağı ile ilgili bir uygulama yapalım. “-” bayrağının düzgün çalışması için öncelikle bir boşluk yani hücre tanımlamamız gereklidir. Aynı tablodaki hücrelerde olduğu gibi printf() ile tanımladığımız hücrelerde de veriler sola ya da sağa hizalı yazdırılabilir. Burada hücrenin uzunluğu karakter sayısı olarak ele alınmaktadır. Sonrasında ise sola hizalamak için (-) yazıyoruz ve sağa hizalamak için ise boş bırakıyoruz. Örnek programımız şu şekildedir.

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

main()
{
    int sayi1 = 200;
    int sayi2 = 2000;
    int sayi3 = 20000;
    int sayi4 = 200000;
    int sayi5 = 2000000;
    printf ("% -10i \n", sayi1);
    printf ("% -10i \n", sayi2);
    printf ("% -10i \n", sayi3);
    printf ("% -10i \n", sayi4);
    printf ("% -10i \n", sayi5);
    printf ("%10i \n", sayi1);
    printf ("%10i \n", sayi2);
    printf ("%10i \n", sayi3);
```

```
printf ("%10i \n", sayi4);  
printf ("%10i \n", sayi5);  
}
```

Burada güzel gösterebilmek adına beş adet farklı basamaklarda sayı değişkeni tanımladık ve bunları ilk önce %-10i formatında gösterdik. Bu bilgisayara “Bana 10 karakterlik bir hücre ayarla ve sola hizalı olarak tam sayı yazdır.” demenin kısa yoludur. Burada 10 yazarak değerin yazdırılacağı boşluğun genişliğini belirlemiş olduk. Sonrasında ise yine normal bir şekilde değerleri yazdırdık. Bu sefer sola hizalı değil de sağa hizalı olarak yazdırıldı. Önemli nokta burada belli bir genişlik değerinin verilmesidir. Genişlik değeri olmadan sola ya da sağa hizalamanın bir anlamı kalmayacaktır. Programın ekran görüntüsü şu şekildedir.




```
C:\Users\SalonPC\Documents\Project13.exe  
200  
 200  
  200  
   200  
    200  
     200  
-----  
Process exited after 0.02842 seconds with return value 12  
Press any key to continue . . .
```

Şimdi “+” bayrağını kullanmak için ufak bir program yazalım ve nasıl çalıştığını görelim.

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

```
main()
{
    int sayi1 = 200;
    int sayi2 = -300;
    printf("Sayi1 = %+i \nSayi2 = %+i", sayi1, sayi2);
}
```

Bu programı çalıştırdığımızda ekran görüntüsü şu şekilde olacaktır.

 C:\Users\SalonPC\Documents\Project13.exe

```
Sayi1 = +200
Sayi2 = -300
-----
Process exited after 0.02269 seconds with return value 26
Press any key to continue . . .
```

Anlatmaya gerek yok görüyorsunuz. Bu bayrak sadece pozitif ve negatif sayıları daha belirgin gösterebilmek adına pozitif sayıların önüne + işareti koymakta. Bir gün gelir de bu şekilde yazdırmanız gerekirse böyle bir özelliğin olduğunu bilin.

Genişlik kısmını (width) yukarıda gösterdiğim için tekrar tekrar göstermeyeceğim. Yukarıda bazı bayraklar için kendiniz örnek yapıp deneyebilirsiniz. Bunlar çok sık kullanılan özellikler olmadığı için mümkün mertebe önemsiz örnekler vermemeye çalışıyorum. Burada aslında en önemli noktalardan birisi ondalıklı sayıların nasıl gösterileceğidir. Örneğin biz PI sayısını 3.14 olarak da gösterebiliriz 3.14159265359 olarak da gösterebiliriz. Bu durumda bazen istemediğimiz haneler hesaba katılabilir. Bunun için

yuvarlama işlemi yapılmaktadır. Ondalık sayılarda noktadan sonra kaç hane gösterileceğine precision adı verilmektedir. Bu terim hassasiyet ve doğruluk için de elektronikte kullanılmaktadır.

Ondalık sayılar için şu şekilde bir format belirteci kullanılır;

%.sayif

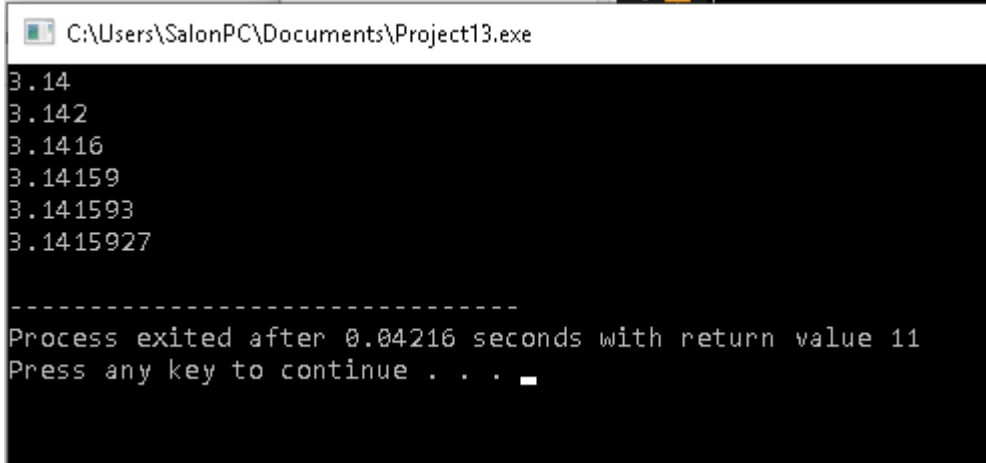
Yani biz `%.2f` yazdığımızda virgülden sonraki iki hane gösterilir, `%.5f` yaptığımızda ise virgülden sonraki beş hane gösterilir. Geri kalan sayılar ise **yuvarlatılır**. Bu yuvarlama işleminden sonra 3.14... diye devam eden bir sayı 3.15 olarak yazdırılabilir ve böyle işlem yapılabilir. Hassas işlemlerde buna dikkat etmeniz çok önemlidir. 5'i 2'ye bölerken tam sayı kullanmamanız gerektiğini şimdiye kadar öğrenmiş olmalısınız. C dilinde ondalıklı sayılarla ilgili ikinci en önemli tüyo da budur. Şimdi yuvarlama işleminin nasıl yapıldığını görmek için bir örnek yazalım.

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

main()
{
    float pi = 3.14159265;
    printf("%.2f \n", pi);
    printf("%.3f \n", pi);
    printf("%.4f \n", pi);
    printf("%.5f \n", pi);
    printf("%.6f \n", pi);
    printf("%.7f \n", pi);
}
```


}

Program çalıştığında ekran görüntüsü şöyle olacaktır.



```
C:\Users\SalonPC\Documents\Project13.exe
3.14
3.142
3.1416
3.14159
3.141593
3.1415927
-----
Process exited after 0.04216 seconds with return value 11
Press any key to continue . . . _
```

Görüldüğü gibi burada bir yerde 3.142 derken bir yerde 3.1416 demektedir. O da aslında 3.1415 olması gerekirken yuvarlama sonucunda bu değeri almıştır. Buna dikkat ettiğiniz sürece ondalıklı sayıların gereksiz basamaklarından kurtulmak için müthiş bir özellik olduğunu söyleyebiliriz.

Kaçış Sekansları

Kaçış sekansları printf() fonksiyonunun karakter dizisi sabiti içerisinde yer alan ve fonksiyon içerisinde bazı karakterleri yazdırmak için veya bazı komutları işletmek için kullanılan özel karakterlerdir. Bunlardan yeni satıra atlamak için “\n” sekansını sıkça kullandığımızı biliyorsunuz. Bunları da lazım olunca kullanacağınız için sadece tablosunu vermekle yetinelim.

Kaçış Sekansı	Açıklama
\'	Tek tırnak işareti (') yazdırır.
\"	Çift tırnak işareti (") yazdırır.

\?	Soru işareti (?) yazdırır.
\\	Backslash işareti (\) yazdırır.
\a	Bilgisayardan uyarı sesi gelir. (Bende Windows'dan uyarı sesi geliyor. Eski sistemlerde PC hoparlöründen bip sinyali alınabilir veya sisteme göre değişir.)
\b	İmleci bir karakter geriye götürür.
\f	İmleci bir sonraki mantıksal sayfanın başına götürür
\n	İmleci yeni satıra götürür.
\r	İmleci mevcut satırın başına götürür.
\t	Yatay satır atlama (tab)
\v	Dikey satır atlama

Artık C formatlarına dair anlatacağımız bir konu kalmadı. Artık C dilinin çoğu konusunu bitirdiğimizi söyleyelim ve bir sonraki konumuza geçelim.

-44- Yapılar

Şimdiye kadar anlattığımız C konuları içinde veri yapıları olarak değişkenleri ve dizileri gördünüz. Değişkenler bağımsız ve bireysel veri tipleri iken diziler ise bu değişkenlerin ipe boncuğun dizilmesi gibi birbiri ardınca sıralanmasından oluşuyordu. En önemli nokta ise bir dizide bütün elemanlar aynı tip olmak zorundaydı. Örneğin hem karakter hem de tam sayı içeren bir dizi tanımlamanızın imkanı yoktu. Üstelik bu çok boyutlu diziler için de geçerliydi. Örneğin bir veri tipinde biz bir öğrencinin adını, okul numarasını ve notunu tutmak isteyebiliriz. Bunun için ad değeri karakter dizisi, numara tam sayı ve not da tam sayı olmak zorundadır. Bunu dizilerle yapmamız mümkün değildir. O halde daha karmaşık bir veri yapısına ihtiyacımız vardır. Bu veri yapısı içerisinde farklı tipte değişkenleri barındıracak ve bir arada bulunduracak bir veri yapısı olmalıdır.

Yapılar

İşte burada yapılar devreye girmektedir. Yapılar aslında nesne tabanlı programlamanın da temelini oluşturmaktadır. Çünkü yapılarla aslında bir fonksiyonsuz, işlevsiz ve sadece özelliklere sahip bir nesne tanımlarız. Bu veri yapısı bir bakımdan veri tabanlarındaki kayıtlara benzemektedir. Örneğin bir karton kutu yapısında en, boy ve yükseklik olmak üzere üç ayrı özelliğe ihtiyacımız vardır. Bu durumda int en, boy, yukseklik olarak üç ayrı değişkeni tanımlayabiliriz fakat normal değişken tanımlama ile bu değişkenler arasında bağ kuramayız. Ayrıca aynı tip özelliklere sahip fakat farklı üyeler tanımlamak istediğimizde bunlar için birbiriyle alakasız boy2, en2, yukseklik2 ve boy3, en3, yukseklik3 diye devam eden değişkenler tanımlamamız gerekir. Bu da

programı işin içinden çıkılmaz hale sokacaktır. Yapılar bütün bu iki problemi de çözmekte ve daha karmaşık veri yapıları üzerinde çalışmamıza imkan tanımaktadır. Şimdi yapıların ne olduğuna derinlemesine göz atalım.

Yapılar birbiriyle **alakalı** değişkenlerin bir isim altında toplanmasıdır. Bu birbiriyle alakalı değişkenler dizilerde de olsa da diziler ancak tek bir tipte değer içerebilir ve karmaşık bir veri depolamaya imkan tanımaz. Yapılar ise farklı tipte değişkenleri bir arada barındırdığı gibi dizilerdeki ipe dizilen boncuklar gibi olan alaka yerine burada aynı kutuya konan parçalar gibi bir alaka sağlanmış olur. Umarım benzetmeden meseleyi anlamışsınızdır. Yapılar birbiriyle ilişkili değerleri bir çatıda topladığı gibi birden çok yapı da daha büyük bir yapı çatısı altında toplanabilir. Bu veri hiyerarşisi dizilerdeki çizgisel ilişkiden çok daha karmaşıktır.

Şimdi size bir yapının nasıl olabileceğini resmedelim. Öncelikle dizi elemanı oluşturalım ve buna değerleri verelim.

dizi[0]	dizi[1]	dizi[2]	dizi[3]	dizi[4]	dizi[5]	dizi[6]
50	233	458	66	441	665	236
0x00	0x01	0x02	0x03	0x04	0x05	0x06

Burada varsaydığımız bir örnek dizide elemanların birbiriyle ilişkisi dizi erişim operatörünün değeri ve sıralamasından ibarettir. Bu diziler nicelik olarak çok olabilse de tek bir niteliği ifade etmektedir. Şimdi örnek bir yapının tablosuna bakalım

kutu1		
en	boy	yukseklık
100	50	45

Gördüğünüz gibi kutu1 adlı yapıda en, boy ve yukseklik adlarında üç ayrı değer yer almakta ve her birinin hücresi birbirinden farklı olmakta. Burada dizi elemanları gibi sıra olmayıp farklı adlar da yer alabilmektedir. Ama hepsinin kutu1 çatısı içerisinde olduğunu unutmayın. Bu değişkenlere ancak kutu1 yapısına erişmekle erişme imkanımız olur. Aynı fonksiyonların içinde yer alan değerler gibi buradaki değişkenler kutu1 yapısının dışında tek başlarına erişim imkanı olmadığı gibi bir anlam da ifade etmemektedirler. Şimdi yukarıda tablosunu çizdiğimiz yapıyı C dili ile tanımlayalım.

```
struct kutu
{
    int en;
    int boy;
    int yukseklik
}; // Noktalı virgölü unutma
```

Biz böyle diyerek sadece bir adet yapı değişkeni tanımlamadık. Bir yapı tipi tanımladık. Aynı int, char, float gibi birbirinden bağımsız değişken tipleri gibi bu yapı tipiyle de istediğimiz kadar birbirinden farklı yapı değişkeni tanımlayabiliriz. Burada bir nevi kendi değişken tipimizi oluşturduk diyebiliriz. Yalnız bu değişken tipi mevcut değişkenlerin bir araya getirilmesiyle daha karmaşık bir değişken olarak oluşturuldu. Şimdi kutu1 adında bir yapı değişkeni tanımlayabiliriz.

```
struct kutu kutu1;
```

Burada kutu adında bir yapı tipi tanımlasak da bunu değişken tanımlar gibi kutu kutu1; şeklinde tanımlama imkanımız burada yoktur. İleride tip tanımlayıcıları (typedef) konusuna geldiğimizde yapıların sadece adları ile de tanımlanabildiğini göreceğiz. İster struct kutu kutu1; diyin isterseniz de tip tanımlayıcıları ile kutu kutu1; yapının sonuç hep aynı olacaktır. Sadece biri daha kolaylaştırılmış şeklidir ve C++ dilindeki sınıf tanımlamalarına daha çok benzemektedir.

Yapılara değer atamak ve yapı erişim operatörü

C dilinde yapılar için kullanılan iki özel operatör bulunmaktadır. Biri ok operatörü olan (->) operatörüdür. Bu operatör yapı değerinin adresi (işaretçi) üzerinden değere erişme operatörü olup anlaması biraz karışıktır. Standart olan operatör ise nokta operatörü (.) olup doğrudan yapı değerine ulaşmaktadır. Şimdi bu operatörün nasıl kullanılıp değer atandığını size gösterelim.

```
struct kutu kutu1;  
kutu1.en = 50;  
kutu1.boy = 100;  
kutu1.yukseklik = 150;
```

Gayet basit görünüyor değil mi. Biz struct kutu kutu1 diyerek kutu tipinde bir yapı değeri tanımladık. Kutu tipindeki yapıya baktığımızda içinde en, boy ve yükseklik olmak üzere üç adet int tipinde değer olduğunu görürüz. O halde kutu1.en dediğimizde kutu1 yapısının içindeki en değişkenine erişmiş oluruz ve = operatörü ile 50 değerini atarız. Eğer eşittirin sağ tarafına koysaydık yapı üyesinin içindeki değeri de getirme imkanımız olurdu. Bunlar üzerinde aritmetik işlem de yapabiliriz. Nokta operatörünü koyduktan sonra aynı değişkenler gibi çalışmaktadırlar. Bu değerlere erişme aynı değişken

kapsamlarında olduğu gibi sınırlandırılmaktadır. Yani bir fonksiyonun içinde yapı tanımladıysak başka bir fonksiyonun içinde buna erişemeyiz. Ama yapı değişkenlerinin de aynı değişkenler gibi fonksiyonlara argüman olarak gönderilebileceğini unutmayalım.

Yukarıda tanımladığımız kutu1 adlı yapı değişkenine değerlerini üç ayrı satır kod yazarak verdik. Bunu her zaman böyle yazmak pek pratik olmayabilir. O yüzden dizilere değer atar gibi süslü parantezler içerisine de bu değerleri yazabiliriz. Böylelikle ilk tanımlamada değerlerini vermiş oluruz.

```
struct kutu kutu1 = { 50, 100, 150 };
```

Burada 50, 100, 150 diye yazdığımız değerler **sırayla** yapı değişkeninin en, boy ve yükseklik elemanlarına aktarılmaktadır. Burada yapı prototipini göz önünde bulundurmak ve yapı tanımlarken bunlara uygun değerleri atamak çok önemli. Elimizin altında basit bir değişken yok bir yapı var. Bu durumda daha dikkatli olmalıyız.

Yapı tanımlama ile alakalı bir özellik de yapı tipini tanımladığımız yerde yapı değişkenlerini tanımlama imkanımızdır. Böylelikle kullanılacak değişkenler derli toplu halde olmaktadır. Örneğin yukarıdaki kutu tipinde yapıda üç adet yapı değişkeni tanımlamak istersek şu şekilde yazabiliriz.

```
struct kutu
{
    int en;
    int boy;
    int yukseklik
} kutu1, kutu2, kutu3;
```

Ayrıca yapılar dizi halinde de tanımlanabilir. Hem yapı hem de dizi olacak bir veri tipi size kafa karıştırıcı görünmüş olabilir. Ama daha veri yapılarına geçeceğiz! Bunlar size çocuk oyuncağı gibi gelmeli.

```
struct kutu kutular[20];
```

Burada kutu tipinde ve kutular adında toplam yirmi adet yapı değişkeni oluşturduk. Bu yapı dizisindeki üyelerin elemanlarına erişmek aynı yapıların elemanlarına erişmek gibidir. Örneğin kutular dizisindeki 5 numaralı elemanın boy değerine ulaşalım.

```
deger = kutular[5].boy;
```

Değişkenlerin işaretçilerinin olduğunu biliyoruz. Ama aynı zamanda her yapı tipinin de bir işaretçisi bulunur ve yapı tipini işaret eder. Yukarıda bahsettiğimiz ok operatörünü (->) işte bu durumlarda kullanırız. Şimdi bir kutu yapısının işaretçisini tanımlayalım.

```
struct kutu *kutuptr;
```

Ayrıca bu işaretçiye değer atamak için addressof(&) operatörünü yapıların adresine erişmek için kullanabiliriz. Örnek bir kullanım şu şekilde olabilir.

```
kutuptr = &kutu1;
```

Bu durumda kutu1'i gösteren kutuptr yapı işaretçisinden kutu1'in örneğin en değerine erişmek istiyorsak şu şekilde bir kod yazarız.

```
deger = kutuptr->boy;
```

Yukarıda yazdığımız kod aslında **(*kutuptr).boy** komutuna eşdeğerdi. Yani * operatörüyle kutuptr'nin işaret ettiği yapı değişkeninin kendisine ulaşıyoruz sonra ise (.) operatörüyle elemanının değerine ulaşıyoruz. Ama genellikle böyle bir kullanım yerine ok operatörü (->) kullanılmaktadır. Gömülü sistemlerde çalışırken bunu örneğin STM32'nin HAL (Donanım soyutlama

katmanı) kütüphanesinde bolca görebilirsiniz. Ne yazık ki pek çok kaynak ve kitap ok operatörünü es geçmektedir. Biz temel C programlama derslerinde bile bundan bahsetme ihtiyacı duyuyoruz.

Yapılarla ilgili bir diğer özellik de aynı tipteki iki yapı değişkeni birbirine atanabilir. Yani istersek kutu1 = kutu2 şeklinde bir kullanımda bulunabiliriz.

Yapılar hakkında anlatacaklarımız bu kadardı. Bir sonraki yazıda yapılar ile alakalı uygulama yapıp, argüman olarak yapıların nasıl aktarıldığını göstereceğiz. Ayrıca yapılar ile doğrudan alakalı olan typedef tip tanımlamasından bahsedeceğiz.

-45- Yapı Uygulamaları ,typedef ve union (birlikler)

Daha önceki başlıkta yapılara giriş yapmış ve yapılarla ilgili bilmeniz gerekenleri size aktarmıştık. Burada öncelikle yapılarla basit uygulamalar yapmak istiyorum sonrasında ise typedef tip tanımlayıcısını sizlere açıklayacağım. Şimdi yapı uygulamalarına geçelim.

Basit bir yapı uygulaması

Daha önce bahsettiğimiz gibi bir kutu yapısı oluşturalım ve bu yapı değerleri üzerinde biraz oynama yaparak bunu ekranda gösterelim. Uygulamamız gayet basit olacaktır.

```
#include <stdio.h>
#include <string.h>
struct kutu
{
    int en;
    int boy;
    int yukseklik;
};
int main()
{
    struct kutu kutu1 = { 10, 15, 30 };
    printf("kutu1 En: %i \n", kutu1.en);
    printf("kutu1 Boy: %i \n", kutu1.boy);
    printf("kutu1 Yukseklik: %i \n", kutu1.yukseklik);

    kutu1.boy = 20;
    kutu1.en = kutu1.en * 2;
    kutu1.yukseklik = 100;
```

```
printf("Veriler Guncellendi \n \n");  
printf("kutu1 En: %i \n", kutu1.en);  
printf("kutu1 Boy: %i \n", kutu1.boy);  
printf("kutu1 Yukseklik: %i \n", kutu1.yukseklik);  
}
```

Burada öncelikle kutu adında bir yapının tanımlaması yapılmaktadır. Daha önce anlattığımız yapının aynısı olarak en, boy ve yukseklik olmak üzere üç adet int tipinde değişken içermektedir. Şimdi main fonksiyonu içerisinde komutların nasıl işletildiklerine göz atalım.

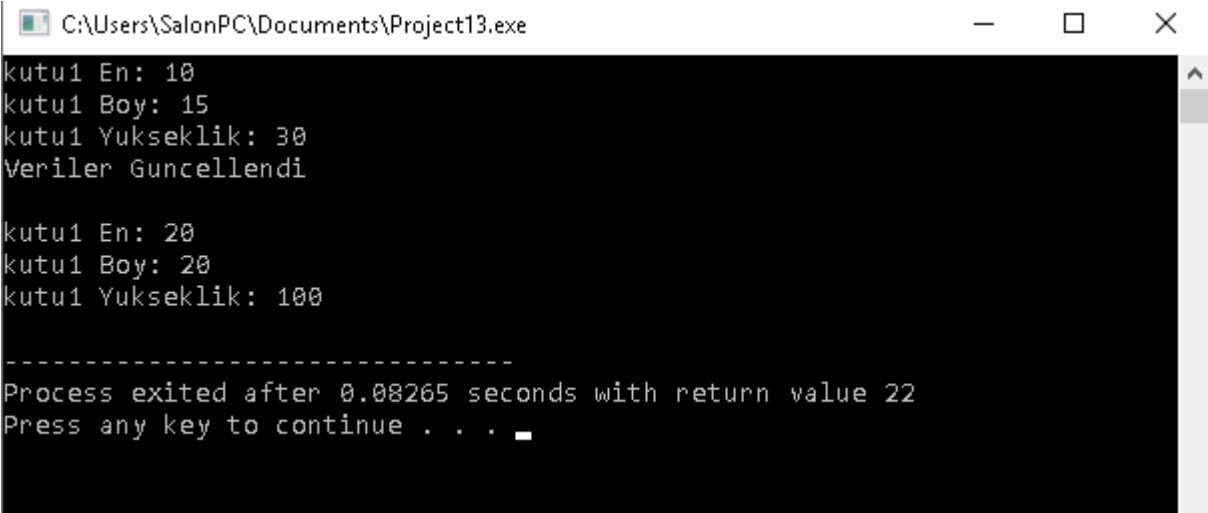
struct kutu kutu1 = { 10, 15, 30 }; Burada kutu tipinde ve kutu1 adında bir yapı değişkeni tanımladık ve ilk değerini sırayla 10, 15 ve 30 olarak verdik. Hangi değer hangi yapı elemanına denk geldiğini yukarıdaki yapı prototipinden anlıyoruz. Burada değerler en, boy, yukseklik diye sıra ile verilmektedir.

printf("kutu1 En: %i \n", kutu1.en); Burada printf() fonksiyonuna argüman olarak kutu1.en ifadesini yazdık. Bildiğiniz üzere yukarıda bunu int en; diye tanımladık. Yani elimizde bir tam sayı var bunun için %i belirtecini kullandık. Dikkat edin burada argüman olarak yapı değeri değil kutu1 yapısının içindeki int tipindeki değişken gidiyor.

kutu1.boy = 20; kutu1 yapı değişkenini tanımlarken buna 10 değerini atamıştık fakat programın herhangi bir yerinde yapı erişim operatörü ile yapı elemanına erişip değerini değiştirebiliriz.

kutu1.en = kutu1.en * 2; Gördüğünüz gibi yapı elemanları üzerinde aritmetik işlemler yapabiliriz ve bunu tekrar yapı elemanlarına atayabiliriz. Yapılar üzerinde çalışmadığımıza dikkat edin.

Programımız çalıştığında ekran görüntüsü şu şekilde olacaktır.



```
C:\Users\SalonPC\Documents\Project13.exe
kutu1 En: 10
kutu1 Boy: 15
kutu1 Yukseklik: 30
Veriler Guncellendi

kutu1 En: 20
kutu1 Boy: 20
kutu1 Yukseklik: 100

-----
Process exited after 0.08265 seconds with return value 22
Press any key to continue . . .
```

Fonksiyona argüman olarak yapıları aktarmak

Yapıların en güzel yanlarından biri de fonksiyonlara aynı değişkenler gibi argüman olarak aktarılabilmesidir. Üstelik bu argüman olarak aktarma işi oldukça kolay yapılmaktadır. Aynı değişken adı yazar gibi yapı adı yazarak argüman aktarabiliriz. Şimdi yukarıda yazdığımız programı fonksiyonlar vasıtasıyla yapalım.

```
#include <stdio.h>
#include <string.h>
struct kutu
{
    int en;
    int boy;
    int yukseklik;
```

```

    };

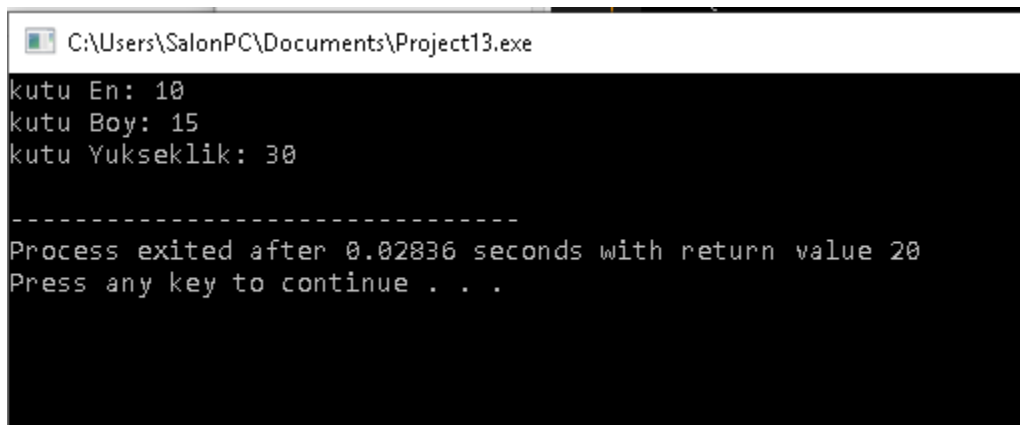
void yazdir (struct kutu kutu);

int main()
{
    struct kutu kutu1 = { 10, 15, 30 };
    yazdir(kutu1);
}

void yazdir (struct kutu kutu)
{
    printf("kutu En: %i \n", kutu.en);
    printf("kutu Boy: %i \n", kutu.boy);
    printf("kutu Yukseklik: %i \n", kutu.yukseklik);
}

```

Programı çalıştırdığımızda ekran görüntüsü şu şekilde olacaktır.



```

C:\Users\SalonPC\Documents\Project13.exe
kutu En: 10
kutu Boy: 15
kutu Yukseklik: 30
-----
Process exited after 0.02836 seconds with return value 20
Press any key to continue . . .

```

Burada yapı değerini argüman olarak değeri ile gönderdik. Yani bu fonksiyon orjinal yapı değişkeninin değerlerini kopyalar ve kullanır fakat onun üzerinde değişiklik yapamaz. Eğer yapı dizisi şeklinde veya işaretçi olarak

gönderseydik aynı değişkenlerde olduğu gibi referans olarak aktarmış olacak ve değişiklik yapabilecektik.

typedef

Şimdi C ile yazılmış gelişmiş kütüphanelerde gördüğümüz tip tanımlama özelliğine bir göz atalım. Tip tanımlama en yalın anlatımıyla kendi değer tipimizi oluşturma anlamına gelir. Örneğin normalde int, float, char gibi değişken tipleri varken biz uzunluk, led, dugme gibi değişken tipleri oluşturabilir ve bunlarla tanımlama yapabiliriz. Aynı zamanda tip tanımlamayı yapı tiplerinde de kullanabiliriz. Tip tanımlama (typedef) en sık yapı tiplerinde kullanılmaktadır. Tip tanımlamanın ktek bir kullanım sebebi vardır o da kolaylaştırmadır. Şimdi biz yukarıda kutu tipinde bir yapı değişkeni tanımlarken struct kutu kutu1; gibi bir kod yazıyorduk. Yani her tanımlamada struct yazmak zorundayız. Bunu ortadan kaldırmak ve bağımsız bir tip tanımlamak için şöyle bir yol izleriz.

```
typedef struct {  
    int en;  
    int boy;  
    int yukseklik;  
} kutu;
```

Artık kutu adında bir yapı tipi tanımladığımıza göre bu tipte bir değişken tanımlayabiliriz. Onun için şöyle bir kod yazmamız gerekir.

```
kutu kutu1;
```

İstersek typedef özelliğini değişken adlarında da kullanabiliriz. Normalde 8-bitlik veri tipine C dilinde char adı verilir. Ama veri olarak bunun bayt verisi

olduğunu biliyoruz. O halde kendimize byte adında bir değişken tipi tanımlayalım.

```
typedef unsigned char byte;
```

```
byte deger1, deger2;
```

Görüldüğü gibi oldukça basit bir kullanıma sahip. Şimdi union yani birlikler konusundan kısaca bahsedelim ve yapı konusunu bitirelim.

union (birlikler)

union tipi aynı yapı tipine benzese de burada sadece tek bir değişken tanımlanabilmekte ve farklı değişkenler sadece tek bir yapı değişkenini işaret etmektedir. Yani union tipinde bir ana veri tipi tanımlanır ve bunlara farklı değişkenler vasıtasıyla erişilir. Bellekten tasarruf etmek amacıyla kullanılabilir. Birbiriyle alakalı değişkenlerin bir araya getirilmesi gibi düşünebiliriz. O yüzden adı union yani birlik demektir.

```
union kutu {  
    int en;  
    int boy;  
    int yukseklik;  
};
```

```
union kutu kutu1, kutu2, kutu3; // üçü de aynı kutu değişkeni  
demek.
```

Bir sonraki konuda C dili eğitimlerinde biraz göz ardı edilen ama gömülü sistemlerin olmazsa olmazı olan bit bazlı işlemleri size anlatacağım. Bit bazlı

işlemler aynı işaretçiler gibi alt seviye programlamanın olmazsa olmaz parçalarından biridir. Bunun önemini bir sonraki makalede göreceksiniz.

-46- Bit Manipülasyonu ve Bitwise Operatörleri

Programcılığa yeni başlayan biri olsaydım C ile alakalı bir kitabı açıp okuduğumda bit bazlı işlemler ve operatörler kısmını gereksiz bir ayrıntı olarak görüp çok üzerinde durmazdım. Aslında üst seviye programlama ile uğraşanlar için de bir ayrıntıdan öte gitmemektedir. Fakat gömülü sistemler

üzerinde çalışan biri olarak şimdi C dilinin en önemli konularından biri olarak görmekteyim.

Pek çoğunuz C dilini gömülü sistemlerde çalışmak için öğreniyorsunuz. Bu zamanda C dili ne Web programcılığında ne mobil programcılıkta kullanılıyor. Masaüstü programcılıkta ise aslında bakarsanız pek pratik bir kullanım alanına sahip değil. Fakat bütün bu alanlarda irdelenen bir dil gömülü sistemlerde bir numaralı dil olmayı başarabiliyor. Zamanında C dili bilgisayarlarda kullanılan bir numaralı dildi (90'ların başı) fakat günümüzde neredeyse hiç kullanılmamakta. Yani günümüzde C dilinin ait olduğu yer sistem programcılığı ve gömülü sistemlerdir.

C dilinin yaygın olarak uygulandığı gömülü sistemlere baktığımızda ise bit bazlı işlemlerin ve işaretçilerin aşırı derecede fazla kullanıldığını görüyoruz. Yani C dilinin alt seviye noktaları gömülü sistemlerde daha ön plana çıkmakta. Hatta gömülü sistemlerde “Merhaba Dünya!” anlamına gelen led yakma uygulamasında bile bit bazlı işlemler kullanılmakta. Yani bit bazlı işlemleri yapmadan bazen en basit programı bile yazmanız mümkün olmuyor.

Bir diğer konu da gömülü sistemlere yeni başlayan biri klasik C eğitimini görmüş olsa bile bit bazlı işlemleri gördüğünde kafa karıştırıcı tuhaf semboller olarak yorumlayacaktır. Çünkü eğitimde bit bazlı işlemlere yeterince ağırlık verilmemektedir. Biz bunun önüne geçmek ve gömülü sistemlerde zorluk yaşamamanız için en fazla önemi bu konuya vereceğiz.

Bitwise Operatörler

C dilinde bit bazlı işlem yapabilmek için bitwise operatörleri yani bit bazlı operatörler yer alır. Bu operatörlerin işlevi adından da anlayabileceğiniz gibi diğer operatörler gibi değişken veya değerler ile değil bitler ile işlem

yapmalarıdır. Bu sayede bir değişkenin içine girebilir 1 ve 0 olmak üzere her bir biti istediğiniz gibi düzenleyebilirsiniz. Bu da size muhteşem bir güç verecektir!. İşaretçilerle bütün adreslere ve bit operatörleri ile de bütün bitlere hakim olduğunuz zaman artık donanım avcunuzun içinde demektir.

Şimdi sizi iyice meraklandırmadan bit operatörlerini gösterelim.

Operatör	Açıklama
&	Bitwise AND (AND işlemi)
	Bitwise OR (OR işlemi)
^	Bitwise XOR (Ex-OR işlemi)
<<	Sola Kaydırma
>>	Sağa Kaydırma
~	Bir'in tümleyeni (Tersleme)

Burada operatörlerin işlevlerine baktığınız zaman eğer dijital mantık, dijital elektronik ve mikroişlemci mimarisi bilginiz yoksa bunların çok tuhaf olduğunu düşünebilirsiniz. Bunlar bizim bildiğimiz matematik operatörlerine pek benzemese de mikroişlemcilerin komut kümelerine bile eklenen özel komutlar olduğunu bilmeniz gerekir. Yani mikroişlemci tasarımcıları bu işlemlerin mikroişlemcinin sahip olması gereken en önemli yeteneklerden biri olarak görmekte ve doğrudan donanımsal olarak bunu mikroişlemciye eklemektedir. Bunlar Assembly dilinde olduğu gibi gördüğüm bütün programlama dillerinde de mevcuttur. Bu işin en temeline indiğinizde ise dijital mantık ve mantık kapılarını görürsünüz.

Öncelikle bu operatörlerin çalışma mantığını size tek tek açıkladıktan sonra gömülü sistemler üzerinde kullanımını yani uygulamalı olarak kullanımını sizlere göstereceğiz. Sonrasında ise örnek kodlarla bunu pekiştirmeye çalışacağız.

& – AND (VE) Operatörü

Bu operatör iki değeri alır ve bit bazlı VE işlemi uygular. Bu iki integer değer olsa da her birinin bitlerini birbiri ile karşılaştırır ve sonrasında sırasına göre bu bitleri sonuç olarak verir. Yani bu operatörün aslında değişkenin 200 mü 300 mü olduğu ile bir işi yoktur. Sadece her bir sıradaki bitin 1 veya 0 olup olmadığı ile ilgilenir. $1 \& 1 = 1$, $1 \& 0 = 0$, $0 \& 0 = 0$ diyerek her bir bit ayrı ayrı VE işlemine tabi tutulur. Bu VE işleminin ne olduğunu bilmiyorsanız dijital elektronik yazılarımıza göz atabilirsiniz. Bunu hakkıyla anlayabilmek için bu yönde de bilginiz olması lazımdır. Şimdi VE işleminin doğruluk tablosunu verelim.

A	B	C
0	0	0
0	1	0
1	0	0

1	1	1
---	---	---

Tablodan da göreceğiniz üzere ancak A ve B'nin 1 olması durumunda sonuç 1 olmaktadır. Bunu mantık kapılarında da anlattım ve çok tekrarlamak istemiyorum. Aşağıdaki bağlantıdan lütfen okuyunuz. Aksi halde yarım öğrenmiş olursunuz.

<http://www.lojikprob.com/elektronik/dijital-elektronik-8-mantik-kapilari/>

Şimdi C dilinde bir AND işleminin nasıl yapıldığını görelim.

```
char degisken1 = 0b00011100;
char degisken2 = 0b11011000;
sonuc = degisken1 & degisken2;
```

Burada degisken1 0b00011100 değerine degisken2 ise 0b11011000 değerine sahip. Bu durumda bu ikisine AND işlemi uyguladığımızda operatör her bir biti hanesine göre ele alacak ve AND işlemini uygulayıp sonuca yazacaktır. Bunu mikroişlemci yaparken her bir bite işlemi uygulamak için bir komut harcamamaktadır. Örneğin 64-bitlik bir mikroişlemci tek bir komutta iki 64 bitlik değişkene AND işlemi uygulayabilir. Fakat bu işlemin bağımsız bitler üzerinde yapıldığını unutmamak gerekir. Yani her zaman bit & bit = bit gibi bir işlem ortaya çıkmaktadır. En azından bizim bunu böyle anlamamız gerekli.

Yukarıdaki tabloya göre işlemler şu şekilde yapılacaktır.

0	0	0	1	1	1	0	0
1	1	0	1	1	0	0	0
0	0	0	1	1	0	0	0

Görüldüğü gibi AND işlemi her bir bite ayrı olarak uygulanmakta ve sırasına göre bu sonuca yazılmaktadır. İlk satır degisken1, ikinci satır degisken2, üçüncü satır ise sonuc değişkeninin satırıdır.

| – OR (VEYA) Operatörü

Bu operatör yukarıda verdiğimiz AND operatörü gibi çalışmakta fakat operandları farklı bir işleme tabi tutmaktadır. O da mantıksal (VEYA) işlemidir. Bu mantıksal işlem yine bit bazında olmaktadır. Aşağıda mantıksal OR işleminin doğruluk tablosunu görebilirsiniz.

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Gördüğünüz gibi A veya B sayılarından herhangi biri 1 ise sonuç 1 olmaktadır. Sonucun sıfır olması için bütün sayıların sıfır olma şartı vardır. Bu operatör de oldukça basit gibi görünse de asıl karmaşıklık uygulamada yer almaktadır. Bu operatörleri hiç ummadığınız bir iş için kullanabilirsiniz. Örneğin bir biti bir yapmak için maskeleme işlemi yapılır ve burada anahtar işlem OR işlemidir. Şimdi C dilinde iki değişkeni OR işlemine tabi tutalım ve sonuçlarını görelim.

```
char degisken1 = 0b00011100;  
char degisken2 = 0b11011000;  
sonuc = degisken1 | degisken2;
```

0	0	0	1	1	1	0	0
1	1	0	1	1	0	0	0
1	1	0	1	1	1	0	0

Yukarıdaki sonuçta göreceğiniz üzere OR işlemi bütün bitlere yukarıdaki doğruluk tablosuna göre uygulanmakta ve çıkan sonuç sonuc atlı değişkenin bitlerine yazılmaktadır. Bu işlemi C dilinde & operatörü ile x86 Assembly dilinde AND komutu ile dijital elektronikte mantık kapıları ile elektronikte ise transistörler vasıtasıyla yapabiliriz. Neden bunun alt seviye bir özellik olduğunu buradan anlayabilirsiniz.

^ – XOR Operatörü

XOR operatörü exclusive-OR anlamına gelmektedir. Bunu kolay yoldan anlamak için şöyle diyebiliriz. Operatörler birbirinden farklı mı değil mi operatörü denebilir. Yani operatörler karşılaştırılır farklıysa 1 değilse 0 çıkışı verilir. Bu durumda doğruluk tablosu şu şekilde olacaktır.

A	B	C
0	0	0
0	1	1

1	0	1
1	1	0

Bunu C dilinde uygulamak istersek şöyle bir program yazabiliriz.

```
char degisken1 = 0b00011100;
char degisken2 = 0b11011000;
sonuc = degisken1 ^ degisken2;
```

Bu durumda çıkışımız şu şekilde olacaktır.

0	0	0	1	1	1	0	0
1	1	0	1	1	0	0	0
1	1	0	0	0	1	0	0

NOT (Değil) ya da Bir'in tümleyeni (One's Complement) Operatörü

Temel mantık işlerini yapan operatörlerin sonuncusu NOT (DEĞİL) operatörüdür. Bu operatörün prensibi oldukça basittir. 1 ise 0 yapar, 0 ise 1 yapar. Tekil bir operatör olduğu için sadece bir operand üzerinde işlem yapmaktadır. Bunun doğruluk tablosu şu şekildedir.

A	Sonuç
0	1

1	0
---	---

Şimdi NOT operatörünün nasıl işlediğini göstermek için bir program yazalım ve sonucuna bakalım.

```
char degisken1 = 0b00011100;
sonuc = ~degisken1;
```

0	0	0	1	1	1	0	0
1	1	1	0	0	0	1	1

Görüldüğü gibi birbirinin tam tersi olmakta. Aynı bir bütünün diğer yarısı gibi görünüyor. O yüzden birin tümleyeni adını aldığını görebilirsiniz. 1'in ve 2'nin tümleyeni işlemleri boolean cebirinde önemli bir yere sahiptir. Yine bunları hakkıyla anlayabilmek için temel bilginizin olması gerektiğini görüyorsunuz. O yüzden bilgisayar mühendisliği gibi bölümlerde dijital elektronik gibi dersler yer almaktadır. Ne kadar iyi programcı olursanız olun işin temelini bilemedikçe her zaman eksik anlamış olursunuz.

Bit Kaydırma Operatörleri

C dilinde yer alan << ve >> operatörleri değişkenlerin bitlerini sola ve sağa kaydırmak için kullanılır. Bu kaydırma işlemi gerçek manada uygulanmaktadır. Bu operatörler kaydırılacak veri ve kaç adım (bit) kaydırılacağına dair iki ayrı operand almaktadır. Bu bit kaydırma işlemleri için x86 Assembly dilinde SHR (Shift Right) ve SHL(Shift Left) komutu yer almaktadır. Yani bu işlemlerin C'de operatör olarak verilmesi gibi makine dilinde ayrı komut olarak karşılığı bulunmaktadır.

Şimdi 8 bitlik bir verimiz var diyelim. Bunu sola veya sağa kaydırduğımızda bitler bir bütün halinde yer değiştirecektir. Örneğin bir adım kaydırduğımızda en sağdaki bit kendi solunda yer alan bitin yerine geçecek. Kendi solunda yer alan bitte yine kendi solunda yer alan bitin yerine geçecektir. Böyle böyle en baştaki bite kadar gidecektir. En baştaki bite ne olacak dersiniz artık o bit ortadan kalkacaktır. Bu sola kaydırma işlemi aynı matematikte 5050 sayısının ilk basamağını silip son basamağına ise bir 0 ekleyerek 0505 sayısını elde etmeye benzer. Şimdi bunun nasıl yapıldığına bakalım.

```
char degisken1 = 0b00011100;
```

```
sonuc = degisken1 << 1;
```

Burada sonucu bir adım kaydırarak. Buna göre bütün bitler bir adım sola kayacaktır. Değişkenin ilk değeri ve işlemten sonraki değeri şu şekilde olacaktır.

0	0	0	1	1	1	0	0
0	0	1	1	1	0	0	0

Şimdi aynı değişkeni bir adım sağa kaydırmak için ise şöyle bir kod yazarız.

```
char degisken1 = 0b00011100;
```

```
sonuc = degisken1 >> 1;
```

Bu durumda sonucumuz şöyle olacaktır.

0	0	0	1	1	1	0	0
0	0	0	0	1	1	1	0

Görüldüğü gibi oldukça basit değil mi? Kaç adım kaydırmak istiyorsak adım sayısını yazarak işlemimizi yapıyoruz. Bu bazen küçük veri hücrelerindeki büyük verileri düzenlemek için de kullanılabilir. Örneğin iki adet char (8-bit) değişkeninde 16 bitlik bir veriyi saklamak istiyorsak `degerH` ve `degerL` olmak üzere alçak ve yüksek bitleri saklayacağımız iki değişken tanımlamak zorundayız. Normalde 16-bitlik bir değeri 8 bitlik bir değişkene aktarmak istesek ilk sekiz bitini (burada bitler sağdan başlar) aktaracaktır. Soldan sekiz bitini aktarmak için bunu 8 bit sağa kaydırıp yazdırmamız gereklidir. Bu 8-bitlik gömülü sistemler üzerinde çalışırken sıkça yaptığımız uygulamalardan biridir.

Buraya kadar bit operatörlerinin tamamını anlattık. Fakat uygulamada kullanımını anlatmazsak yarım kalacaktır. Çünkü uygulamada oldukça farklı kullanım yöntemleri mevcuttur. Bunu bir sonraki başlıkta sizlere açıklayacağız.

-47- Bit Bazlı Uygulamalar (Set, Reset, Toggle)

Daha önceki başlıkta bitwise operatörleri anlattığımıza göre bu başlıkta sadece bunun uygulamasını anlatacağız. Bit bazlı operatörlerin kullanım alanları oldukça çeşitlilik gösterebilir. Temelde basit elemanlar olsalar da aynı toplama, çıkarma işlemleri gibi temel işlemlerden biridir. O yüzden bu temel işlemler kullanılarak oldukça karmaşık uygulamalar yapılabilir. Genel kullanım alanları matematik işlemleri, performans uygulamaları ve gömülü sistemlerdeki yazmaç tabanlı işlemlerdir.

Bit biti 1 yapmak (set)

Gömülü sistemlerde çalışırken bir değişkende veya adreste yer alan bir bitin değerini 1 yapmak isteyebilirsiniz. Bunu sadece sayı değeri olarak düşünmeyin. Örneğin gömülü sistemlerde 0x20 adresinde yer alan 8 bitlik bir yazmaç (veri hücresi) içerisinde yer alan 2 numaralı bit aygıtta zamanlayıcıları etkileştirme veya devre dışı bırakma görevini yerine getirebilir. Bilgisayarlardaki donanım kontrolü de belirli bellek noktalarında belirli değerleri değiştirme vasıtasıyla yapılmaktadır. Biz tam olarak o biti 1 yapmak istersek şöyle bir yöntemi kullanmamız gerecektir.

```
bellek_hucresi |= (1<<bit_degeri);
```

Bu işleme maskeleme adı verilmektedir. Bilmeyen birine tuhaf işaretlerin bir araya gelmesi gibi görünebilir. Gömülü sistem geliştiriciliğinde de başta en çok zorluk çekeceğiniz nokta budur. Bit işlemlerini anlayan ilerler anlamayan ise sıfır noktasında kalır. Bunun için bunu anlamanız noktasında azami gayret sarf etmekteyim. Şimdi yukarıda gördüğünüz örnek söz dizimini gerçek bir koda çevirelim ve bunu en derinlemesine açıklayalım.

```
PORTD |= (1<<7);
```

Gömülü sistemlerde yazmaç seviyesinde çalışıyorsanız bir ledi yakmak için bile böyle bir kod yazmak zorunda kalırsınız. Bunu anlamadan en basit bir işlem olan led yakma işini bile yapamazsınız.

Burada her C programında olduğu gibi öncelikle eşittirin veya birleşik operatörün sağ tarafında gerçekleştirilen işleme bakmamız gereklidir. Zaten parantez içinde olduğu için de doğrudan parantez içlerine odaklanmamız gerekli. Parantezin içine odaklandığımızda $1 \ll 7$ işlemini görürüz. Yani 1 sabit değeri 7 adım sola kaydırılmakta. Bunun nasıl işlediğini tablo vasıtasıyla gösterelim.

0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0

İkili formatta göstermek istersek bu işlemten önce elimizde 1 değeri yani 0b00000001 değeri yer alırken bu işlem gerçekleştikten sonra bit değeri yedi adım kayarak yedinci bite gelmiş durumdadır. Burada bitlerin sıfırdan ve sağdan başladığını unutmayın. Yedinci bit 8-bitlik bir verinin son basamağıdır. Burada değerimiz 0b10000000 olmakta. Buradan da anlıyoruz ki $1 \ll 7$ yazmak 0b10000000 sabitini yazmanın kısa yoludur. Benim için $(1 \ll 7)$ ifadesi daha anlamlı gelmektedir. Çünkü bunu (değer<<bitkonumu) olarak kodlamaktayım.

$(1 \ll 7)$ diyerek yedinci biti bir yapacağımızı öğrendik. Şimdi bunu PORTD adı verilen değişkene (adrese) uygulama zamanı. Başlangıçta bunun atama operatörü (=) ile uygulanacağını düşünebilirsiniz. Ama atama operatörü ile atadığımız değer PORTD'nin bütün bitlerini değiştirecektir. Yani eğer =

operatörünü kullanırsak PORTD'nin değeri 0x10000000 olacaktır. Biz bunu istemiyoruz çünkü diğer bitlerde farklı farklı görevler için değerler yer almakta. Bu yüzden sadece PORTD'nin 7 numaralı bitinin bir yapılmasını istiyor geri kalana dokunulmamasını istiyoruz. Bunu ise mantıksal bir işlem olan OR işlemi ile yapacağız.

OR işleminin burada bunun için kullanılması şaşırtıcı değil mi? Görüldüğü gibi bu operatörler oldukça esnek ve işimize gelen yerde kullanılabilmesi için biraz kafa yormak gerekiyor. Şimdi |= operatörünün ne iş yaptığını size anlatalım.

Aynı toplama, çıkarma, çarpma ve bölme işlemlerinde işi kısaltmak adına birleşik operatörler (+=, -=, *=, /=) kullanılıyorsa mantık işlemlerinde de kodu kısaltma adına birleşik operatörler kullanılmaktadır. Yani &=, |= ve ^= operatörleri ile bit kaydırma operatörlerini bu şekilde birleşik halde görebilirsiniz. Ama uygulamada bit kaydırma operatörleri genelde ayrı kullanılmaktadır.

Bu durumda,

$A \&= B$ operatörü $A = A \& B$, $A |= B$ operatörü $A = A | B$, $A ^= B$ operatörü ise $A = A \wedge B$ işlemine eşittir.

Bu kurala göre yukarıdaki $PORTD |= (1 << 7);$ işlemine baktığımızda aslında şu işlemin yapıldığını görürüz.

```
PORTD = PORTD | (1 << 7);
```

Görüldüğü gibi $(1 << 7)$ işleminden dönen değer PORTD üzerine OR işlemi ile uygulanıyor. Bu yüzden buna maskeleme denmekte. O halde bunu daha anlaşılır olması için şu halde yazıp sonrasında ise nasıl çalıştığını gösterebiliriz.

```
PORTD = PORTD | 0b10000000
```

Şimdi PORTD'nin değerleri ve 0b10000000 değerinin PORTD'ye OR işlemi ile uygulandıktan sonra değerlerine bakalım.

x	x	x	x	x	x	x	x
1	0	0	0	0	0	0	0
1	x	x	x	x	x	x	x

Neden bu kadar çok x var diye sorarsanız bunlar bizi ilgilendirmeyen değerleri temsil için gösterilmiştir. Bu durumda PORTD'nin 7 numaralı bitinin de x olduğunu görebilirsiniz. Bunun değeri de aslında bizi ilgilendirmemektedir. Çünkü ister 1 olsun ister 0 olsun OR işlemine tabi tutulduktan sonra muhakkak 1 olacaktır. Geri kalan değerleri ise 0 ile OR işlemine tabi tuttuğumuzdan ister 1 olsun ister 0 olsun herhangi bir şekilde değişmeyecektir. Bir biti bir yapmanın en garanti yolunun OR işleminden geçtiğini görmüş oldunuz. Harvard mimarisinde veya bazı mikrodenetleyici mimarilerinde makine komutu olarak bit biti bir yapma komutu yer alıp bizi o kadar çok uğraştırmasa da C dili von neumann mimarisine göre tasarlanmıştır. Von neumann mimarisinde ise makine komutları bu şekilde olduğundan bize bitwise operatörler olarak miras kalmıştır. Makine dilinde olmasına rağmen C dilinde bir biti bir veya sıfır yapma operatörü veya komutu bulunmayışı o dilin eksikliği olarak görünebilir. Eğer bu dil o mimariye göre tasarlansaydı hak verebilirdik. Fakat burada bu işlem mimari farklılığından dolayı farklı bir yolla da olsa yapılabilmektedir.

Bir biti 0 yapmak (reset)

Bir biti bir yapmayı öğrenseniz de aynı kodla bir biti sıfır yapmanız mümkün değildir. O yüzden bu sefer farklı operatörlerle kısmen benzer bir işlemi

gerçekleştireceğiz. Aslında bir biti sıfır yapmanın bir yapmaktan biraz daha zor olduğunu göreceksiniz. Fakat bunları bir noktadan sonra ezberleyeceğiniz için sadece değerleri değiştirerek printf fonksiyonunu yazmak kadar kolay bir şekilde yazabilirsiniz. Biz yazdığınız kodları anlamamanızın gerektiğini savunduğumuz için başta teorik bilgiyi sizin en iyi şekilde anlamanız için çaba sarf ediyoruz. Pratik kazanmak kişisel bir tecrübedir ve size bağlıdır. Kişisel deneyimlerimden yola çıkarak size şunu söyleyebilirim. Bir teoriyi anlamak 5 pratik yapmaya bedeldir. Çünkü teoriyi anlayarak aslında zihninizde bir pratik yapıyorsunuz ve tecrübe kazanıyorsunuz.

Bir biti sıfır yapmak için şöyle bir kod prototipi kullanılabilir.

```
degisken_adi &= ~(1<<bit_numarasi);
```

Yukarıda bir biti bir yapmak için OR işleminin anahtar eleman olduğunu burada ise AND işleminin bu görevi yerine getirdiğini görebiliriz. İki mantık işlemine baktığımızda aralarında birbirine zıt bir ilişki görebiliriz. OR işlemi dört ihtimalin üçünde 1 çıkışını vermekte AND işlemi ise dört ihtimalin üçünde 0 çıkışını vermektedir. Şimdi gerçek bir kod yazalım ve bunun nasıl çalıştığını görelim.

```
PORTD &= ~(1<<7);
```

Burada (1<<7) diyerek yine 7 numaralı bit üzerinde işlem yapacağımızı anlayabilirsiniz. Burada elimizde 0b10000000 sabiti var fakat parantezin solunda bir operatörü daha görmekteyiz. Bu operatör NOT operatörü olup tersleme görevini yerine getirmektedir. Yani (1<<7) 0b10000000 iken ~ operatörü tarafından 0b01111111 değerine çeviriliyor. Burada neden (0<<7) diye düz bir şekilde yazmak varken (1<<7) yazıp sonra da bunu ters çeviriyoruz diyebilirsiniz. Bunun iki sebebi vardır. Birincisi (0<<7) işleminden dönen değer 0 olacaktır. Çünkü sıfır nerede olursa olsun sıfırdır. İkincisi ise

bizim diğer bitlerde 1 değerine ihtiyacımız vardır. Bu maskeleye için gereklidir çünkü eğer ilk bit 1 ise onun 1 kalabilmesi için $1 \& 1 = 1$ işlemine tabi tutulması gerekecektir.

O halde PORTD şöyle bir değer ile AND işlemine tabi tutulmaktadır.

```
PORTD &= 0b01111111;
```

```
// yani
```

```
PORTD = PORTD & 0b01111111;
```

PORTD ile 0b01111111; işleminin nasıl gerçekleştiğine bir göz atalım.

x	x	x	x	x	x	x	x
0	1	1	1	1	1	1	1
0	x	x	x	x	x	x	x

Burada bir değere 1 ile AND işlemi uygulanırsa o değerde herhangi bir değişim olmayacaktır. Bu aynı 0 ile OR işlemi uygulamak gibidir. Bu yüzden sayıları ters çevirdik. Biz OR işlemi ile bir sayıyı bir iken sıfır asla yapamayız. AND işlemi ile de sıfır iken bir yapamayız. Fakat tam tersini kolaylıkla yapmamız mümkündür. Burada da 0 değerini hangi değere AND işlemi ile beraber uygularsak o değer muhakkak sıfır olacaktır. Diğer değerlere 1 ile AND işlemi uygulandığından dolayı değerleri ne olursa olsun asla değişmeyecektir.

Bir biti değiştirmek

Bu sefer biti sıfır veya bir yapacağımızı veya bitin değerinin ne olduğunu bilmiyoruz diyelim. Tek kuralımız bu bit bir ise sıfır olacak, sıfır ise bir olacak.

Başka herhangi bir işlem söz konusu olmayacak. Yani açıp kapama (toggle) işlemi yapmaktayız. Bunu bu sefer de XOR işlemi ile yaparız.

```
degisken_adi ^= (1 << bit_konumu);
```

Burada XOR işleminin “Aynı ise 1, farklı ise 0 yap” prensibi kullanılmaktadır. 1 değerini bir bite uyguladığımızda o bit 1 ise sıfır olmakta. Sıfır ise 1 ile farklı olduğu için 1 olmaktadır.

Bunu uzun uzadıya açıklamamıza gerek olmadığını düşünüyoruz çünkü mantığını anladınız. Aynı konuyu C ile AVR programlama derslerinde de anlatmıştım fakat bunu oraya bakmadan bir daha anlattım. İki yazıdan da ayrı ayrı faydalanabilirsiniz.

-48- Bit Alanları ve Enümeratörler

C programlama dilinin diğer konuları arasında yer alan bit alanı (bit field) ve enümeratör (enumeration) konularını bu yazıda anlatacağız. Normalde baktığımızda çok da gerekli olmayan ayrıntı konularmış gibi görünse de bunlar ileri seviye C programlamada bize yardımcı olmaktadır. Acemi biri pek sık kullanmasa da ileri seviye programlarda ve kütüphanelerde karşımıza

çıkmaktadır. O yüzden başta sık kullanacak olmasak da bunları bilmemiz, karşımıza çıkacak kütüphaneleri anlamamız yönünde önemlidir.

Bit Alanları

Ben bit alanlarına kitaptan baktığımda performansı artırmak için teferruattan ibaret olduğunu ve karşıma pek sık çıkmayacağını düşünüyordum. Ama ne zaman STM32 için CMSIS kütüphanesini incelediysem o zaman bit alanlarının en alt seviye programcılıkta bile kullanıldığını gördüm. Aşağıda bit alanlarının kullanıldığı bir kütüphanedeki örnek kod parçasını görebilirsiniz.

```
/**
 * \brief Union type to access the Application Program Status Register (APSR).
 */
typedef union
{
    struct
    {
        uint32_t _reserved0:27;           /*!< bit: 0..26 Reserved */
        uint32_t Q:1;                     /*!< bit: 27 Saturation condition flag */
        uint32_t V:1;                     /*!< bit: 28 Overflow condition code flag */
        uint32_t C:1;                     /*!< bit: 29 Carry condition code flag */
        uint32_t Z:1;                     /*!< bit: 30 Zero condition code flag */
        uint32_t N:1;                     /*!< bit: 31 Negative condition code flag */
    } b;                                   /*!< Structure used for bit access */
    uint32_t w;                           /*!< Type used for word access */
} APSR_Type;
```

Bit alanları kısaca belli bir bit genişliğinde veri tanımlamak ve o verileri hafızada bir yere sığdırmak demektir. Örneğin biz bir değer için 16 bitlik bir değişken tanımlarken 16 bit içerisinde 4 adet 4 bitlik değer tanımlayıp saklayabiliriz. Yani değişken içerisinde birden fazla küçük değişkeni saklama işlemine bit alanı denmektedir. Bu hafızadan tasarruf etmemizi sağlamaktadır. Örnek bir bit alanı kodu şöyle olabilir.

struct

```
{  
uint32_t rezerve:24;  
uint32_t ayar_biti:2;  
uint32_t bayrak_biti:2;  
uint32_t kontrol_biti:2;  
uint32_t kesme_biti:2;  
  
} zamanlayici_yazmaci;
```

Burada elimizde 32 bitlik bir bellek bölümü olsun. Bu bellek bölümü eğer özel fonksiyonların yürütüldüğü bir bölümse SFR (Special Function Register) yani özel fonksiyon yazmacı olarak da adlandırılabilir. Biz bu değişkende bazı konumda yer alan bitleri değiştirdiğimiz zaman özel fonksiyonlar yürütülmektedir. Mesela bir sistemdeki kesmeleri denetlemek istediğimizde 32 bitlik değişkende son iki biti denetlememiz gereklidir. Yani 0x11, 0x00, 0x10 ya da 0x01 değerlerini kullanabiliriz. Bu dört değer için koca bir 32-bitlik yeri işgal etmeye gerek var mıdır? Bu durumda bit alanları vasıtasıyla 32 bitlik veri alanında kendimize 2 bitlik bir yer açarız. Yukarıda *uint32_t kesme_biti:2;* komutunda bunu yaptık. Bit alanları bu yüzden gömülü sistemlerde oldukça kullanışlı olmaktadır. Çünkü gömülü sistemlerde bazen bir hafıza yerinde pek çok fonksiyon yer alabilir. Sürekli bunlar üzerinde farklı değişkenlerin değerini maskeleye yöntemi ile aktararak çalışmak yerine doğrudan bit alanları ile 32 bitlik yazmaç verisi üzerinde çalışma imkanına sahip oluruz. Üstelik struct olarak yapı değişkeni yaptığımız için pek çok aynı iskelet üzerine kurulmuş farklı yazmaçları bu şekilde kontrol edebiliriz.

Bit alanlarını CMSIS kütüphanesini incelediğim vakit uygulamalı olarak göstereceğim. Şimdilik sadece bu kadarını bilmeniz yeterli.

Enumeratörler

Enumeratörler bir değişkenin değerlerini adlandırma yöntemidir. Böylelikle program içinde değişkenin değerleri ile değil o değerlere karşılık gelen adlar ile işlem yapma imkanımız olur. Enümeratörlerin kullanım sebebi tamamen kolaylık sağlama adınadır. Sayısal olan değerleri sözel olan değerlerle değiştirerek program daha yüksek seviyeye çıkarılmaktadır. Örneğin ay adında bir değişkenimiz olsun ve bu ay verisi içersin. Elbette bu ay verisi sayısal olmak zorunda olduğu için Ocak, Şubat şeklinde değil 0, 1 ... şeklinde temsil edilmek zorundadır. Bir değişkene Ocak, Şubat, Mart ... diye değer atayıp bu değerler üzerinde kontrol ve aritmetik işlem yapmak o kadar kolay değildir. Eğer karakter dizilerinden bahsediyorsanız o apayrı bir seviyededir ve beraberinde türlü zorluklar getirmektedir. Bu yüzden program yazarken temsil edilmek üzere enümeratörler kullanılmaktadır. Bir örnekle size gösterelim.

```
enum aylar
{
OCAK, SUBAT, MART, NISAN, MAYIS, HAZIRAN, TEMMUZ, AGUSTOS,
EYLUL, EKIM, KASIM, ARALIK
};

// Tanımlama
```

```
enum aylar ay1, ay2;
```

Burada aylar tipinde bir değer dizisi tanımladık ve değer olarak kendisine OCAK, SUBAT, MART, NISAN, MAYIS... diye giden değerleri verdir. Bu değerlerin hepsi birer ilizyondur! İşin aslınca OCAK = 0, SUBAT = 1, MART = 3 ... diye giden değerlere sahiptir. Yani biz burada OCAK yazdığımızda 0 yazmaktan başka bir şey yapmamış oluyoruz. Bu değerler printf gibi

fonksiyonlarla yazdırıldığı zaman gerçek sayı değerlerini göstermektedir. Fakat biz program yazarken kolaylık olsun diye `ay1 = OCAK;` yazabiliriz. Bu durumda `ay1` değerine 0 aktarılmaktadır. Ayrıca bu değer üzerinde aritmetik veya mantık işlemleri de yapabiliriz. Bu sadece program yazarken bize kolaylık sağlamakta ve verilerin konuşma diline daha yakın sembolleri ile program yazma yolu açılmaktadır.

Kullanmak şart mıdır? Elbette hayır. Fakat programın daha okunur olmasını ve kolay anlaşılmasını istiyorsanız kullanabilirsiniz. Ben şahsen bu tarz şeyler için önışlemci direktifi olan `#define` tanımlamasını kullanmaktayım. Böylelikle hafızada herhangi bir yer işgal edilmemiş oluyor.

-49- Dosya İşlemleri ve `fopen()` fonksiyonu

Biz bilgisayar kullandığımızda bütün bilgilerin dosyalar halinde saklandığını, okunduğunu ve işlendiğini fark ederiz. Buraya kadar konsol ekranına yazdırdığımız ve RAM belleğe kaydedip üzerinde işlem yaptığımız verileri harddisk, CD sürücü, SSD gibi ikincil belleğe kaydetmek veya başka bir program tarafından işletmek için bizim dosya haline getirmemiz şarttır. Dosya sistemini ve idaresini işletim sistemine bırakıp işletim sistemine özgü komutlarla işletsek de C dilinde standart kütüphanedeki başlık dosyalarında dosya işlemlerinin yer aldığı fonksiyonlar vardır. Bu fonksiyonlar arka planda

işletim sisteminin dosyalama birimine göre uygun komutları üreterek dosya oluşturur, dosyaları kayıt eder, dosyaları açar ve dosyaları okur. Dosya fonksiyonları ile dosyalar üzerinde bütün işlemleri yapabiliriz.

Peki bu dosya işlemlerini nasıl yapacağız dersiniz bir string işleminden veya sdtout akışına yazdırdığımız printf(), scanf() fonksiyonlarından çok farklı olmadığını söyleyebiliriz. C dili dosyaları büyük bir karakter dizisi olarak görmektedir. İşin aslında dosyalar da bu şekilde işlenmekte ve okunmaktadır. Bir dosya bellekte belli bir aralıktaki sektörleri kaplamakta ve baştan itibaren bayt bayt okunmaktadır. Aynı zamanda dosyalar ele alma bakımından karakter dizileri ile ortak noktalara sahiptir. Örneğin önceden anlattığımız gibi karakter dizileri '\0' işareti ile biterken dosyalar ise EOF (End of File) işareti ile bitmektedir.

Bir dosya açıldığı zaman akış bu dosya ile ilişkilendirilir. Daha önce söylediğimiz gibi standart giriş, standart çıkış ve standart hata akışları artık konsol ekranı veya klavye değil dosya üzerinde işlem yapar. Bu akışlar vasıtasıyla klavye, ekran ve dosya arasındaki iletişim sağlanır. C dilinde dosyalar ayrı bir veri kaynağı olarak ele alınmaktadır. Dosya işlemlerinin olması sayesinde yazdığımız program bir Excel tablosu, veri tabanı üretebileceği gibi resim ve ses dosyaları da üretebilir. Bunun sınırı görmek oldukça güçtür ve hangi tipte nasıl dosya üreteceğiniz ve dosyaları okuyacağınız size kalmıştır.

C dilinin dosyaları bir karakter dizisi gibi gördüğünü bilmeniz gerekir. Yani bir dosya açtığınız zaman dosya 0 numaralı bayttan itibaren sıra ile okunup işlem yapılır. C programı dosyanın nerede bittiğini kestiremez. O yüzden EOF işareti denetlenir ve dosyanın nerede bittiği bulunur. Dosya işlemlerine

baktığınızda aslında şimdiye kadar yaptığımız konsol uygulamalarından çok da farklı olmadığını göreceksiniz.

Dosya İşlemlerine Giriş

Öncelikle dosya işlemlerinden bahsetmeden C standart kütüphanesinde `wchar.h` adında bir başlık dosyası olduğunu belirtelim. Normalde biz en fazla genişletilmiş ASCII (8-bit) karakterleri kullansak da `wchar.h` ile 16 bit karakterler kullanılabilir hale gelmiştir. Artık günümüzde dosya işlemlerinde genişletilmiş karakter verileri ile de karşılaşabileceğinizden bunu dipnot olarak buraya bırakalım.

C dilinde temel olarak dosyalarla şu işlemleri yaparız:

- Yeni bir dosya oluşturmak
- Mevcut bir dosyayı açmak
- Dosyadan veri okumak
- Dosyaya veri yazmak
- Dosyada belli bir yere gitmek
- Dosyayı kapatmak

Bu işlemleri yapmak için `fopen()`, `fscanf()`, `fgetc()`, `fprintf()`, `fputs()`, `fseek()`, `frewind()` ve `fclose()` fonksiyonları kullanılır. Burada fonksiyon adları size tanıdık gelmelidir. `scanf()` yerine `fscanf()`, `printf()` yerine `fprintf()` kullanıyoruz. Fonksiyonlar işlev bakımından da benzer olup sadece kullanım alanları farklı olmaktadır. O yüzden siz önceden bahsettiğimiz gibi dosya fonksiyonlarını kullanırken yabancılık çekmeyeceksiniz.

Şimdi bu saydığımız fonksiyonların ne işe yaradığına bir bakalım.

fopen() fonksiyonu

fopen() fonksiyonu dosya işlemlerinde en başta kullanacağımız fonksiyondur. Adından da anlaşılacağı üzere bir dosya açmak (oluşturmak) veya var olan bir dosyayı amak için kullanılmaktadır. fopen() fonksiyonu FILE tipinde bir dosya işaretçisi geri döndürmektedir. Geri döndürülen işaretçiyi biz kendi oluşturduğumuz dosya işaretçisine aktararak bu değer üzerinde okuma, yazma ve diğer işlemleri yaparız. Şimdi fonksiyonun prototipine bakalım.

```
FILE * fopen ( const char * filename, const char * mode );
```

Burada yukarıda dediğimiz gibi fopen() fonksiyonu FILE adında dosya işaretçisini geri döndürmektedir. Bunu dosyanın bellekteki konumu olarak düşünebilirsiniz. Bu adresten itibaren bellek üzerinde dosya verisini okur ve yazarız. Aldığı parametreler baktığımızda ise filename olarak dosya adı ve mode olarak hangi modda ele alınacağı yazmakta. Bu dosya adı işletim sisteminin dosyalama sistemine gönderilecek addır. Kısacası burada program şu dosyayı getir demekte ve işletim sisteminden bu dosyayı okumaktadır. Burada işletim sisteminin Path (Yol) kurallarına uygun bir söz diziminde dosya adını yazmamız gerektiğini unutmayın. Örneğin src klasöründeki main.c dosyasına erişmek için “src/main.c” yazmamız gerekecektir. Ayrıca dosya uzantısına da dikkat etmemiz gerektiğini söyleyelim. Dosya oluştururken de hangi uzantıda oluşturacaksak uzantıyı buraya belirtmemiz gereklidir. Dosya adını anladığınıza göre mod kısmına bir göz atalım.

Biz fopen() fonksiyonunun hem dosya oluşturabildiğini hem de var olan dosyayı okuyabildiğini söylemiştik. Bu mod parametresine göre fopen() fonksiyonu dosyayı sadece okur veya hem okur hem yazar ya da yeni bir dosya oluşturabilir. Biz mod kısmına yazacağımız karakter dizisi sabitleri ile bunu belirleriz. Hangi karakter dizilerini kullanmamız gerektiğini ise aşağıdaki tablodan görebilirsiniz.

“r”	Okuma yapar. Sadece dosyadan veri okumaya izin verir ve mevcut bir dosyayı açar.
“w”	Yazmak üzere yeni bir dosya oluşturur ve boş dosya üzerinde yazma işlemi uygular. Eğer aynı adda bir dosya varsa eskisini siler ve yenisini onun yerine kaydeder.
“a”	Bu fonksiyon dosyanın sonundan itibaren veri yazmak için kullanılır. Eski dosyadaki veriler silinmez fakat üzerine ilave edilir.
“r+”	Belgeyi hem okuma hem de yazma için açar. Belgenin mevcut olması gerekir.
“w+”	Boş bir dosyayı hem okuma hem de yazma yapmak için oluşturur. Eğer aynı adda bir dosya varsa eski dosya silinir ve yenisi onun yerine yazılır.
“a+”	Hem yazma hem de okuma yapmak için dosyayı açar ve dosyanın sonundan itibaren işleme alır. Belge yoksa yeniden oluşturulur. Okuma için atlama fonksiyonları (imleç gibi düşünebilirsiniz) olsa da yazma her zaman EOF işaretinden itibaren yapılır.

Yukarıda gördüğünüz üzere bir dosyayı farklı amaçlar için açabiliriz. Bu amaçlara yönelik de çeşitli işlemler yapmamıza izin verilir. Bu arada C dilinde dosyaların iki tipte açıldığından bahsetmemiz gerekir. Dosyalar metin olarak veya ikilik değerde (binary) ele alınabilir. Eğer ikilik olarak ele alınan dosya

varsa en sona “b” koymamız gerekecektir. “rb”, “wb”, “ab”, “r+b”, “w+b”, “a+b” gibi...

Şimdi fopen() fonksiyonunun nasıl çalıştığını görmek için örnek bir kod yazalım. Daha dosya işlemlerini yapmadığımız için uygulamaya geçemiyoruz. Örnek kodumuzda bir dosya tipinde işaretçi değişken oluşturalım ve buna dosya işaretçisini ekleyelim.

```
#include <stdio.h>
int main()
{
FILE *dosya;
dosya = fopen("metin.txt","w");
}
```

Bu şekilde kodumuzu yazdıktan sonra artık metin.txt dosyası üzerinde yapacağımız bütün işlemleri dosya adlı dosya işaretçisi vasıtasıyla yapabiliriz. Kısacası bundan sonra metin.txt dosyasının adını unutacağız ve dosya adlı işaretçiyi kullanacağız.

Bir sonraki yazıda açtığımız dosya üzerinde işlem yapmayı göreceğiz.

-50- fprintf() ve fclose() ile En Basit Dosya Uygulaması

Önceki başlıkta fopen() fonksiyonu ile dosya açmayı anlatmış ve buna oluşturduğumuz dosya işaretçisini nasıl eklediğimizi göstermiştik. Şimdi oluşturduğumuz dosya işaretçisi vasıtasıyla dosya üzerinde değişiklik yapalım ve bunu kaydedelim. Bunun için fopen() fonksiyonunun yanında iki fonksiyona daha ihtiyacımız olacaktır. Bunlar fprintf() ve fclose() fonksiyonlarıdır.

fprintf() fonksiyonu

Bu fonksiyon dosyaya formatlı biçimde karakter dizisi verisi yazmak için kullanılır. Formatlama aynı printf() fonksiyonunda olduğu gibidir. Fakat bu sefer ekran yerine dosyaya veri yazılmaktadır. İşleyiş bakımından sprintf() fonksiyonuna da benzemektedir. Fonksiyonun prototipi şu şekildedir.

```
int fprintf ( FILE * stream, const char * format, ... );
```

Burada FILE * stream olarak belirtilen dosya akışıdır. Bizim fopen() ile açıp dosya işaretçisine eklediğimiz dosyayı buraya yazmamız gereklidir. const char* format ile belirtilen karakter dizisi sabiti ise printf()'den bildiğiniz üzere iki tırnak arasına yazılan metin ve format bilgilerini içermektedir. Sonrasında ise aynı printf()'de olduğu gibi formatları yazarak fonksiyonu bitiririz. Yine değer olarak int tipinde toplam yazılan bayt sayısını bize vermektedir. Formatları tekrar tekrar buraya yazmıyorum merak edenler printf() fonksiyonunu açıkladığım kısma tekrar bakabilir.

fclose() fonksiyonu

fprintf() gibi stdio.h başlık dosyasında yer alan fclose() fonksiyonu açılmış dosyayı kapatır ve akıştan keser. Bu durumda yazılmamış ve okunamamış tampon veri de ortadan kaybolmaktadır. Biz fopen() ile açtığımız dosyayı işimiz bittikten sonra fclose() ile kapatmamız gerekir. Fonksiyonun prototipi şu şekildedir.

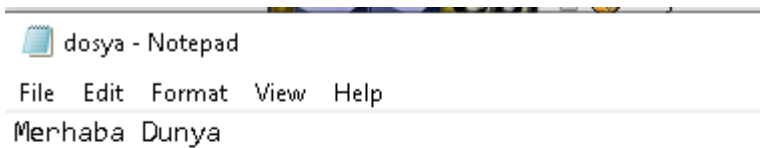
```
int fclose ( FILE * stream );
```

Görüldüğü gibi FILE *stream kısmına FILE tipindeki dosya işaretçisinin adını yazıyoruz ve bağlı olduğu dosyayı kapatıyor. Şimdi öğrendiğimiz bu üç fonksiyon ile basit bir uygulama yapalım. Uygulamamız bir dosya oluşturacak, bu dosyaya veri yazacak ve bu dosyayı kapatacaktır. Derslerin en başında yaptığımız “Merhaba Dünya!” programını dosyalar vasıtasıyla yapalım.

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

```
int main(int argc, char** argv) {  
    FILE *dosya;  
    dosya = fopen ("dosya.txt", "w");  
    fprintf(dosya, "Merhaba Dunya");  
    fclose(dosya);  
    return 0;  
}
```

Programı çalıştırdığımızda konsol ekranında herhangi bir çıkış vermese de .exe dosyasının olduğu klasörde bir dosyanın meydana geldiğini fark edeceksiniz. Bunu notepad veya diğer bir metin işlem programı ile açtığımızda içinde “Merhaba Dunya” yazısını göreceksiniz.



Bunca zaman ekrana yazdırdığımız değerler silinip gidiyordu. Ama dosya öyle değildir. Artık sonuçları dosyaya yazdırmayı ve bunu kalıcı hale getirmeyi biliyorsunuz. Basit görünse de bu üç fonksiyonla artık programlarınızın çıktısını kalıcı hale getirmeyi öğrendiniz. Üstelik bu program çıktılarını farklı programlar da okuyup yorumlayabilir. Bazen Excel gibi programlarla bu verileri grafik haline getirip görselleştirebilir veya diğer programlar tarafından kullanılabilir hale getirebilirsiniz.

Bu yazıda olabilecek en basit dosya uygulamasını yapsak da ilerleyen uygulamalar biraz daha teferruatlı olduğu için kafanızı karıştırabilir. Bu arada bütün dosyalarla ilgili fonksiyonları `stdio.h` başlık dosyasında bulabilirsiniz. Şimdilik üç tanesini anlatsak da ilerleyen başlıklarda devamını anlatacağız.

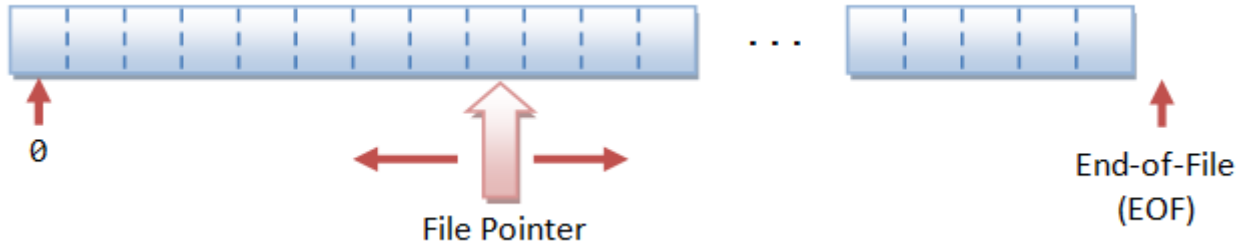
-51- Dosya İşaretçisi ve `feof()` Fonksiyonu

Daha öncesinde bir dosya oluşturmayı ve bu dosyaya veri yazmayı anlattık. Fakat dosya işlemleri sadece bunlarla sınırlı değildir. `stdio.h` başlık dosyasına baktığımızda daha farklı dosya okuma ve yazma fonksiyonlarının olduğunu görmekteyiz. Fakat daha öncesinde dosya işaretçisi ve `feof()` fonksiyonlarından bahsetmek istiyoruz.

Dosya Okuma ve Yazma Fonksiyonları

Daha öncesinde bir dosya işaretçisinin olduğunu ve dosyaya aynı bir karakter dizisi gibi bakıldığını söylemiştik. Dosya işaretçisi dosyanın başından EOF

işaretine yani dosya bitimine kadar ileri ve geri olarak çizgisel olarak ilerlemekte ve bu işaret ettiği bölgeler üzerine okuma ve yazma yapmaktadır. Bunu bir teyp bandı ve teyp kafası gibi düşünebilirsiniz. Teyp bandının başlangıcı ve sonu olmakta ve teyp kafası sadece bir noktayı okumaktadır. Teyp sağa veya sola ilerleyerek okuma konumunu değiştirmektedir. Aşağıdaki resim tam da bizim dediğimizi göstermektedir.



Resim

https://www.ntu.edu.sg/home/ehchua/programming/java/images/IO_RandomAccessFile.png

Burada dosya işaretçisi her yazma ve okuma yaptığı zaman işaretçiyi sağa kaydırmakta ve dosyanın en son kısmına geçmektedir. Bunu aynı yazı yazdığınız zaman yanıp sönen imlece benzetebilirsiniz. Klavyeden her harfi girdiğiniz zaman imleç birer birer sağa kaymakta ve yazdığınız harfleri boş kısma eklemektedir. Siz dilediğiniz zaman imleci başa veya metinde herhangi bir yere alabilirsiniz. Bu imleçle (işaretçi) ile alakalı fonksiyonları daha sonra anlatacağız. Şimdilik konuyu basit seviyede tutmak için sadece okuma ve yazma üzerinden gideceğiz ve imleci kendi halinde bırakacağız.

Burada EOF (End of File) işaretini de dikkat çekmek isterim. EOF işareti karakter dizilerinde kullanılan '\0' gibi veya klavyedeki Return, Backspace tuşları gibi bir ASCII karakteri değildir. Aslına baktığınızda EOF işaretinin C standartında bile olmadığını görürsünüz. Bu işaret sistem tarafından belirlenir

ve EOF diyerek kullanılır. Bunun değeri sisteme göre de değişmektedir. EOF dosyanın bittiği haberini bize vermektedir. EOF işareti bir ASCII karakteri olmasa da bunu EOF olarak bayt şeklinde okuyup değerlendirebiliriz. Yalnız her zaman EOF diye belirtmemiz gereklidir. Çünkü EOF'un ne değerde olacağı kesin değildir. Peki EOF'u dosya içerisinde nasıl bulacağız?, diye soracağınızı biliyorum. O halde feof() fonksiyonunu anlatarak konumuza devam edelim.

feof() fonksiyonu

Bu fonksiyon dosya işaretçisinin işaret ettiği dosya bölümünde EOF işaretinin olup olmadığını denetler. Eğer EOF işareti varsa sıfırdan farklı bir değer döndürür. Eğer EOF işareti yoksa sıfır değerini döndürür. Genellikle dosya okuma ve yazma işlemlerinde if karar yapısı veya while döngüsü ile beraber kullanılır. EOF işaretine rastlanıldığında dosya kapatılır veya oradan itibaren yazıma devam edilir.

Şimdi feof() fonksiyonunun prototipine bakalım.

```
int feof ( FILE * stream );
```

Bu fonksiyon yukarıda bahsettiğimiz gibi int tipinde değer döndürmekte. Dönen değer sıfırdan farklı ise EOF tespit edilmiştir eğer sıfır ise EOF tespit edilmemiştir. Aldığı argüman FILE * stream olduğu için dosya işaretçisi tipinde yani açıp üzerinde çalıştığımız FILE tipindeki işaretçiyi buna argüman olarak aktarmamız gereklidir. Unutmayın, feof() fonksiyonu dosyada EOF olup olmadığını değil dosya işaretçisinin işaret ettiği yerde EOF olup olmadığını bize bildirir. Şimdi bununla ilgili bir örneğe göz atalım. Bu örneği cplusplus.com sitesinden aldığımı belirteyim.

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



```

int main ()
{
    FILE * pFile;
    int n = 0;
    pFile = fopen ("dosya.txt","rb");
    if (pFile==NULL) perror ("Dosya Bulunmadi");
    else
    {
        while (fgetc(pFile) != EOF) {
            ++n;
        }
        if (feof(pFile)) {
            puts ("End of File Okundu.");
            printf ("Toplam okunan bayt: %d\n", n);
        }
        else puts ("End of File Okunamadi.");
        fclose (pFile);
    }
    system("PAUSE");
    return 0;
}

```

Bu program ele alınan dosyanın mevcut olup olmadığına ve mevcutsa bulundurduğu içeriğe göre farklı çıkışlar verecektir. Fakat öncesinde sizin bu programı iyice sindirmeniz için satır satır açıklayacağım. Siz programa bakıp hangi komutun ne amaçla kullanıldığını söyleyebilmeniz gereklidir.

FILE * pFile; Burada üzerinde açıp işlem yapacağımız dosya için bir dosya işaretçisi değişkeni oluşturuyoruz. FILE tipindeki değişkenler özel değişkendir ve dosya üzerinde işlem yapmak için tanımlanması zaruridir.

int n = 0; Programın devamında kullanılmak üzere bir sayaç değişkenini burada tanımladık. n değişkeni burada n adet bayt olarak tanımlanmıştır. Devamında ne maksatla kullanıldığını daha iyi göreceksiniz.

pFile = fopen (“dosya.txt”,”rb”); Burada uygulamanın klasöründe yer alan dosya.txt adlı dosyayı açıyor ve binary olarak okuyoruz. Bunu “rb” argümanı ile belirttik. Eğer dosya.txt diye bir dosya mevcut değilse bu fonksiyon NULL değerini yani boş değeri döndürmekte. Sonrasında ise geri dönen değeri pFile adlı dosya işaretçisine aktarıyoruz.

if (pFile==NULL) perror (“Dosya Bulunmadi”); Burada eğer pFile, NULL değerine eşitse perror fonksiyonu ile bir hata mesajı yazdırıyoruz. Burada perror() fonksiyonunu ilk defa gördüğümüzden bu fonksiyonu sizlere açıklayalım. Daha öncesinde akışları anlatırken stdin, stdout ve stderr adlı akışlar olduğundan bahsetmiştik. stdin ve stdout akışlarını şimdiye kadar birden fazla kez anlattık. stderr akışının da hata mesajlarının yazıldığı bir akış olduğundan bahsetmiştik. İşte burada da hata akışına bir mesaj yazdırmak için perror() fonksiyonunu kullanmaktayız. perror() fonksiyonu içine yazacağımız karakter dizisi hata mesajı olarak yorumlanacak ve yazdırılacaktır. Bu üretilen hata mesajı platforma göre değişiklik göstermektedir.

while (fgetc(pFile) != EOF) { ++n; } Yukarıda dosya işaretçisinin NULL olup olmadığını denetledik ve NULL ise hata mesajı yazdırmasını söyledik. Eğer değilse yani dosya varsa bu sefer de fgetc() fonksiyonu ile dosyayı karakter

karakter okumaktayız. Normalde `getc()` fonksiyonu klavyeden bir karakter okumak için kullanılmaktaydı. Aynı şekilde `fgetc()` fonksiyonu bunun dosya fonksiyonu sürümü olarak karşımıza çıkmaktadır. EOF okunana kadar `n` değeri birer birer artırılmaktadır. Bu da dosyanın başından sonuna kadar kaç karakter okunduğunu `n` değişkeninde bize vermektedir.

if (feof(pFile)) Bu komut eğer EOF okunursa çalıştırılacak bir karar yapısıdır. Yukarıda zaten while döngüsü ile EOF kullanılmış ve aslında bunu kullanmaya pek gerek kalmamıştır. Eğer EOF okunursa toplam okunan baytı yazdırmak için `stdout` akışına `n` değişkeni yazdırılır. Eğer EOF okunmazsa EOF'un okunmadığı hatası verilir. Eğer EOF okunmadıysa yukarıdaki döngüden çıkma ihtimali de yoktur. Daha önce dediğim gibi zamanım olmadığından hazır örnek kullanayım dedim.

Ben daha önceki örnekte yer alan “Merhaba Dünya” yazılı `dosya.txt` dosyasını okuttum ve şöyle bir sonuç verdi. Siz değişik dosyalarla veya dosyasız da deneyip sonuçları gözlemleyebilirsiniz.



```
C:\Users\SalonPC\Documents\DosyaDeneme.exe
End of File Okundu.
Toplam okunan bayt: 13
Press any key to continue . . .
```

Bu yazıda `feof()`, `perror()` ve `fputc()` fonksiyonlarını öğrendik. Diğer dosya fonksiyonlarını öğrenmeye devam edeceğiz.

-52- Dosya Uygulamaları

Bu yazımızda artık bütün dosya okuma ve yazma fonksiyonlarına göz atacağız ve örnek uygulamalara geçeceğiz. Bu dosya okuma ve yazma fonksiyonlarının ilkel olduğunu ve herhangi bir dosya formatını içermediğini görebilirsiniz. C dilinde hazır formatlar olmadığı için bunları da kendiniz yazmanız gereklidir. O yüzden hangi tip dosya üretecekseniz veya okuyacaksanız o dosya formatlarına göre programı yazmanız gereklidir. Dosya fonksiyonlarını okuma, yazma ve kontrol fonksiyonları olarak tanımlamamız mümkündür. Daha önce feof() fonksiyonunda gördüğünüz gibi bazı fonksiyonlar okuma ve yazmanın dışında bir görev yapmaktadır. Şimdi uygulamalara geçmeden önce bütün dosya fonksiyonlarının ne iş yaptığına dair bir özet tablo hazırlayalım.

Fonksiyon	Açıklama
fopen()	Bir dosya açar veya oluşturur.
freopen()	Akıştaki dosyayı başka bir adla açar.
fclose()	Dosyayı kapatır.
fflush()	Mevcut dosya ile çıkış akışını senkronize eder.

fwide()	Geniş karakter (16-bit) giriş çıkış ile normal karakter giriş çıkışı arasında seçim yapar.
setbuf()	Dosya akışı için tampon oluşturur.
setvbuf()	Dosya akışı için tampon oluşturulur ve onun boyutu belirlenir.
fgetc()	Dosya akışından bir karakter okunur.
fread()	Dosyadan okuma yapılır.
fwrite()	Dosyaya yazma yapılır.
fgets()	Dosyadan karakter dizisi okunur.
fputs()	Dosya akışına karakter dizisi yazılır.
ftell()	Dosya konum işaretçisinin mevcut konumunu bildirir.
fgetpos()	Dosya konum işaretçisini alır.
fseek()	Dosya konum işaretçisini dosyada belirli bir noktaya atar.
fsetpos()	Dosya konum işaretçisini dosyada belirli bir noktaya atar.
rewind()	Dosya konum işaretçisini dosyanın başına atar.
remove()	Dosyayı siler.
rename()	Dosyayı yeniden adlandırır.
tmpfile()	Otomatik silinen geçici bir dosya oluşturur.
tmpnam()	Özel dosya adını geri gönderir.

Bu listeyi okuyunca biraz fazlaca olan dosya fonksiyonlarının aslında üç temel işi yaptığını görebilirsiniz. Okuma ve yazma dışında kontrol fonksiyonları yer almakta ve bu fonksiyonların büyük çoğu da dosya işaretçisinin konumunu belirlemede kullanılmaktadır. Dosya’da bir imleç gibi bir işaretçinin olduğunu ve bunun ileri ve geri giderek veri dizisi üzerinde işlem yaptığını söylemiştik. İşte dosyada nerede işlem yapacaksak bu imleci oraya götürmemiz gerekecektir.

Bundan başka dosya silme ve adlandırma fonksiyonlarını görmekteyiz. Okuma ve yazma fonksiyonları ise konsol ekranı ve klavye üzerinden işlem yaptığımız fonksiyonlara çok benzemektedir. Şimdi dosya uygulamalarına geçerek konumuzu bitirelim.

Dosya Yazma Uygulaması

Bu uygulamada klavyeden girdiğimiz bilgiler belli bir düzene göre dosyaya kaydedilmektedir. Programın yapısı oldukça basit olup klavyeden okuduğumuz değeri bir karakter dizisine ve tam sayı değişkenine kaydetmekte ve hemen sonrasında bunları dosyaya kaydetmektedir. Kaç öğrenci kaydedeceğimize göre bu döngü devam etmekte ve öğrenci sayısı da programın başında bizden istenmektedir. Şimdi programı satır satır inceleyelim ve nasıl çalıştığına bir göz atalım.

```
#include <stdio.h>

int main()
{
    char isim[50];
    int ogrenci_not, i, num;
    printf("Ogrenci Sayisini Giriniz: ");
    scanf("%d", &num);
    FILE *fptr;
    fptr = (fopen("dosya.txt", "w"));
    if(fptr == NULL)
    {
        printf("Hata!");
        system("PAUSE");
    }
    for(i = 0; i < num; ++i)
    {
```

```

        printf("Ogrenci%d\n Adi: ", i+1);
        scanf("%s", isim);
        printf("Notu Giriniz: ");
        scanf("%d", &ogrenci_not);
        fprintf(fp_ptr, "\nAd: %s \nNot=%d \n", isim,
ogrenci_not);
    }
    fclose(fp_ptr);
    system("PAUSE");
}

```

Program öncelikle bir başlangıç ve hazırlık aşamasından geçmekte ve sonrasında ana program döngüsü devam etmektedir. Ana program döngüsü bittikten sonra program da bitmektedir. Program en başka belli değişkenleri tanımlamakta ve sonrasında kullanıcıdan bazı değerleri istemektedir. Şimdi onlara bakalım.

char isim[50]; int ogrenci_not, i, num; Burada program esnasında kullanılacak değişkenler tanımlanmıştır. Programımız öğrencinin ismini ve aldığı notu dosyaya kaydeden basit bir program olduğu için bizim iki ana değerimiz isim adındaki karakter dizisi ve ogrenci_not adındaki not verisini içeren tamsayı değişkendir. Geri kalan i ve num değerleri döngüde kullanılacak sayaç değişkenleridir.

printf("Ogrenci Sayisini Giriniz: "); scanf("%d", &num); Burada kullanıcıdan öğrenci sayı istenmekte. Çünkü öğrenci sayısına göre program döngüsü yürütülecek ve çıkmak için öğrenci sayısını doğru bilmek gerekecek. Elinizde onlarca sınav kağıdı var ve bunu program başında tek tek saymak zorundasınız. Bunun daha iyisi belli bir karakteri girince çıkmamızı sağlayan bir program da yapabiliriz. Daha iyisi EOF karakterini girerek bunu yapabiliriz. **EOF karakteri Windows sistemlerde Ctrl + Z, Unix/Mac/Linux sistemlerde Ctrl + D ile girilir.** ç

fptr = (fopen(“dosya.txt”, “w”)) Burada dosya.txt yazma modunda açılmakta. Eğer açılmazsa bir sonraki karar yapısında hata mesajı verilmekte.


for(i = 0; i < num; ++i) Okunan öğrenci sayısı kadar döngü devam ettirilmekte.

Döngünün içine baktığımızda klavyeden (stdin) alınan veriyi dosya üzerine yazdığını görmekteyiz. Bu dosyanın formatı tamamen fprintf ile formatlanmakla meydana gelmektedir. Dosya formatlarının önemine dikkat ediniz. Örneğin bir .xyz uzantılı dosya x y z (yeni satır) x y z diye veri alırken bir .csv uzantılı dosya x,y,z (yeni satır) x,y,z şeklinde veri almaktadır. Bu araya virgül koymak, boşluk eklemek veya nereye hangi parametreyi koyacağınız sizin sorumluluğunuzdadır.

fprintf(fptr,”\nAd: %s \nNot=%d \n”, isim, ogrenci_not); Burada dosyaya formatlanmış bir biçimde bilgiler yazılmaktadır. Siz okuyacakken aynı bu şekilde formatlı olmasına dikkat ediniz. Burada basit bir metin belgesi olsa da normalde dosyaların formatlarına göre veri yazılır ve formatlarına göre veri okunur. Programı çalıştırdığımızda ekranda ve dosyada görebileceğimiz bir çıktı şu şekildedir.



```
C:\Users\SalonPC\Documents\DosyaDeneme.exe
Ogrenci Sayisini Giriniz: 3
Ogrenci1
  Adi: YilanSerdar
Notu Giriniz: 50
Ogrenci2
  Adi: DukeNukem
Notu Giriniz: 60
Ogrenci3
  Adi: ayca_22
Notu Giriniz: 60
Press any key to continue . . . _
```


 dosya - Notepad

File Edit Format View Help

Ad: YilanSerdar
Not=50

Ad: DukeNukem
Not=60

Ad: ayca_22
Not=60

Dosya okuma

Bu örnek en basit dosya okuma yöntemini size göstermektedir. `fscanf()` gibi formatlı dosya okumak için dosya formatını bilmeniz gerekse de dosyayı ham şekilde yani karakter karakter okuyup ekrana yazdırmanız için dosyanın formatını bilmenize gerek yoktur. Bu sayede sadece karakter dizilerinden ibaret bir veri elde edersiniz. Elbette bu veriyi değerlendirmek biraz daha zor olacaktır. Fakat burada biz okuduğumuz dosyayı ekranda görmekten öte bir şey istemediğimizden `fgetc()` fonksiyonunu kullanmayı yeterli görüyoruz. Şimdi programa göz atalım.

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

```
    FILE *fp1;
    char c;
```

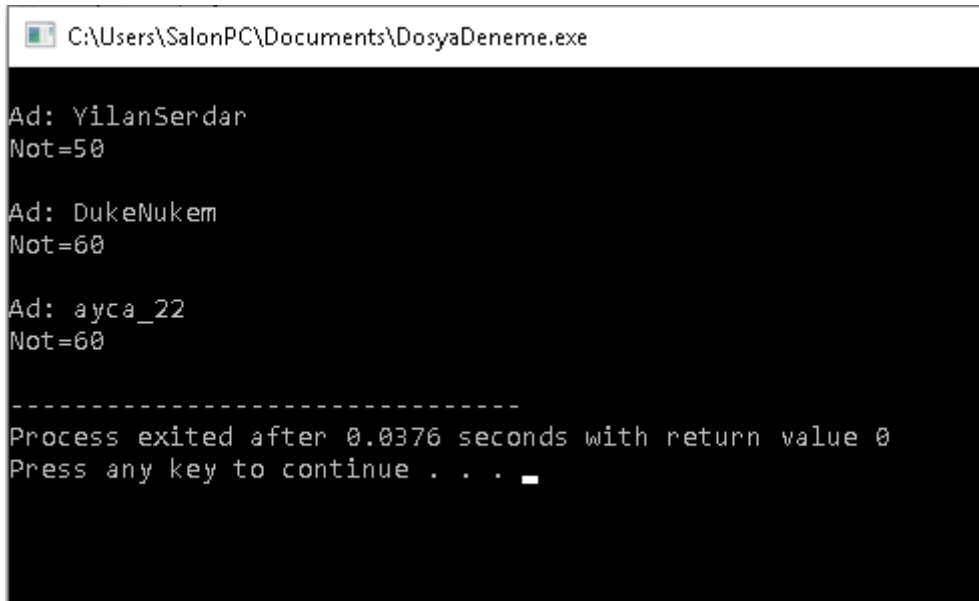
```

fp1= fopen ("dosya.txt", "r");

while(1)
{
    c = fgetc(fp1);
    if(c==EOF)
        break;
    else
        printf("%c", c);
}
fclose(fp1);
return 0;
}

```

Gördüğünüz gibi oldukça basit bir programla karşı karşıyayız. Her zaman olduğu gibi FILE tipinde bir işaretçi oluşturup bir dosyayı “r” yani okuma modunda açtıktan sonra bir döngü içerisinde fgetc() fonksiyonu ile dosya işaretçisinin işaret ettiği karakteri okuyor ve bunu c karakterine aktarıyoruz. Dosya işaretçisi her fgetc() fonksiyonundan sonra bir artırılır. Yani okuma yaptıktan sonra bir tekrar çalıştırırsan bir sonraki karakter verisini okuyacaktır. Böyle böyle sırayla bütün dosyayı baştan itibaren okumaya başlar. Bunu sonsuza kadar böyle yapmak mümkün değildir. Bir yerde dosya bitecek ve fgetc() fonksiyonu EOF değerini okuyacaktır. Biz de EOF okununca döngüden çıkmayı istiyoruz. O yüzden **if(c==EOF) break;** komutu kullanılmıştır. Eğer c değişkeninde EOF değeri varsa döngüden break komutu ile çıkılmaktadır. Program çok zor olmadığı için çok ayrıntılı açıklama gereği görmedim. Ben yukarıdaki örnekten arta kalan dosyayı kullandığımda şöyle bir ekran çıktısı aldım.



```
C:\Users\SalonPC\Documents\DosyaDeneme.exe

Ad: YilanSerdar
Not=50

Ad: DukeNukem
Not=60

Ad: ayca_22
Not=60

-----
Process exited after 0.03376 seconds with return value 0
Press any key to continue . . . _
```

Şimdilik temel C programlama konusu için bu örnekleri yeterli görmekteyim. Daha ileri seviye dosya konularını başka bir eğitimde sizlere açıklayacağız. Eğer ihtiyaç duyarsanız siz de kendi araştırmalarınızla bu yönde örnekleri bulabilirsiniz. Buraya kadar konuyu anladığınızı düşünüp yeni konuya geçiyorum.

-53- Ön İşlemci Komutları (Preprocessor)

Temel C programlama konumuzun en sonunda size ön işlemci komutlarından bahsedeceğiz. Ön işlemci komutları ile normal C komutlarını biraz

ayırmamızda fayda vardır. Ön işlemci komutları derleyiciye göre değişiklik gösterebilir ve bu özel komutları derleyicilerin kılavuzlarında okuyabilirsiniz. Fakat C dilinin olmazsa olmazı bazı ön işlemci komutları da vardır. Biz konumuzda standart dışı komutlara yer vermeyeceğiz. Kullandığınız platform ve derleyiciye göre değişen bu komutları sizin öğrenip kullanmanız gereklidir.

Ön işlemci komutlarını şimdiye kadar bütün programlarımızda kullandık. Örneğin `#include` ile dosyaya bir kütüphane başlık dosyası eklerken kullandığımız `#include` komutu bir ön işlemci komutuydu. Bu komutlar doğrudan derleyiciye hitap etmekte ve program komutu değil derleyici komutu olarak çalışmaktadır. O yüzden derleyiciler program kodu derlenmeden önce bu komutları çalıştırmaktadır. Ön işlemci komutları sadece bir kütüphane dosyasını eklemek için değil sabit tanımlamak, makro tanımlamak, şartlı tanımlar ve derleyici yönergeleri için kullanılmaktadır. Şimdi belli başlı ön işlemci komutlarına göz atalım.

#include yönergesi

Biz derslerin en başında “Merhaba Dünya!” uygulamasında bile `#include` yönergelerini kullandık. C dili oldukça sade yapıya sahiptir. Bu sade yapıdan dolayı `#include` ile kütüphane dosyalarını eklemezsek hemen hemen hiçbir şey yapamayız. En temel işlerden biri olan giriş ve çıkış fonksiyonları bile ayrı bir kütüphanede yer almakta ve `#include` ile programa dahil edilmektedir. Eğer böyle olmasaydı dilin içine ve standarta eklemek gerekecekti ve bütün platformlara uyumlu olmayacaktı. Örneğin AVR programlarken kimse `printf()` veya `scanf()` gibi fonksiyonları kullanma gereği duymaz. O halde bunlar dilin kendi yapısı olamayacağı için ayrı kütüphane dosyaları olarak bize gelmiştir. Bu kütüphane dosyaları harici kütüphaneler ve standart kütüphaneler olarak ikiye ayrılmaktadır. Standart kütüphanelerde belli başlı işleri yürüten belli

fonksiyonlar yer almaktadır. Biz `#include` yönergesini şu iki şekilde kullanabiliriz.

```
#include <dosyaadi>
#include "dosyadi"
```

Burada birinde `<` ve `>` kullanılırken diğerinde çift tırnak arasına (” “) yazılmıştır. Bu ikisi arasında büyük fark vardır ve karıştırmanız halinde dosya bulunamayabilir. Kural olarak `<` ve `>` işaretleri yeri bilinen standart kütüphane dosyalarını eklemek için kullanılırken çift tırnak (”) ise yeri bilinmeyen kullanıcı dosyalarını eklemek için kullanılır. `<` ve `>` arasına yazdığınız dosyanın yolu derleyicide belirlenmiştir ve derleyici bunun nerede olduğunu bilmektedir. Fakat iki tırnak arasına dosyanın yolu belirtilmezse bu dosya bulunamaz.

#define yönergesi

`#define` yönergesi en başta sabit sembollerini tanımlamak için kullanılmaktadır. Burada sabitleri tanımlamanın oldukça esnek olduğunu söylememiz gerekir. Aslına bakarsanız `#define` yönergesi ile derleyici bir yerde bir sembolü gördüğü zaman onu keser ve bizim yazdığımızı oraya yapıştırır. Böylelikle kodu baştan sona düzenlemiş olur. Şimdi `#define` yönergesinin söz dizimini size gösterelim.

```
#define sembol yerlestirilecek_deger
```

Buradan pek bir şey anlamadıysanız bunun nasıl kullanılacağına dair bir örnek verelim. `#define` yönergesini biz sabit değişkenler yerine kullanabiliriz. Üstelik hafızadan da tasarruf edeceğimiz için gömülü sistemlerin vazgeçilmezlerinden biridir. `const int pi = 3.14` demek yerine `#define PI 3.14` dediğimizde derleyici `PI` yazan yeri gördüğü zaman `3.14` ifadesini oraya

yapıştıracaktır. Böylelikle değişken oluşturup bunu da hafızaya kaydedip sürekli sürekli hafızadan okumak zorunda kalınmaz. Değişmeyen bir değer olduğu için sabit olarak kullanmak daha performanslıdır.

```
#define PI 3.14  
const float PI = 3.14F;
```

Biz float diye değişken tanımlayarak 32-bitlik yani 4 baytlık bir yer işgal ettiğimiz gibi mikroişlemci her PI üzerinde işlem yaparken bu PI değişkeninin olduğu adresi okuyarak ekstradan komut harcamak zorunda kalacaktır. Oysa ki biz sabit olarak tanımlasak bu 3.14 değeri komutun içerisinde yer alacak ve okunur okunmaz işletilecektir. Fakat biz programın her yerine 3.14 yazmak yerine kolayca PI yazmak istiyorsak `#define PI 3.14` yazmamız gerekecektir. Üstelik `#define` yönergesi sadece sabitler için kullanılmaz. Programın programlama aşamasında bazı ayarlarını değiştirmek için de kullanabiliriz. Örneğin lambaları yakıp söndüren bir programımız var ve bu aşamada bir bekleme süresi tahsis etmemiz gerekiyor. Bu durumda ya değişken tanımlayacağız ve hafızadan harcayacağız ya da `#define` ile tanım yapacağız.

```
#define bekleme 100
```

```
// komutlar  
// komutlar  
delay(bekleme);  
// komutlar  
// komutlar  
delay(bekleme);  
// komutlar  
// komutlar  
delay(bekleme);
```

Gördüğünüz gibi bekleme sabitini pek çok yerde kullanıyoruz. Biz bu sembolü kullanmak yerine doğrudan sabit değer olan 100 değerini yazsaydık değiştirmek istediğimiz zaman bütün kodu taramamız gerecekti. Burada `#define` yönergesinde yer alan değeri değiştirmekle bütün program üzerinde değişikliği yapmış oluyoruz. Üstelik bunun için de değişken kullanmamıza gerek kalmıyor.

`#define` sadece bununla sınırlı kalmaz. Yukarıda anlattığımız sabit değişkenleri kullanmanın daha ekonomik yollarından biriydi. Biz `#define` ile sadece değer değil bir kod bloku veya bir komutu hatta bir anahtar kelimeyi bile değiştirebiliriz. Aşağıda bunun birkaç örneğini görebiliriz.

```
#define LEDYAK digitalWrite(16,HIGH)
#define if eger
```

```
LEDYAK;
```

```
eger (i < 10)
komutlar;
```

Bu sayede kendi programlama dilimizi yapmaya kadar giden bir yolun olduğunu görsek de bunlar standart dışı olacağından biz bildiğimiz C programlama dilinden şaşmayalım.

`#define` yönergesinin en önemli başka bir kullanım alanı ise makrolardır. Makrolar aynı inline (satır arası) fonksiyonlar gibi görev yapmaktadır. Derleyici burada kopyala yapıştır yapmaktan öte gidemediği için pratik olsa da her

zaman ekonomik olduğunu söyleyemeyiz. Şimdi örnek bir makroya göz atalım.

```
#define DIKDORTGEN_ALANI( x, y ) ( ( x ) * ( y ) )
```

Burada dikdörtgenin alanını hesaplayan bir makro görmekteyiz. Bunu aynı fonksiyonu nasıl kullanıyorsak öyle kullanabiliriz. Örnek bir kod şöyle olabilir.

```
alan = DIKDORTGEN_ALANI( 5 , 10 );
```

#define ile tanımlanan sabitler #undef ile de kaldırılabilir. Bu makrolar kütüphane ve derleyici dosyalarında sıkça kullanılmaktadır. Bazen bu makroların iç yapısını ve ne işe yaradığını bilmeniz gerektiğinden dolayı karşınıza çıktığı zaman iyi çözümlemeniz gerekecektir.

Şartlı Derleme

Biz her zaman aynı programı aynı amaç doğrultusunda yazmayız. Örneğin yazdığımız bir program üç farklı amaç için kullanılıp sadece birkaç satır kod fark içeriyorsa bazı komutların bazı durumdaki derlenmesini bazen de bu komutların programdan çıkarılmasını isteyebiliriz. Bu durumda şartlı derleme işlemi adı verilen bir işlem kullanılır. Biz burada tanımladığımız değerlere göre bazı program kodlarını derlemeye alabilir veya bunları çıkarabiliriz. Bir de eğer bazı sabitleri tanımlamadıysak kütüphanede bu sabitleri standart olarak tanımlanmasını isteyebiliriz. İndirdiğiniz kütüphaneleri incelediğinizde bu tarz kullanımların fazlaca olduğunu göreceksiniz.

#if yönergesi

Bu yönerge if komutu gibi çalışmaktadır. Fakat derleyici bazında yapılan bu işlemle if bloku içerisinde yer alan program kodu programa dahil edilmektedir. Eğer if şartı sağlanmazsa bu kod bloku programdan tamamen çıkartılmakta

ve derleme esnasında programa dahil edilmemektedir. #if yönergesi her zaman #endif yönergesi ile bitmek zorundadır.

```
#if DEBUG
    printf("Hata Ayıklama Surumu");
#endif
```

Kütüphanede böyle bir kodu gördüğümüzde burada DEBUG adında yer alan sabitin değerini bilmemiz gerektiğini anlamamız gereklidir. Burada DEBUG değerinin 1 veya 0 olmasına göre bu komut programa dahil edilecektir. Bu durumda başlık dosyasında kütüphane ayarlarının olduğu kısma bakmamız ve bunu değiştirmemiz gereklidir. **Pek çok C kütüphanesinde .h başlık dosyasındaki ayarları bizim değiştirmemiz gerekebilir.** Burada da #define DEBUG 0 ya da #define DEBUG 1 yaparak bu kodun programa dahil olup olmamasını belirleyebiliriz.

#ifdef yönergesi

Bu yönerge “Eğer tanımlanmışsa” anlamı taşımaktadır. Yani biz bir sembolü #define ile tanımladıysak şart bloku içindeki program komutları programa dahil edilecektir ve işletilecektir. Biz bir değer tanımladıysak kullanıcıya yönelik bir program işletilebilir. Örnek programda bunu görmekteyiz.

```
#include <stdio.h>

#define YAS 10

int main()
{
    #ifdef YAS
```

```
printf("Kullanici %i Yasindadir\n", YAS);  
#endif  
  
return 0;  
}
```

Burada eğer kullanıcı YAS diye bir tanım yaparsa program kullanıcının yaşını da ekrana yazdırmaktadır. Eğer böyle bir tanım yoksa bu komutun işletilmesine ve hatta programa dahil edilmesine hiç gerek yoktur.

#ifndef yönergesi

Genelde #ifndef yönergesi tanımlanmamış bir değere standart bir değer atamak için kullanılır. Bir kütüphanede tanımlanmamış bir sabit olursa ve bu sabit pek çok yerde kullanılıyorsa kütüphane sırf değer tanımlanmadığı yüzünden çalışmayacaktır. Bunu kullanıcı unutsa da kütüphaneyi yazan programcı bunu hesaba katarak standart bir değer belirler.

```
#ifndef KULLANICI_ADI  
#define KULLANICI_ADI Player  
#endif
```

Burada oyunlardan aşına olduğunuz bir durum karşımıza çıkmakta. Eğer kullanıcı adını kullanıcı belirlemezse oyun kullanıcıya “Player” diyecek ve sunucularda adını böyle yazacaktır.

#ifndef yönergesinin en önemli kullanım alanlarından biri de kütüphane başlık dosyalarının tekrarlanmasını önlemektir. İncelediğiniz kütüphanelerde her zaman başlık dosyalarının şu şekilde başladığını görürsünüz.

```
#ifndef MAIN_H_
```

```
#define MAIN_H_
```

```
// Burada bütün başlık dosyası kodu var
```

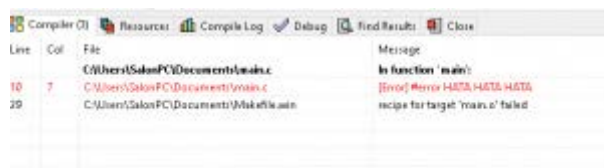
```
#endif
```

Siz de başlık dosyası yazarken bu alışkanlığı edinirseniz iyi olur.

Bu `#if`, `#ifdef` ve `#ifndef` yönergeleriyle beraber **`#elif`** (else if) ve **`#else`** yönergeleri de kullanılmaktadır. Bu yönergeler aynı normal if şart komutlarında olduğu gibi daha karmaşık şart işlemlerini yapmamızı sağlamaktadır.

`#warning` ve `#error` yönergeleri

Biz program yazarken derleyiciye hata veya uyarı mesajı vermek istiyorsak bu yönergeleri kullanırız. Bazı durumlarda bizim kodumuzu kullanan programlayıcının ne yapacağını kestiremeyiz. Bu durumda böyle yaptığı zaman hata veya uyarı mesajı vermek zorunda kalabiliriz. Elbette bu kütüphane yazarları ilgilendiren bir durumdur. Nadiren de olsa bazen kendi yazdığımız kütüphanelerde de bu hatalardan korunmak için kendimize mesaj verme ihtiyacı duyabiliriz. `#warning` ile yazdığımız program derlense de yazdığımız hata mesajı derleyici konsolunda programcıya gösterilecektir. `#error` ise program derlemesini durdurur. Örneğin benim program içerisine yazdığım `#error HATA HATA HATA` mesajı derleyici konsolunda şöyle göründü.



warning ve #error yönergelerinin söz dizimi şu şekildedir.

#warning Uyarı mesajı

#error Hata Mesajı

Buraya kadar önışlemciler hakkında öğrendiğiniz bilgiler sizin için yeterli olacaktır. Bunları yazmadan önce kütüphaneleri kullandığınız zaman kütüphaneler içerisinde göreceksiniz ve anlayıp yorumlayabilmeniz gerekecek. Siz nasıl kod yazılacağını öğrenmek istiyorsanız kitapları değil yazılmış hazır kodları ve kütüphaneleri incelemeniz gereklidir. Kitaplar ise bu kodları nasıl anlayacağınızı size bildirir.

Son Söz

Temel C programlama derslerinin sonuna bir sonsöz ekleme gereksinimi duyuyoruz. Çoğu programlama kitaplarına baktığımızda konular tamamen anlatılsa da bu konular bittikten sonra ne yapılacağı, öğrencinin ne öğrenmesi gerektiği veya kendini nasıl geliştireceği konusunda bir bilgi yer almamakta. Aslında bunu öğrencinin kendisine bırakmaktalar ve öğrenci bunu çalışarak kendisi öğrenmekte. Fakat Türkiye’de öğrenciler ne yapacağı konusunda pek

bir fikre sahip olamıyor ve bu konuda yol gösterilmeye muhtaç halde oluyorlar. Ben de şahsen neyi nasıl öğrenmem gerektiğini, kendimi nasıl geliştirebileceğimi çok daha sonraları öğrenebildim. İlk aldığım programlama kitabında bütün örneklerin baştan sona konsol üzerinde yer aldığını ve konsoldan nasıl çıkılacağı, nasıl uygulama yazılacağı, bu dilin nerelerde kullanılacağı konusunda bir bilgiye rastlamadığımdan açıkcası programcılık bana aşırı derecede zor göründü. Konsol uygulamalarını ben de anlatsam da programlama dilinin temelini anlatmanın ancak böyle mümkün olmasından dolayı tercih ettiğim bir yöntem oldu. Bu seriden sonra “Gömülü C” ve “İleri C” gibi konularda uygulamalara ve günümüzde kullanılan teknolojilere yer verip uygulamalar üzerinden C dilini anlatmaya devam edeceğim. Burada yazdıklarım sadece yolun başıdır.

Ben bütün bu dersleri öğrendim, artık ne yapmalıyım? diye soracak seviyedeyseniz öncelikle size ana kaynaklardan haber vermem gerekir. Benim yazdığım ancak “ikincil kaynak” statüsündedir ve birincil ve diğer ikincil kaynaklara sizi muhtaç bırakmaktadır. Açıkcası bu konuda hiçbir kitap size meselenin tamamını öğretecek seviyede değildir. Bu yüzden pek çok kaynağa müracaat etmeniz kendinizi geliştirme açısından en isabetlisi olur. Dikkat edin, burayı bırakıp gidip Youtube videosu izleyin demiyorum. Çünkü bir Youtube videosundan öğrenebileceklerinizin kat kat ilerisini buradan öğrenebiliyorsunuz. Fakat bizim de anlatamadığımız konular mevcut. O yüzden aşağıda vereceğim kaynaklara muhakkak bakınız.

- **C How to Program – Deitel**
- **The C Programming Language – Kernighan & Ritchie**
- **Expert C Programming – Van der Linden**
- **21st Century C – Klemens**
- **Advanced C – Peter D. Hipson**

- **Algorithms in C – Robert Sedgewick**
- **Data Structures Using C – Reema Thereja**
- **The GNU C Library Reference Manual**
- **Modern C – Jens Gustedt**
- **The Standard C Library – P.J. Plaucher**
- **C Standard**

Yukarıda standart, kütüphane kılavuzu gibi birincil kaynakların yanında ikincil kaynak statüsünde olan kitapları da görebilirsiniz. Bu kaynakları bir kenara bırakıp dördüncül kaynak statüsünde bile olan Youtube videolarını seyretmeniz sizi ileriye değil geriye götürecektir.

Ne yapabilirim? sorusuna gelecek olursak günümüzde C dilinin en popüler olduğu alan gömülü sistemlerdir. Eğer C bilginiz iyiye hatta bu derslerde yazanları bile öğrendiyseniz gömülü sistemlerde rahatlıkla program yazabilir ve iş yapabilirsiniz. Elbette işin donanım boyutunu görmezden gelirsek durum böyledir. Sadece yazılım bilgisi ile ancak Arduino gibi platformlarda kısmen çalışabilmeniz mümkündür. Ama ben donanım öğrenmenin çok da zor olduğunu düşünmüyorum.

Bundan başka aşağıdaki awesome listelerinde pek çok yazılım alanıyla alakalı C kütüphanelerini görmeniz mümkün. Bunları inceleyerek neler yapabileceğiniz hakkında fikir edinebilirsiniz.

<https://github.com/IMCG/awesome-c>

<https://github.com/aleksandar-todorovic/awesome-c>

<https://github.com/kozross/awesome-ch>

<https://github.com/uhub/awesome-c>

C dilini öğrenmenin en büyük avantajlarından biri de artık günümüz programlama dillerinin atasını öğrendiğiniz için diğer diller sizin gözünüzde çocuk oyuncağı gibi görünecektir. Python örneklerine göz attığınızda çok çok daha kolay bir dille karşı karşıya kalırsınız. Ya da C# veya Java dilinin C dilinden ciddi derecede esinlendiğini görürsünüz. C bilenler için en uygun dil ise C++ dilidir. C++ dilinin bu kadar popüler olmasının en temel sebebi C dili üzerine bina edilmesidir. Siz C bilginizi unutmadan sadece üzerine yeni konuları öğrenerek C++ dilini öğrenmiş oluyorsunuz. C++ dilini C dilinin modifiyeli hali olarak tanımlayabiliriz. C++ dili günümüzde daha yaygın bir kullanım alanına sahiptir. Çoğu oyun motoru C++ dili üzerine kurulduğu gibi Microsoft'un Visual C++ desteği halen devam etmektedir. C++ üzerine kurulan kütüphaneler, yazılım çatıları ve platformlar oldukça fazladır. Ülkemizde C++'yı pek öğrenmek isteyen çıkmasa da C++ dili diğer popüler programlama dillerinden daha sağlam temeller üzerinde gibi görülmekte.

Ben bu dersleri kitap yazar gibi yazsam da bol bol örnek yaptırmaktan ziyade kısa sürede mantığı kavramanız için yoğunlaştım. Çünkü asıl mesele programlama dilini anlayabilmektir. İstersem onlarca örnek de ilave edebilirdim ama yazmaya bile zor zaman buluyorum. O yüzden uygulama yönünde kendiniz biraz çaba sarf etmeli ve bulduğunuz uygulamaları denemeli ve kendi uygulamalarınızı yazmalısınız. Kütüphaneleri ve kaynak kodları inceleyip çözmeniz çok önemlidir. Sadece siz bir şeyler yapmaya çalışmayın. Yapılmış işleri de alıp inceleyin ve kendiniz de yapabilecek noktaya gelin. Nasıl kod yazılacağını, nerede ne yapılacağını bazen de başkasının eline bakarak öğrenmeniz gerekli. Başkasının eline bakma yöntemi bu meslekte hazır yazılmış kodları incelemek demektir. Bunun için internette çeşitli yerlerde kod bulabilseniz de Github sitesi sizin birinci kaynağınız olmalıdır.

