Derleme: TD Software -Yazar: Çağatay ÇEBİ

C Programlama Dersleri

Bu yazıda öğreneceksiniz:

- Bilgisayar ve Programlama nedir?
- Program yazmak için gerekli araçlar ve temin edebileceğiniz adresler
- Algoritma geliştirme ve bunun üzerine basit bir örnek
- printf fonksiyonunun kullanımı ve bununla ilgili örnekler

Merhaba;

Sanırım, C ve C++ adını bilgisayarla az çok haşır neşir olan herkes en az bir kez duymuştur. Sizde bu isimleri duyanlardansanız ve nedir, ne değildir, nasıl kullanılır gibi birçok soruya yanıt arıyorsanız, doğru yerdesiniz. Çünkü bu yazıyla başlayarak C ve C++ ile programlamaya gireceğiz. Önce C ile yolumuza koyulup, belli bir olgunluğa ulaştıktan sonra C++ ile devam edeceğiz.

Okuyucularımızın genelini düşünerek, konuyu en temelden almayı daha doğru buldum. Yani hedefimiz, programlamayı hiç bilmeyen bir insanın burada okuduklarıyla belli bir yerlere ulaşması. İleri derece de olanlarsa sıkılmamak için biraz beklemeli. Lafı fazla uzatmadan başlayalım.

Bilgisayar ve Programlama nedir?

Bilgisayar çok basit düşündüğümüzde üç ana görevi yerine getiren bir makinedir. Girilen bilgiyi alır (INPUT), işler (PROCESSING) ve bu işlenmiş veriden bir sonuç (OUTPUT) çıkarır. Bilgisayar, sadece donanım olarak çalışmaz. Çünkü yazılım olmadan, donanım ne yapacağını bilemez. Bilgisayar donanımına ne yapacağını söyleyecek bir komutlar dizisi gerekir. Yapacağı görevleri, ona anlatan komutlara program diyebiliriz. Yani donanıma “sen şunu yap, sonra bulduğun sonucu şöyle şuraya ekle” gibisinden işler yaptırmak programın veya bir başka deyişle yazılımın işidir. Bir programcı olarak bundan fazlasını bilmek elbette ki avantajdır. Ama bilgisayarın bütün özelliklerini bilmeniz gerekmez. Yani yazacağınız bir program için o bilgisayarın özelliklerini bilmeseniz de olur.

Bilgisayarın anladığı tek dil, Makine Dilidir. Bu 16’lık (Hexadecimal) sistemden oluşan bir programlama tipidir. Makine dilini anlamak çok zordur ve bu dili kullanmak için o bilgisayarın donanım özelliklerini mutlaka bilmeniz gerekir. C de ekrana yazı yazmanızı sağlayan “printf();” gibi çok basit bir fonksiyon, makine dilinde 1A BB 0D BC D5 FF C2 F7... gibi çok daha karmaşık ve hiçbir anlam ifade etmeyen bir hâle dönüşür. Makine dili programlama dilleri arasında en alt seviyedir.

Makine dilinden sonra Assembler Dili gelir. Makine dilini kullanmanın zorluğu ve karmaşası üzerine geliştirilen Assembler, daha basit bir yapıdadır. Ama yine de C ile mukayese ederseniz çok daha zordur ve kullandığınız bilgisayarın donanımına dair hâlen bilgiye gereksinim duyarsınız. Assembler aşağıdaki gibi karmaşık bir yapıdadır.

```
SEGMENT COM WORD PUBLIC 'CODE'
ASSUME CS : COMDS : COM
      ORG 100H
ENTRY: MOV DVX,OFFSET MSG
       MOV AH,g
           .
           .
           .
```

Şuan bunu anlamaya çalışıp, hiç zamanınızı harcamayın. Çünkü öğreneceğimiz dil C, işlerimizi ve dolayısıyla hayatımızı çok daha kolaylaştırmaktadır. C, orta seviye bir programlama dilidir. Bunun anlamı, hem yazması kolay, hemde üst seviye dillere göre daha çok erişim hakkınızın olduğudur. Üst seviye programlama dilleri ise BASIC, PASCAL, gibi dillerdir. Üst seviye dillerde, yazması göreceli olarak daha kolay olsa da C ile yapabileceklerimiz daha çoktur.

Program yazmak için ne gerekir?

Program yazabilmek için hiçbir şeye ihtiyacınız yoktur. Program yazmak için Windows'un not defterini veya Linux'da Gedit, Kwrite gibi bir program bile kullanabilirsiniz. Önemli olan yazılan programın derlenmesidir. Derlemeye "compile" ve derleme işini yapan derleyiciyeyse "compiler" denir. C için internet üzerinden birçok Compiler bulabilirsiniz. Ben, program uygulamalarını GCC üzerinden yapacağım. Aynı şekilde bu derleyiciyi kurmanızı tavsiye ederim. GCC gelmiş geçmiş en iyi derleyicilerden biri olduğu gibi, özgür bir yazılımdır! Richard Stallman tarafından açık kaynak koduyla sunulmuştur ve arzu ettiğiniz takdirde, sonuna kadar değişiklik yapma imkanınız vardır.

Şayet Windows işletim sisteminiz varsa GCC'yi kurmanız biraz sıkıntılı olabilir. Basit bir Google araştırmasıyla, Bloodshed Dev-C++ adında bir program buldum. GCC üzerine kurulmuş bir yapısı varmış. GCC ile uğraşmak istemeyen Windows kullanıcıları, bu programı deneyebilir.

Algoritma Geliştirmek

C dilini ve komutlarını öğrenmek, programlamaya başlamak için şarttır ama algoritma oluşturamadığımız sürece bir program oluşturmazsınız. Algoritma, mantıktır. Yani neyi, nasıl yapacağınızı belirtir. Algoritma türetmek için geliştirilmiş bir metot yok. Her program için o metodu sizin bulmanız gerekiyor. Ama hiç merak etmeyin, yazdığınız program sayısı arttıkça, algoritma kurmanız daha kolaylaşır.

Algoritma, programlamanın bel kemiğidir. C dilinde kullanılan komutlar, BASIC veya FORTRAN gibi başka dillerde işe yaramaz. Fakat programlama mantığını bir kere oturttursanız, C komutlarının yerine pekâlâ başka dillere ait komutları da öğrenebilir ve büyük bir zorluk çekmeden diğer dillerde de program yazabilirsiniz.

Basit bir örnek üzerinden düşünelim. Bir markete gittiniz, kasada ki görevliye aldığınız ürünü gösterdiniz, parayı uzattınız, paranın üstünü aldınız. Günlük hayatta gayet normal olan bu durumu biraz değiştirelim. Karşınızda insan değil, elektronik bir kasiyer olsun. Ona göre bir algoritma geliştirirsek,

1-)*Ürüne bak;*
2-)*Ürün Fiyatını bul;*
3-)*Parayı al;*
4-)*Alınan paradan ürün fiyatını çıkar;*
5-)*Kalan parayı ver.*

İnsan zekasının otomatik hâle getirdiği eylemleri, ne yazık ki bilgisayar bilmez ve ona biz öğretmek zorundayız. Öğretirken de hata yapma hakkımız yoktur, çünkü yanlış öğreti yanlış programlamayla sonuçlanır.

C Programlama Dili

Temel Giriş/Çıkış İşlemleri (BASIC I/O):

C ile ilgili olarak bu ve önümüzdeki yazılarda birçok komut/fonksiyon göreceğiz. Ama hep kullanacağımız ve ilk öğrenmemiz gerekenler temel giriş çıkış fonksiyonlarıdır. C de klavyeden bir değer alabilmek için scanf(); fonksiyonunu kullanırız. Ekrana herhangi bir şey yazdırmak içinse printf(); fonksiyonu kullanılır.

Bir örnekle görelim;

```
#include<stdio.h>
int main( void )
{
        printf("Hello World");
}
```

Eğer bunu derleyicinizde yazıp derlerseniz ve sonrasında çalıştırırsanız ekrana Hello World yazılacaktır. #include<stdio.h>, standart giriş çıkış başlık dosyasını, programa dahil et gibi bir anlam taşır. C'de (ve hemen hemen bütün diğer programlama dillerinde) bir kütüphaneyi dahil etmek son derece rutin bir iştir. Aksi halde giriş-çıkış fonksiyonlarını dahi her seferinde bizim baştan tanımlamamız gerekirdi.

main( ), bir programdaki ana fonksyiondur. Ondan sonra gelen ayraç standarttır. Bir bloğu temsil eder. İki ayraç işareti arasındaki alan main fonksiyonuna ait bir bloğu oluşturur. printf ise yazdığımız metini, ekrana bastırmaya yarayan, standart bir fonksiyonudur. Çift tırnak işaretleri içersine yazdığınız herşey printf sayesinde ekrana basılır.

Dikkat ettiyseniz, her satır sonuna noktalı virgül koyduk. Aslında her satır değil, her komutan sonra noktalı virgül koyduğumuzu söylemek daha doğru olacak. Çünkü noktalı virgül C dilinde komut ayracı anlamına gelir.

Şimdi yukarıda yazdığımız basit programı, biraz daha geliştirelim:

```
#include<stdio.h>
int main( void )
{
        printf("Hello World\n");
        printf("Merhaba Dünya");
        return 0;
}
```

Birkaç yeni satır görüyorsunuz. Sırayla ne olduklarını açıklayalım. Az evvel yazdığımız "Hello World" yazısının sonuna "\n" ekledik. "\n" bir alt satıra geç anlamına geliyor. Eğer "\n" yazmazsak, ekranda "*Hello WorldMerhaba Dünya*" şeklinde bir yazı çıkar. "\n" kullanırsak, "*Hello World*" yazıldıktan sonra, bir alt satıra geçilir ve ikinci satırda "*Merhaba Dünya*" yazdırılır. En altta "*return 0;*" adında yeni bir komut fark etmişsinizdir. Bunu eklemezseniz, program yine çalışır; ancak uyarı verir. Çünkü main fonksiyonu, geriye bir tam sayının dönmesini beklemektedir. Yazmış olduğumuz return ifadesiyle bu uyarılardan kurtulabilirsiniz. Detayına girmek için henüz erken, return konusuna ileride değineceğiz.

Yukarıda ki programın aynısını şöyle de yazabilirdik:

```
#include<stdio.h>
int main( void )
{
        printf("Hello World");
        printf("\nMerhaba Dünya");
```

```
        return 0;
}
```

Bir önce ve şimdi yazdığımız programların ekran çıktısı aynıdır. Bu örnekle anlatmak istediğim, printf() fonksiyonunda '\n' konulan yerden sonrasının bir alt satıra düşeceğidir.

```
#include<stdio.h>
int main( void )
{
        printf("Hello World\nMerhaba Dünya");
        return 0;
}
```

Gördüğünüz gibi tek bir printf(); kullanarak aynı işlemi yaptırdık.

Varsayalım, ekrana çok uzun bir cümle yazmamız gerekti. Örneğin;

```
#include<stdio.h>
int main( void )
{
        printf("Benim adım Çağatay ÇEBİ ve Yazılım Mühendisiyim.\n");
        return 0;
}
```

Bu yazdığımız program hata vermemesine karşın, çalışma verimini azaltır. Çünkü yazacaklarınız editör penceresine sığmazsa, yazılanı okumak daha zahmetli olur. Önemsiz bir detay gibi gelebilir, ama kod yazma verimini ciddi oranda düşüreceğinden emin olabilirsiniz.

Bu programı aşağıdaki gibi yazmamız daha uygundur:

```
#include<stdio.h>
int main( void )
{
        printf("Benim adım "
                "Çağatay ÇEBİ"
                "ve Yazılım Mühendisiyim.\n");
        return 0;
}
```

Tek bir printf(); fonksiyonu kullanılmıştır. Ancak alt alta yazarak, metini tek seferde görülebilir hâle getirdik. Programı derleyip çalıştırırsanız, alt alta üç satır yazılmaz. Cümle bütün olarak gösterilir ve bir önceki örnekle tamamen aynıdır. (Satırların alt alta görünmesini isteseydik; daha önce bahsettiğimiz gibi '\n' koymamız gerekirdi.)

Ekrana, *Ali: "Naber, nasılsın?" dedi.* şeklinde bir yazı yazdırmamız gerekiyor diyelim. Bu konuda ufak bir problem yaşayacağız. Çünkü printf(); fonksiyonu gördüğü ilk iki çift tırnak üzerinden işlem yapar. Böyle bir şeyi ekrana yazdırmak için aşağıdaki gibi bir program yazmamız gerekir:

```
#include<stdio.h>
int main( void )
{
        printf("Ali: \"Naber, nasılsın?\" dedi.\n");
        return 0;
}
```

printf(); fonksiyonunu kullanmayı sanırım iyice anladınız. printf(yazıp, sonra çift tırnak açıyor, yazmak istediklerimizi yazıyor, çift tırnağı sonra da parantezi kapatıyor, sonuna noktalı virgül ekliyoruz. Alt satıra geçmek içinse, yazdıklarımızın sonuna '\n' ekliyoruz. Çift tırnaklı bir şey kullanmak içinse \“ ... \” kullanıyoruz. Hepsi bu!

scanf(); fonksiyonuna gelince, bu başında bahsettiğimiz gibi bizim giriş (Input) fonksiyonumuzdur. Ancak yazımı burada noktalıyorum. Çünkü değişkenler işin içine girmekte ve onları anlatmam uzun

sürecek. Gelecek haftaki yazımda kaldığımız yerden devam edeceğiz. Yazdıklarımla ilgili öneri, eleştiri veya sorunuz varsa, bana ulaşabilirsiniz.

C Programlama Dersi - II

Bu yazıda öğreneceklerinizi:

- Bloodshed Dev-C++'in kullanımı
- Değişkenler ve değişken tanımlamaları
- Değişken tipleri ve maksimum-minimum alabileceği değerler
- scanf() fonksiyonunun kullanımı
- Aritmetik operatörler
- İşlemlerde öncelik sırası

Geçen hafta bilgisayar ve programlamaya dair temel bilgileri ve printf( ); fonksiyonunu öğrendik. Bu hafta, kaldığımız yerden devam edeceğiz. İlk yazıyla ilgili herhangi bir sorunuz varsa, bana cagataycebi@gmail.com adresinden ulaşabilirsiniz. Bu ufak bilgiden sonra, kaldığımız yerden devam edelim.

Bloodshed Dev-C++

Okuyucularımızın bir kısmı, Bloodshed Dev-C++'in kullanımıyla ilgili çeşitli sorunlar yaşamış. Programı nasıl kullanabileceğinize dair ufak bir açıklamayla yazımıza başlamak yerinde olacaktır. (Bu bölüm C derleyicisi olmayanlara yardımcı olmak için yazılmıştır. Eğer hâli hazırda bir derleyiciniz varsa ve sorunsuz kullanıyorsanız, "*Değişken nedir? Tanımı nasıl yapılır?*" bölümünden devam edebilirsiniz.)

Dev-C++ kullanımı oldukça basit bir program. Bloodshed Dev-C++ web sitesinin Download kısmından Dev-C++'i indirebilirsiniz. Dilerseniz bu bağlantıya tıklayarak yüklemeniz de mümkün. (Zaman içinde bağlantı adresi çalışmayabilir.) Programı başarıyla indirip kurarsanız, geriye yapacak fazla bir şey kalmıyor.

Program menülerinden, File -> New-> Source File yaparak yeni bir kaynak dosyası açın. (Ctrl + N ile de aynı işlemi yapabilirsiniz.) Aşağıdaki kodu deneme amacıyla, açtığınız dosyaya yazın:

```
#include<stdio.h>
int main( void )
{
   // Hello World yazar.
   printf( "Hello World" );
   // Sizden herhangi bir giriş bekler.
   // Böylece program çalışıp, kapanmaz.
   getchar( );
   return 0;
}
```

File -> Save As sekmesiyle, yazdığınız dosyayı kaydedin. (Ctrl + S ile de kaydedebilirsiniz.) Dosyanın adını verdikten sonra sonuna *.c* yazın. Örneğin *deneme.c* gibi...

Execute -> Compile sekmesine tıklayın. (Kısayol olarak Ctrl + F9'u kullanabilirsiniz.) Artık programınız derlendi ve çalışmaya hazır. Execute -> Run ile programınızı çalıştırın. (Ctrl + F10'u da deneyebilirsiniz.) Ekrana "*Hello World*" yazacaktır.

Eğer yazdığınız kodu tek seferde derleyip, çalıştırmak isterseniz, Execute -> Compile & Run yolunu izleyin. (Bu işlemin kısayol tuşu, F9'dur.)

Yazdığınız kodu nereye kaydederseniz, orada sonu *.exe* ile biten çalıştırılabilir program dosyası oluşacaktır. Örneğin C:\Belgelerim klasörüne *deneme.c* şeklinde bir dosya kaydedip, F9'a bastığınızda, *deneme.c*'nin bulunduğu klasörde *deneme.exe* diye bir dosya oluşur. Oluşan bu dosyayı istediğiniz yere taşıyabilir, dilediğiniz gibi çalıştırabilirsiniz.

Değişken nedir? Tanımı nasıl yapılır?

Değişkenler, girdiğimiz değerleri alan veya programın çalışmasıyla bazı değerlerin atandığı, veri tutucularıdır. Değişken tanımlamaysa, gelecek veya girilecek verilerin ne olduğuna bağlı olarak, değişken tipinin belirlenmesidir. Yani a isimli bir değişkeniniz varsa ve buna tamsayı bir değer atamak istiyorsanız, a değişkenini tamsayı olarak tanıtmanız gerekir. Keza, a'ya girilecek değer eğer bir karakter veya virgüllü sayı olsaydı, değişken tipinizin ona göre olması gerekirdi. Sanırım bir örnekle açıklamak daha iyi olacaktır.

```
#include<stdio.h>
int main( void )
{
        int a;
        a = 25;
        printf("a sayısı %d",a);
        return 0;
}
```

Şimdi yukarıdaki programı anlamaya çalışalım. En baş satıra, int a -int, İngilizce de integer'ın kısaltmasıdır- dedik. Bunun anlamı, tamsayı tipinde, a isimli bir değişkenim var demektir. a=25 ise, a değişkenine 25 değerini ata anlamına geliyor. Yani, a artık 25 sayısını içinde taşımaktadır. Onu bir yerlerde kullandığınız zaman program, a'nın değeri olan 25'i işleme alacaktır. printf(); fonksiyonunun içersine yazdığımız %d ise, ekranda tamsayı bir değişken değeri gözükecek anlamındadır. Çift tırnaktan sonra koyacağımız a değeri ise, görüntülenecek değişkenin a olduğunu belirtir. Yalnız dikkat etmeniz gereken, çift tırnaktan sonra, virgül koyup sonra değişkenin adını yazdığımızdır. Daha gelişmiş bir örnek yaparsak;

```
#include<stdio.h>
int main( void )
{
        int a;
        int b;
        int toplam;
        a = 25;
        b = 18;
        toplam = a + b;
        printf("a sayısı %d ve b sayısı %d, Toplamı %d.\n", a, b, toplam);
        return 0;
}
```

Bu programın ekran çıktısı şöyle olur; a sayısı 25 ve b sayısı 18, Toplamı 43. Yazdığımız bu programda, a, sonra b, üçüncü olarakta toplam ismiyle 3 adet tamsayı değişken tanıttık. Daha sonra a'ya 25, b'ye 18 değerlerini atadık. Sonraki satırdaysa, a ile b'nin değerlerini toplayarak, toplam ismindeki değişkenimizin içersine atadık. Ekrana yazdırma kısmı ise şöyle oldu: üç tane %d koyduk ve çift tırnağı kapattıktan sonra, ekranda gözükme sırasına göre, değişkenlerimizin adını yazdık. printf(); fonksiyonu içersinde kullanılan %d'nin anlamı, bir tamsayı değişkenin ekranda görüntüleneceğidir. Değişkenlerin yazılma sırasındaki olaya gelince, hangisini önce görmek istiyorsak onu başa koyar sonra virgül koyup, diğer değişkenleri yazarız. Yani önce a değerinin gözükmesini istediğimiz için a, sonra b değerinin gözükmesi için b, ve en sonda toplam değerinin gözükmesi için toplam yazdık.

Bu arada belirtmekte fayda var, elimizdeki 3 tamsayı değişkeni, her seferinde int yazıp, belirtmek zorunda değiliz. int a,b,toplam; yazarsakta aynı işlemi tek satırda yapabiliriz.

Şimdi, elimizdeki programı bir adım öteye taşıyalım:

```
#include<stdio.h>
int main( void )
{
        int saat;
        float ucret, toplam_ucret;
        char bas_harf;
        printf("Çalışanın baş harfini giriniz> ");
        scanf("%c",&bas_harf);
        printf("Çalışma saatini giriniz> ");
        scanf("%d",&saat);
        printf("Saat ücretini giriniz> ");
        scanf("%f",&ucret);
        toplam_ucret = saat * ucret;
        printf("%c başharfli çalışanın, alacağı ücret:
%f\n",bas_harf,toplam_ucret);
        return 0;
}
```

Bu yazdığımız program basit bir çarpım işlemini yerine getirerek sonucu ekrana yazdırıyor. Yazılanların hepsini bir anda anlamaya çalışmayın, çünkü adım adım hepsinin üzerinde duracağız. Programı incelemeye başlarsak; değişken tanımını programımızın başında yapıyoruz. Gördüğünüz gibi bu sefer farklı tiplerde değişkenler kullandık. Biri int, diğer ikisi float ve sonuncusunu da char. int'ın tamsayı anlamına geldiğini az evvel gördük. float ise 2.54667 gibi virgüllü sayılar için kullanılır. char tipindeki değişkenler, a,H,q,... şeklinde tek bir karakter saklarlar. Konu biraz karmaşık gözükse de, değişken tanımında bütün yapmanız gereken, değişkeninizin taşıyacağı veriye göre programın başında onun tipini belirtmektir. Bunun için de tıpkı yukarıdaki programda olduğu gibi, önce tipi belirtir, sonra da adını yazarsınız.

Programımıza dönersek, çalışma saati bir tamsayı olacağından, onu saat isminde bir int olarak tanıttık. Ücret virgüllü bir sayı olabilirdi. O nedenle onu float (yani virgüllü sayı) olarak bildirdik. Adını da saatucret koyduk. Farkettiğiniz gibi, toplamucret isimli değişkenimiz de bir float. Çünkü bir tamsayı (int) ile virgüllü sayının (float) çarpımı virgüllü bir sayı olmaktadır. Tabii 3.5x2=7 gibi tam sayı olduğu durumlarda olabilir. Ancak hatadan sakınmak için toplamucret isimli değişkenimizi bir float olarak belirtmek daha doğrudur.

Üsteki programımızda olmasına karşın, şuana kadar scanf(); fonksiyonunun kullanımına değinmedik. scanf(); geçen haftaki yazımızdan da öğrendiğimiz gibi bir giriş fonksiyonudur. Peki nasıl kullanılır, tam olarak ne işe yarar? scanf(); kabaca klavyeden girdiğiniz sayıyı veya karakteri almaya yarar. Kullanımı ise şöyledir: önce scanf yazar, sonra parantez ve ardından çift tırnak açar, daha sonra alınacak değişkene göre, %d, %f veya %c yazılır. %d int, %f float, %c char tipindeki değişkenler için kullanılır. Bundan sonra çift tırnağı kapatıp, virgül koyarsınız. Hemen ardından & işareti ve atanacak değişken adını yazarsınız. Son olarak, parantezi kapatıp noktalı virgül koyarsınız. Hepsi budur.

Yukarıdaki programda da scanf(); fonksiyonu gördüğünüz gibi bu şekilde kullanılmıştır. Sanırım gereğinden çok laf oldu ve konu basit olduğu halde zor gibi gözüktü. Yukardaki sıkıntıdan kurtulmak için çok basit bir program yazalım. Bu programın amacı, klavyeden girilen bir sayıyı, ekrana aynen bastırmak olsun.

```
#include<stdio.h>
int main( void )
{
        int sayi;
        printf("Değer giriniz> ");
        scanf("%d",&sayi);
```

```
        printf("Girilen değer: %d\n",sayi);
        return 0;
}
```

Gördüğünüz gibi hiçbir zor tarafı yok. Klavyeden girilecek bir tamsayınız varsa, yapmanız gereken önce değişkenin tipini ve adını belirtmek, sonra scanf( ); fonksiyonunu kullanmak. Bu fonksiyonu kullanmaya gelince, scanf(" yazdıktan sonra değişken tipine göre %d, %c, veya %f, yazıp, ardından & işaretini kullanarak atanacak değişkenin adını belirtmekten ibaret. Fark etmişsinizdir, printf(); ve scanf(); fonksiyonlarının her ikisinde de %d koyduk. Çünkü scanf( ); ve printf( ); fonksiyonların değişken tipi simgeleri aynıdır. Aşağıdaki tablodan hangi değişken tipinin nasıl deklare edileceğini, kaç byte yer kapladığını, maksimum/minimum alabileceği değerleri ve giriş/çıkış fonksiyonlarıyla nasıl kullanılabileceğini bulabilirsiniz. Tanımlamalar ve fonksiyon uygulamaları, *degisken* isimli bir değişken için yapılmıştır.

TİP	DEKLARASYON	printf( );	scanf( );	Minimum	Maksimum	Byte
Karakter	char degisken;	printf("%c",degisken);	scanf("%c",&degisken);	-128	127	1
Kısa Tam Sayı	short degisken;	printf("%d",degisken);	scanf("%d",&degisken);	-32768	32767	2
Tamsayı	int degisken;	printf("%d",degisken);	scanf("%d",&degisken);	-32768	32767	2
Uzun Tamsayı	long int degisken;	printf("%ld",degisken);	scanf("%ld",&degisken);	-2147483648	2147483647	4
İşaretsiz Tamsayı	unsigned int degisken;	printf("%u",degisken);	scanf("%u",&degisken);	0	65535	2
İşaretsiz Uzun Tamsayı	long unsigned degisken;	printf("%lu",degisken);	scanf("%lu",&degisken);	0	4294967295	4
Virgüllü Sayı	float degisken;	printf("%f",degisken);	scanf("%f",&degisken);	1,17549e-38	3,40282e+38	4
Uzun Virgüllü Sayı	double degisken;	printf("%lf",degisken);	scanf("%lf",&degisken);	2,22504e-308	1,79769e+308	8

Verilen bu değerler; işletim sisteminden, işletim sistemine, farklılık gösterebilir. En doğru değerleri almak için sizeof( ), fonksiyonunu kullanmak gerekir. Aşağıda yazmış olduğum bir program bulacaksınız. Kendi bilgisayarınızda derleyip, çalıştırırsanız, size değişkenlerin boyutunu ve alabileceği maksimum-minimum değerleri verecektir:

```
#include<stdio.h>
#include<limits.h>
#include<float.h>
int main( void )
{
        printf( "\nTIP\t\t  BOYUT\t\t MIN\t \tMAX\n" );
```

```
        printf("============================================================\n
");
        printf( "char\t\t: %d byte(s)\t%d\t\t%d\n",
sizeof(char),CHAR_MIN,CHAR_MAX );
        printf( "short\t\t: %d byte(s)\t%d\t\t%d\n", sizeof(short), SHRT_MIN,
SHRT_MAX );
        printf( "int\t\t: %d byte(s)\t%d\t%d\n", sizeof(int), INT_MIN,
INT_MAX );
        printf( "unsigned int\t: %d byte(s)\t\t\t%u\n",sizeof(unsigned),UINT_MAX
);
        printf( "long\t\t: %d byte(s)\t%ld\t%ld\n", sizeof(long), LONG_MIN,
LONG_MAX );
        printf( "float\t\t: %d byte(s)\t%e\t%e\n", sizeof(float), FLT_MIN,
FLT_MAX );
        printf( "double\t\t: %d byte(s)\t%e\t%e\n\n", sizeof(double), DBL_MIN,
DBL_MAX );

        return 0;
}
```

Programı inceleyip, detaylara girmeyin. Sadece çalıştırıp, sonuçları görmeniz yeterli. Örneğin, Ubuntu yüklü x86 tabanlı bir bilgisayarda, karşınıza şöyle bir ekran gelecektir:

```
TIP               BOYUT          MIN            MAX
============================================================
char            : 1 byte(s)     -128           127
short           : 2 byte(s)     -32768         32767
int             : 4 byte(s)     -2147483648    2147483647
unsigned int    : 4 byte(s)                    4294967295
long            : 4 byte(s)     -2147483648    2147483647
float           : 4 byte(s)     1.175494e-38   3.402823e+38
double          : 8 byte(s)     2.225074e-308  1.797693e+308
```

Sanırım hiç istemediğim bir şey yaparak, kafanızı karıştırdım. Dilerseniz, burada keselim ve bunlar ileriye dönük olarak bir kenarda dursunlar. Yine de daha fazla bilgi isterseniz, Teach Yourself C in 21 Days yazısına göz atabilirsiniz. Ama dediğim gibi bunların sonraya bırakılması, uygun olacaktır.

Şu ana kadar öğrendiklerimizle girilen herhangi iki sayısının ortalamasını hesaplayan bir program yazalım. Başlamadan önce, değişkenlerimizin kaç tane ve nasıl olacağını düşünelim. Şurası kesin ki, alacağımız iki sayı için 2 farklı değişkenimiz olmalı. Bir de ortalamayı hesapladığımızda bulduğumuz değeri ona atayabileceğimiz bir başka değişkene ihtiyacımız var. Peki değişkenlerimizin tipleri ne olacak? Başında belirttiğimiz gibi yazmamız gereken program herhangi iki sayı için kullanılabilmeli. Sadece tamsayı demiyoruz, yani virgüllü bir sayı da girilebilir. O halde, girilecek iki sayının değişken tipi float olmalı. Bunu double da yapabilirsiniz, fakat büyüklüğü açısından gereksiz olacaktır. Ortalamaların atanacağı üçüncü değişkene gelince, o da bir float olmalı. İki virgüllü sayının ortalamasının tamsayı çıkması düşünülemez. Oluşturduğumuz bu önbilgilerle programımızı artık yazabiliriz.

```
#include<stdio.h>
int main( void )
{
        float sayi1,sayi2,ortalama;
        printf("İki sayı giriniz> ");
        scanf("%f%f",&sayi1,&sayi2);
        ortalama = ( sayi1 + sayi2 ) / 2;
        printf("Ortalama sonucu: %f'dir",ortalama);
        return 0;
}
```

Yukarıda yazılı programda, bilmediğimiz hiçbir şey yok. Gayet basit şekilde izah edersek, 2 sayı alınıp, bunlar toplanıyor ve ikiye bölünüyor. Bulunan değerde ortalama isminde bir başka değişkene

atanıyor. Burada yabancı olduğumuz, sadece scanf(); kullanımındaki değişiklik. scanf(); fonksiyonuna bakın. Dikkat edeceğiniz gibi, değişkenlerden ikisine de tek satırda değer atadık. Ayrı ayrı yazmamız da mümkündü, ancak kullanım açısından böyle yazmak açık şekilde daha pratiktir. Bu konuda bir başka örnek verelim. Diyelim ki, biri int, diğeri float, sonuncusuysa char tipinde 3 değişkeni birden tek scanf(); ile almak istiyorum. Değişkenlerin isimleri, d1,d2 ve d3 olsun. Nasıl yaparız?

```
scanf("%d%f%c",&d1,&d2,&d3);
```

Peki aldığımız bu değişkenleri ekrana tek printf(); ile nasıl yazdırabiliriz?

```
printf("%d %f %c",d1,d2,d3);
```

Görüldüğü gibi bu işin öyle aman aman bir tarafı yok. Fonksiyonların kullanımları zaten birbirine benziyor. Tek yapmanız gereken biraz pratik ve el alışkanlığı.

Aritmetik Operatör ve İfadeleri

Üste yazdığımız programların hemen hemen hepsinde aritmetik bir işlem kullandık. Ama aritmetik işlemleri tam olarak anlatmadık. Kısaca;

```
( + ) : Artı

( - ) : Eksi

( / ) : Bölme

( * ) : Çarpma

( % ) : Modül
```

Burada bilinmeyen olsa olsa modül işlemidir. Modül kalanları bulmaya yarar. Yani diyelim ki 15'in 6'ya olan bölümünden kalanını bulmak istiyorsunuz. O halde 15%6 = 3 demektir. Veya, 7'nin 3'e bölümünden kalanı bulacaksanız, o zamanda 7%3 = 1 elde edilir. Bu C'de sıkça kullanacağımız bir aritmetik operatör olacak.

İşlem sırasına gelince, o da şöyle olur. En önce yapılan işlem parantez ( ) içidir. Sonra * / % gelir. Çarpma, bölme ve modül için, soldan sağa hangisi daha önce geliyorsa o yapılır. En son yapılanlarsa artı ve eksidir. Keza, bu ikisi arasında, önce olan solda bulunandır.

Bölme işlemine dair, bir iki ufak olay daha var. 4/5 normalde 0.8 etmektedir. Ancak C için 4/5 sıfır eder. Çünkü program, iki tamsayının bölünmesiyle, sonucu tamsayı elde etmeyi bekler. İleride tipleri birbiri arasında değiştirmeye değineceğiz. Ama şimdilik bu konuda bir-iki örnek yapalım:

```
*       8/4+2 => 2 + 2 => 4
*       8-4*2+-12 => 8 - 8 + -12 => -12
*       15*4/2%4*7 => 60/2%4*7 => 30%4*7 => 2*7 => 14
*       31+7/2-83%5*2-2 => 31+ 3 - 3 * 2 - 2 => 31 + 3 - 6 - 2 => 26
*       (31-7) * 2 + 83 / (5%2) => 24 * 2 + 83 / 1 => 48 + 83 => 131
```

Bu aritmetik ifadeleri henüz bir C programı için denemedik. Ancak burada keselim. Bunu yapmayı diğer yazımıza saklayalım. Eğer uğraşmak isterseniz klavyeden alınacak 3 sayının ortalamasını bulan bir program yazabilirsiniz. Yada girilecek 2 tamsayı arasında bütün aritmetik işlemleri -ikisin çarpımını, toplamını, birbirine bölümünü ve farkını- bulan ve sonuçları ekrana yazdıran bir program da yazmanız mümkün. Herhangi bir yerde takılır ve bana ulaşmak isterseniz mail adresime yazmanız kafi. Haftaya görüşürüz.

C Programlama Dersi - III

Bu yazıda öğreneceksiniz:

- Kodlarınıza açıklama (comment) koymak
- Cast Operator
- Koşullu (Conditional) İfadeler
- Koşullu Operatörler (if, else, vs...)
- İlişkisel (Relational) Operatörler
- Birleşik (Compound) Operatörler
- Konuyla ilgili örnek sorular

Kodlarınıza açıklama (comment) koymak

Yazılım mühendislerinin en büyük sıkıntısı kod yazmak değildir. Yazılmış bir kodu okuyup anlamak -hele ki büyük bir projeden söz ediyorsak- asıl başınıza bela olacak konudur. Bundan korunmak için kodlarımıza açıklama/yorum koyarız.

C programlama dilinde iki şekilde açıklama koymak mümkündür. Bunlardan bir tanesi, satır bazında yapılır. Diğeriyse, belirli bir bloğu yorumlamaya yarar. Compiler her iki şekilde de, açıklama olarak belirlemiş yerleri işlemeyecektir. Aşağıdaki örnekte satır ve blok olarak, nasıl kodlarınıza açıklama getirebileceğinizi görebilirsiniz:

```
/*Çok satırlı bir açıklama.
Yıldızlar arasında kalan bütün
alan, yorum olarak değerlendirilir
ve derleyici (compiler) tarafından
işlenmez.
*/
#include<stdio.h>
int main( void )
{
        //Tek satırlık bir açıklama.
        printf("Hello World\n");
}
```

Cast Operator

Cast operator'un Türkçe karşılığı olacak bir kelime aklıma gelmedi. Ancak cast operatoru şu şekilde açıklayabiliriz. Bir değişken tipini örneğin (Tam sayı-int), bir başka tipe (virgüllü sayı-float) gibi dönüştürmek isterseniz, o zaman cast operator kullanırız.

Aşağıdaki kodu yazıp derleyin.

```
#include<stdio.h>
int main( void )
{
        int bolunen = 12, bolen = 8;
        float bolum;
        bolum = bolunen / bolen;
        printf("Sonuc: %f\n",bolum);
        return 0;
}
```

Program çıktısı; "*Sonuc: 1.000000*" olacaktır:

Normalde 1.5 çıkmasını beklediğiniz sonucun, 0.000000 çıkmasının nedeni casting kullanmamamızdır. Bir tam sayıyı, bir başka tam sayıya bölerseniz, sonuç bir başka tam sayı çıkar. Ve C programlama dili, bir virgüllü sayıyı tam sayıya atamaya kalktığınızda, herhangi bir yuvarlama işlemi yapmadan, virgülden sonrası atar.

Cast Operator şu şekilde kullanılmalıdır: *degisken_1 = (tip) degisken_2;*

Elimizdeki bu bilgiye göre programımızı tekrar yazalım.

```
#include<stdio.h>
int main( void )
{
        int bolunen = 12, bolen = 8;
        float bolum;
        bolum = (float)bolunen / bolen;
        printf("Sonuc: %f\n",bolum);
        return 0;
}
```

Sonuç bu sefer, beklediğimiz gibi 1.5 çıkacaktır.

Aşağıdaki örneği inceleyelim:

```
#include<stdio.h>
int main( void )
{
        printf("Sonuc: %f\n", 2 / 4);
        return 0;
}
```

Öğrendiğimiz üzere, bunun da sonucu 0.5 yerine, 0 olarak gözükecektir. Sonucu doğru yazdırmak için (float)2/4 şeklinde yazmanız yeterlidir. Ancak dahi basit bir yöntem olarak 2/4.0 veya 2.0/4 yazarsanız yine aynı sonucu elde edersiniz.Çünkü bu durumda sayılardan bir tanesi float olmaktadır.

Kullanılan değişken tiplerinden hangisi büyükse, sonuç o değişkenin tipinde döner. Yüksek değişken bellekte daha fazla yer kaplamaktadır. Bunun doğal bir sonucu olarakta, domine eden o'dur. Değişkenlerin büyüklüğünü daha önceki dersimizde vermiştik. Ancak hatırlamak açısından, aşağıya bakabilirsiniz.

```
(DÜŞÜK) char <-> int <-> long <-> float <-> double (YÜKSEK)
```

Çıkan sonucu, daha düşük bir değişken tipine atamaya kalkarsanız, o zaman veri kaybı yaşanır. Ve örneğin float 1.5 olan sonucu int değişkene 1.0 olarak kaydedilir.

Öğrendiklerinizin pekişmesi için bir program yazalım. Bu programda, klavyeden girilen, bir virgüllü sayının, yuvarlanıp, tam sayı olması gösterilsin:

```
#include<stdio.h>
int main( void )
{
        float girilen_sayi;
        printf("Lütfen bir sayı giriniz> ");
        scanf("%f",&girilen_sayi);
        printf("Sayının yuvarlanmış hali: %d\n", (int)(girilen_sayi+0.5));
        return 0;
}
```

Koşullu (Conditional) İfadeler

if

Bilgisayarda yapılan bütün mantıksal işlemler kaba bir temele dayanır. Şartlar sağlandığı halde yapılacak işlem belirlenir. Ve şartlar sağlandığında, bu işlemler yapılır. Şartların kontrol edilmesini, C (ve daha birçok) programlama dilinde ***if*** operatörünü kullanarak yaparız.

if operatörünün genel kullanım yapısı şu şekildedir:

```
if( koşul ) {
        komut(lar)
}
```

Eğer if'in altında birden çok komut varsa, ayraç işareti (veya küme parantezi) koymamız gerekir. Şayet if'ten sonra, tek komut bulunuyorsa, ayraç koyup-koymamak size kalmıştır. Zorunluluğu yoktur.

Örnek bir program yazalım. Bu programda kullanıcının klavyeden, bir tam sayı girsin. Ve bizde girilen sayı, 100'den büyükse, ekrana yazdıralım:

```
#include<stdio.h>
int main( void )
{
        int girilen_sayi;
        printf("Lütfen bir tam sayı giriniz> ");
        scanf("%d",&girilen_sayi);
        if( girilen_sayi > 100 )
                printf("Girilen sayı 100'den büyüktür\n");
        return 0;
}
```

if-else

Bazı durumlarda, bir koşulun doğruluğuna göre sonuç yazdırmak yetmez. Aksi durumda da ne yapacağımızı belirtmek isteriz. Bunun için if-else yapısını kullanırız.

if-else yapısı şu şekildedir:

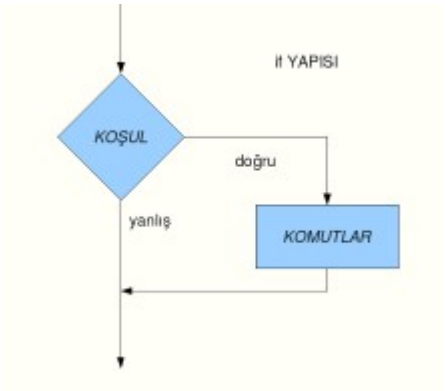
```
if( koşul ) {
        komut(lar)
}
else {
        komut(lar)
}
```

Önceki yazdığımız programı düşünelim; 100'den büyük olduğunda, ekrana çıktı alıyorduk. Bu programa bir özellik daha ekleyelim ve 100'den küçükse, bunu da söyleyen bir yapıyı oluşturalım:

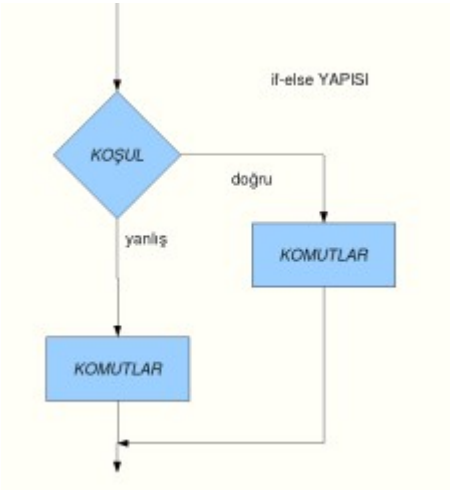
```
#include<stdio.h>
int main( void )
{
        int girilen_sayi;
        printf("Lütfen bir tam sayı giriniz> ");
        scanf("%d",&girilen_sayi);
        if( girilen_sayi > 100 )
                printf("Girilen sayı 100'den büyüktür\n");
        else
                printf("Girilen sayı 100'den küçüktür\n");
        return 0;
}
```

Örnekte gördüğünüz gibi, bir koşulun doğruluğunu program kontrol ediyor ve buna doğru olursa, bazı işlemler yapıyor. Şayet verilen koşul yanlışsa, o zaman daha başka bir işlem yapıyor. Ancak ikisini de yapması gibi bir durum söz konusu değil.Aşağıdaki akış diyagramlarında (flowchart) her iki durumu da görebilirsiniz.

if Yapısı:



if-else Yapısı:



İlişkisel (Relational) Operatörler

Koşullu operatörlerde, koşulun doğruluğunu kontrol ederken kullandığımız ilişkisel operatörler, aşağıda verilmiştir:

<	Küçüktür
>	Büyüktür
==	Eşittir
<=	Küçük eşittir
>=	Büyük eşittir
!=	Eşit değildir

Birleşik (Compound) Operatörler

Bazı durumlarda, kontrol edeceğimiz koşul, tek bir parametreye bağlı değildir. Örneğin, bir kişinin yaşının 65'den küçük olup olmadığına bakabiliriz. Ama 65'den küçük ve 18 yaşından büyük olup olmadığına karar vermek istersek, o zaman Birleşik/Birleştirici Operatörler'i kullanmamız uygun olacaktır.

Compound operator'ler aşağıdaki gibidir:

&&	and	ve
||	or	veya

!	not	tersi

Bu operatörlerin mantıksal (logical) doğruluk tablosu da şu şekildedir:

p	**q**	**p&&q**	**p||q**	**!p**
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Örnek Sorular

Soru 1: Klavyeden alınacak bir değerin, 18 ile 65 arasında olup olmadığını kontrol eden bir program yazınız:

Soru 2: Kendisine verilen iki tam sayıyı, bölecek ve sonucu virgüllü sayı olarak gösterecek bir bölme işlemini programı hazırlayınız. (Şayet bölen 0 olarak verilirse, bölme işlemi yapılmamalıdır.)

Not: Yukarda verilmiş çözümler, daha efektif olabilirdi. Ancak verdiğim cevapların herkes tarafından anlaşılmasını istediğimden, işi biraz uzatmış olabilirim. Sizin vereceğiniz cevapların gidiş yolları elbette ki farklı olabilir.

Cevap1:

```
/*
Girilen yaşın, 18 ile 65 arasında
olup olmadığını belirler.
*/
#include<stdio.h>
int main( void )
{
        int girilen_yas;
        printf("Lütfen yaşınızı giriniz> ");
        scanf("%d",&girilen_yas);
        if( girilen_yas >= 18 && girilen_yas <= 65 )
                printf("Girilen yaş, 18 ile 65 arasındadır.\n");
        //Girilen yaş 18 ile 65 arasında değilse, aşağıdaki else
        //bloğu çalışır.
        else {
                //Girilen yaş 18'den küçükse
                if( girilen_yas < 18 )
                        printf("Girilen yaş, 18'den küçüktür.\n");
                //Girilen yaş 65'ten büyükse
                else
                        printf("Girilen yaş, 65'ten büyüktür.\n");
        }
}
```

Cevap2:

```
/*
Kendisine verilen iki tam sayıyla
bölme işlemi yapan program.
*/
#include<stdio.h>
int main( void )
{
        int bolunen, bolen;
        float sonuc;
        printf("Bölünecek sayıyı giriniz> ");
        scanf("%d",&bolunen);
        printf("Bölen sayıyı giriniz> ");
        scanf("%d",&bolen);
        //Bolen, 0 ise, bir sayı sıfıra bölünemeyeceğinden,
        //program sorun çıkartacaktır. Bu yüzden,
        //bolenin 0 olmaması kontrol ediliyor.
        if( bolen != 0 ) {
                sonuc = (float)bolunen / bolen;
                //.2f, virgülden sonra 2 basamak gösterilmesi
                //içindir.
                printf("Sonuc: %.2f\n",sonuc);
        }
        else
                printf("Hata: Sayı 0'a bölünemez!\n");

}
```

C Programlama Dersi - IV

Bu yazıda öğreneceksiniz:

- İçiçe geçmiş (Nested) koşullu ifadeler
- if - else if merdiveni
- switch - case ifadesi
- Arttırma (Increment) ve azaltma (Decrement) işlemleri
- Gelişmiş atama (Advanced Assignment) yöntemleri
- Conditional Operator (?)
- Konuyla ilgili örnek sorular

İçiçe geçmiş (Nested) İfadeler

Daha önce, koşullu ifadeleri görmüştük. Hatırlatmak için üzerinden geçersek, if ile bir ifadeyi

kontrol ediyor ve doğruysa, buna göre işlemler yapıyorduk. Bir de if - else yapısı vardı. if - else yapısında da, koşulu gene kontrol ediyor, doğruysa if bloğunun altında kalanları yapıyorduk; yanlışsa, else bloğunda olan kodlar işleme alınıyordu. Son derece basit bir mantık üzerine kurulmuş bu yapıyla, yapılamayacak kontrol yoktur. Ancak öyle durumlar vardır ki, if - else yapısı yeterli verimliliği sunamaz.

Diyelim ki, birden fazla kontrol yapmanız gereken bir durum oluştu. Hatta örnek vererek konuyu daha da somutlaştıralım. İstenilen bir programda, klavyeden size yaş bilgisi veriliyor. Siz de bu bilgiye göre, şayet yaş 18'den küçükse çocuk; 18-30 yaş arasında genç; 30-50 yaş arasında ortayaş diye bir mesaj bastırıyorsunuz. Basit bir program.

Şimdi bunu sadece if yapısıyla kuruyor olsaydık, her seferinde yaşın uygun aralıklara düşüp düşmediğini kontrol eder ve ona göre sonucu ekrana bastırırdık. Ama bu son derece verimsiz bir yöntem olurdu. Çünkü zaten yaş bilgisinin genç olduğuna dair bir karar vermişsek, sonrasında tutup bunun yaşlı olup olmadığını kontrol etmenin bir esprisi olmayacaktır. Verilebilecek en kötü cevabı aşağıda bulabilirsiniz:

```
/*
Sorulan soruya verilebilecek en
kötü cevap.
*/
#include<stdio.h>
int main( void )
{
        int girilen_yas;
        printf("Lütfen yaşınızı giriniz> ");
        scanf("%d",&girilen_yas);
        if( girilen_yas < 18 )
                printf("Daha çocuk yaştasınız, hayatın başındasınız.\n");
        if( girilen_yas >= 18 && girilen_yas <= 30 )
                printf("Gençliğin, güzelliği bambaşka!\n");
        if( girilen_yas > 30 && girilen_yas <= 50 )
                printf("Hepsini boşverin, olgunluk ortayaşta başlar!\n");
        return 0;
}
```

Yukarda ki kodu if - else kullanarak daha efektif hale getirebiliriz:

```
/*
if - else yapısıyla daha
efektif bir yapı
*/
#include<stdio.h>
int main( void )
{
        int girilen_yas;
        printf("Lütfen yaşınızı giriniz> ");
        scanf("%d",&girilen_yas);
        if( girilen_yas < 18 )
                printf("Daha çocuk yaştasınız, hayatın başındasınız.\n");
        else {
                if( girilen_yas >= 18 && girilen_yas <= 30 )
                        printf("Gençliğin, güzelliği bambaşka!\n");
                else {
                        if( girilen_yas > 30 && girilen_yas <= 50 )
                                printf("Hepsini boşverin, olgunluk ortayaşta
başlar!\n");
                }
        }
        return 0;
}
```

Yukardaki program daha efektif bir yapı sunmuş olmasına rağmen, eğer kontrol ettiğimiz aralıkların sayısı çok fazla olsaydı, tam bir başbelası olacaktı! Çünkü if - else içinde, bir başka if - else bloğu ve onun içinde bir başkası... bu böyle sürüp gidecekti. Kısacası performans olarak çok bir şey değişmese de, kodu yazan ve/veya okuyacak olan için tam bir eziyete dönüşecekti. İşte bu nedenlerle daha efektif yapılara ihtiyaç duyuyoruz.

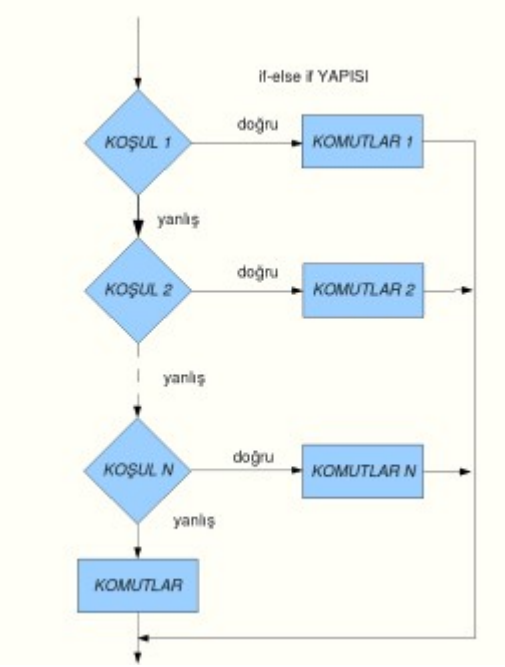
if - else if Merdiveni

if - else if merdiveni yukarda verdiğimiz örnekler için biçilmiş kaftandır. if - else if merdiveni, doğru bir şey bulduğu zaman kontrolu orada keser ve diğer koşulları kontrol etmeden blok sonlandırılır.

Aşağıda if - else if yapısını ve akış diyagramını bulabilirsiniz:

if - else if Yapısı ***if - else if Akış Diyagramı***

```
if( koşul 1 ) {
        komut(lar) 1
}
else if( koşul 2 ) {
        komut(lar) 2
}
        .
        .
        .
else if( koşul n ) {
        komut(lar) n
}
else {
        komut(lar) n
}
```



if - else if ile söylenebilecek son bir şey sonunda ki else'tir. else koymak zorunlu değildir. Ancak hiçbir koşula uymayan bir durumla karşılaştığınızda, else devreye girer. Örneğin yukarda anlatıp, kodunu vermiş olduğumuz programda, belirtilen yaş aralıklarında değer girilmezse, hiçbir şey ekrana bastırılmayacaktır. Çünkü programa tanınmayan yaş aralığında ne yapılacağı öğretilmemiştir. Şimdi bu durumu da içerecek şekilde, programamımızı if - else if yapısıyla tekrar yazalım:

```
#include<stdio.h>
int main( void )
{
        int girilen_yas;
        printf("Lütfen yaşınızı giriniz> ");
        scanf("%d",&girilen_yas);

        if( girilen_yas < 18 )
                printf("Daha çocuk yaştasınız, hayatın başındasınız.\n");
        else if( girilen_yas >= 18 && girilen_yas <= 30 )
                        printf("Gençliğin, güzelliği bambaşka!\n");
        else if( girilen_yas > 30 && girilen_yas <= 50 )
                        printf("Hepsini boşverin, olgunluk ortayaşta
başlar!\n");
        else
```

```
                printf("HATA: Girilen yaş tanımlı değildir!\n");

        return 0;
}
```

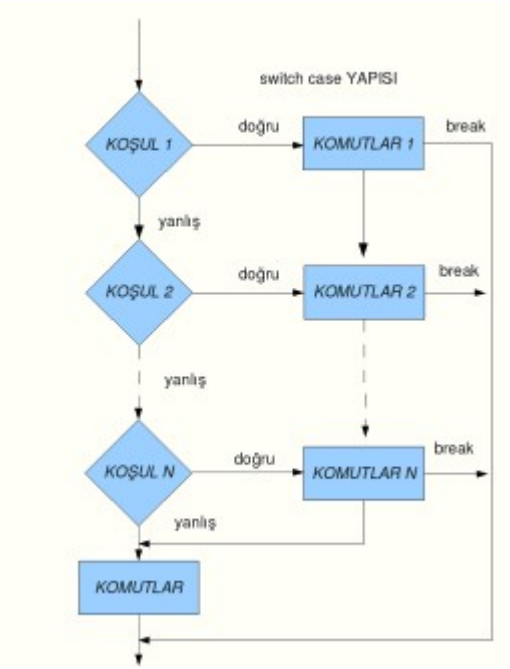
swicth - case ifadesi

switch - case, if - else if yapısına oldukça benzer bir ifadedir. Ancak aralarında iki fark vardır. Birincisi, switch - case yapısında, aralık değeri girmezsiniz. Direkt olarak ifadelerin bir şeylere eşit olup olmadığına bakarsınız. İkinci farksa, switch - case yapılarında, illa ki uygun koşulun sağlanmasıyla yapının kesilmek zorunda olmayışıdır. 'break' komutu kullanmadığınız takdirde, diğer şartların içindeki işlemleri de yapma imkanınız olabilir. switch case en tepeden başlayarak şartları tek tek kontrol eder. Uygun şart yakalanırsa, bundan sonra ki ifadeleri kontrol etmeden doğru kabul eder. Ve şayet siz break koymamışsanız, eşitlik uygun olsun olmasın, alt tarafta kalan case'lere ait komutlarda çalıştırılacaktır. if - else if ise daha önce söylemiş olduğumuz gibi böyle değildir. Uygun koşul sağlandığında, yapı dışarsına çıkılır.

switch case yapısında ki durumu, aşağıdaki tabloda görebilirsiniz:

switch case Yapısı ***switch case Akış Diyagramı***

```
switch( degisken ) {
        case sabit1:
                komut(lar)
                [break]
        case sabit2:
                komut(lar)
                [break]
        .
        .
        .
        case sabitN:
                komut(lar)
                [break]
        default:
                komut(lar);
}
```



Sanırım gözünüze biraz farklı gözüktü. Yapı olarak şimdiye kadar görmüş olduğunuz if else gibi gözükmese de, bir örnekten sonra arasında pek bir fark olmadığını göreceksiniz. Her komut sonunda koyduğum break komutu, zorunlu değildir ve o nedenle köşeli parantezle belirtilmiştir. break koyduğuz takdirde, uygun koşul sağlandıktan sonra, daha fazla kontrol yapılmayacak ve aynen if - else if yapısında olduğu gibi program orada kesilecektir. Ama break koymazsanız, altında kalan bütün işlemler -bir daha ki break'e kadar- yapılacaktır.

Kodun sonunda gördüğünüz default komutu, if - else if yapısında ki sonuncu else gibidir. Uygun hiçbir şart bulunamazsa, default komutu çalışır.

Öğrendiğimiz bilginin pekişmesi için biraz pratik yapalım. Bir not değerlendirme sistemi olsun. 100 - 90 arası A, 89 - 80 arası B, 79 - 70 arası C, 69 - 60 arası D, 59 ve altıysa F olsun. Eğer 100'den büyük veya negatif bir sayı girilirse, o zaman program hatalı bir giriş yapıldığını konusunda bizleri uyarsın. Bunu şimdiye kadar öğrendiğiniz bilgilerle, if - else if yapısını kullanarak rahatlıkla yanıtlayabilirsiniz. Ama şu an konumuz switch case olduğundan, cevabını öyle verelim:

```
#include<stdio.h>
int main( void )
{
        int not;
        printf("Lütfen notu giriniz> ");
        scanf("%d",&not);
        switch( not / 10 ) {
                case 10:
                case 9: printf("NOT: A\n"); break;
                case 8: printf("NOT: B\n"); break;
                case 7: printf("NOT: C\n"); break;
                case 6: printf("NOT: D\n"); break;
                case 5:
                case 4:
                case 3:
                case 2:
                case 1:
                case 0: printf("NOT: F\n"); break;
                default:
                        printf("HATA: Bilinmeyen bir değer girdiniz!\n");
        }

        return 0;
```

```
}
```

Algoritmaya bakalım: Önce sayıyı alıyor ve 10'a bölüyoruz. Yani girilen not, 57 ise 5.7 sonucunu elde ediyoruz. Ancak iki tam sayının sonucu bir virgüllü sayı veremez, tıpkı işleme giren değişkenler gibi tam sayı olarak döner. Dolayısıyla bilgisayarın elde edeceği sonuç, 5.7 değil, sadece 5'tir. switch case yapısında koşullar yukardan başlayarak kontrol ediliyor. case 5'e gelindiğinde eşitlik sağlanıyor. Ama break konmadığı için, switch case'ten çıkılmıyor. Ve altında kalan işlemlerde yapılıyor. Altında herhangi bir işlem veya break olmadığından case 0'a kadar bu böyle sürüyor. Ve case 0'da ekrana bir çıktı alıp switch case yapısı break ile sonlandırılmaktadır.

switch case, if - else if yapısının sunduğu esnekliğe sahip değildir. Daha çok menü olarak sunulacak işlerde kullanılır. Örneğin Unix'in ünlü listeleme komutu ls içersinde, verilen parametrelerin kontrolü switch case kullanılarak sağlanmıştır. Open Solaris, FreeBSD veya Linux kodlarını incelerseniz bunun gibi yüzlerce örnek bulabilirsiniz.

Arttırma (Increment) ve azaltma (decrement) işlemleri

Daha önceki derslerimizde, aritmetik işlemlerden bahsetmiştik. Bunların dışında yapabileceğimiz başka şeylerde bulunmaktadır. Bunlardan biri de, arttırma ve azaltma işlemleridir.

Eğer i adında bir değişkenin değerini 1 arttırmak isterseniz, i = i + 1 olarak yazarsınız. Veya 1 azaltmak isterseniz, benzer şekilde i = i - 1 de yazabilirsiniz. Arttırma ve azaltma işlemleri bu olayı daha basit bir forma sokmaktadır. i = i + 1 yazmak yerine i++ veya i = i - 1 yazmak yerine i-- yazabilirsiniz.

Arttırma ve azaltma işlemleri temelde iki çeşittir. Birinci yöntemde yukarda yazdığımız gibi, arttırma/azaltma sonradan yapılır. İkinci yöntemdeyse arttırma/azaltma ilk başta yapılır. Aşağıdaki örneklere bakalım.

```
/*
Bu programda, arttırma ve azaltma
işlemleri önce yapılacaktır.
*/
#include<stdio.h>
int main( void )
{
        int i = 10, j = 60;
        printf("i = %d ve j = %d\n", ++i, --j);
        return 0;
}
```

Yukardaki programı yazar ve çalıştırısanız elde edeceğiniz çıktı şu şekilde görünecektir:

```
i = 11 ve j = 59
```

Çünkü arttırma ve azaltma işlemleri ekrana bastırmadan önce yapılmış ve i ile j'nin değerleri değiştirilmiştir. Şimdi programı değiştirip şöyle yazalım:

```
/*
Bu programda, arttırma ve azaltma
işlemleri sonra yapılacaktır.
*/
#include<stdio.h>
int main( void )
{
        int i = 10, j = 60;
        printf("i = %d ve j = %d\n", i++, j--);
        return 0;
}
```

Bu sefer program çıktısı şöyle olacaktır:

```
i = 10 ve j = 60
```

Farkettiğiniz üzere hiçbir değişiklik yapılmamış gibi duruyor. Aslında değişiklik yapıldı ve program sonlanmadan önce i 10 olurken, j'de 59 oldu. Ama arttırma ve azaltma işlemleri printf komutu çalıştırıldıktan sonra yapıldığı için, biz bir değişiklik göremedik.

Kısacası önce arttırma (pre-increment) veya önce azaltma (pre-decrement) kullandığınızda, ilgili komut satırında çalışacak ilk şey bu komutlar olur. Ancak sonra arttırma (post increment) veya sonra azaltma kullanırsanız, o zaman bu işlemlerin etkileri ilgili komut satırından sonra geçerli olacaktır. Aşağıdaki özel tabloya bakabilirsiniz:

Form	Tip	İsim	Açıklama
i++	postfix	post-increment	İşlem sonrası arttırma
++i	prefix	pre-increment	İşlem öncesi arttırma
i--	postfix	post-decrement	İşlem sonrası azaltma
--i	prefix	pre-decrement	İşlem öncesi azaltma

Gelişmiş atama (Advanced Assignment) yöntemleri

C'de yazım kolaylığı amacıyla sunulmuş bir başka konu da, gelişmiş aşama yöntemleridir. Biraz daha uzun yazacağınız kodu, kısaltmanıza yaramaktadır.

degisken_1 = degisken_1 (operator) degisken_2 şeklinde yazacağınız ifadeleri, daha kısa yazabilmeniz için, degisken_1 (operator) = degisken_2 şeklinde ifade edebilirsiniz. Gelişmiş atamalarda sunulan genel formlar şu şekildedir:

```
+= , -= , *= , /= , %=
```

Sanırım aşağıdaki örneklere bakarsanız, konuyu çok daha net anlayacaksınız:

```
1-) j = j * ( 3 + x )  ==> j *= ( 3 + x )
2-) a = a / ( 5 - z ) ==> a /= ( 5 - z )
3-) x = x - 5 ==> x -= 5
```

Conditional Operator (?)

Türkçe karşılık bulamadığım bir başka C kavramı da, Conditional Operator. Aslında mot a mot çeviri yaparsam, koşullu operatör anlamına geliyor. Ama şu ana kadar gördüğümüz birçok yapıyı da bu şekilde tanımlamak mümkünken, koşullu operatör ifadesini kullanmayı pek tercih etmiyorum. Neyse lafı uzatmayalım...

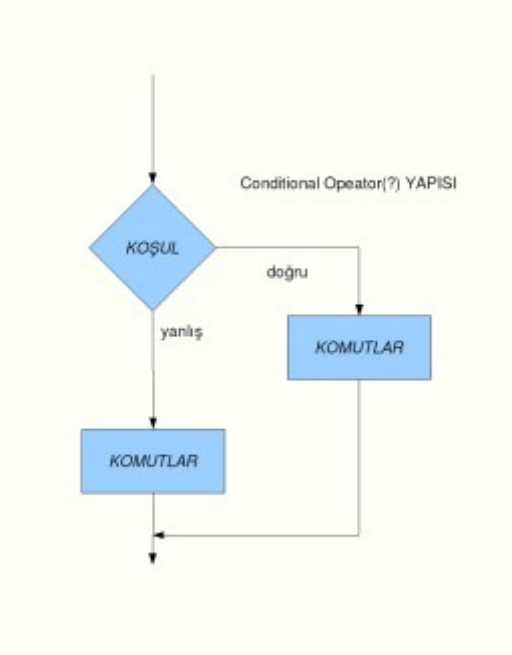
Conditional Operator, if-else ile tamamen aynı yapıdadır. Hiçbir farkı yoktur. Tek farkı koda bakıldığında anlaşılmasının biraz daha zor oluşudur. Bir de if - else gibi yazıyla ifade edilmez. Onun yerine soru işareti (?) ve iki nokta üst üste (:) kullanarak yazarız. Aşağıdaki tabloda if else yapısıyla karşılaştırılmalı olarak, Conditional Operator verilmiştir:

if-else Yapısı	*Conditional Operator (?) Yapısı*

`if( koşul ) {`
`  if_komut(lar)`
`}`
`else {`
	`koşul?if_komut(lar):else_komutlar`

```
        else_komut(lar)
}
```

Conditional Operator (?) Akış Diyagramı



Şimdi de aynı programı, hem if-else, hem de conditional operator kullanarak yazalım:

```
/*
Girilen tam sayının
10'dan büyük olup
olmadığını gösteren
program
*/
#include<stdio.h>
int main( void )
{
        int sayi;
        printf("Lütfen bir sayı giriniz> ");
        scanf("%d",&sayi);
        if( sayi > 10 )
                printf("Sayı 10'dan büyüktür\n");
        else
                printf("Sayı 10'dan küçüktür veya 10'a eşittir\n");
        return 0;
}
```

Şimdi de aynı programı conditional operator kullanarak yazalım:

```
/*
Girilen tam sayının
10'dan büyük olup
olmadığını söyleyen
program
*/
#include<stdio.h>
int main( void )
{
        int sayi;
        printf("Lütfen bir sayı giriniz> ");
        scanf("%d",&sayi);
        ( sayi > 10 ) ? printf("Sayı 10'dan büyüktür\n"):
                        printf("Sayı 10'dan küçüktür veya 10'a eşittir\n");
```

```
        return 0;
}
```

Program gördüğünüz gibi biraz daha kısaldı.

Conditional Operator'ler pek kullanmayı sevmediğim bir yapıdır. Çünkü kodun kısa olmasından çok, anlaşılabilir olması önemli. Ve conditional operator kullanmak ne yazık ki, kodu daha karmaşık hale getiriyor. UNIX filozofisinde bir şeyi akıllıca yapacağınıza, temiz/açık yapın diye bir yaklaşım mevcut. Belki bu yüzden veya belki de tembellik, conditional operator'lere alışamadım gitti... :)

Şimdi örnek sorularımıza gelelim...

Soru 1: Aşağıdaki kodu yorumlayınız:

s = (x < 0) ? -1 : x * x

Soru 2: İki tam sayı alacak ve verilecek operatöre göre (+, -, *, /, %) işlem yapacak bir program yazınız.

Soru 3: Verilecek üç sayıdan en büyüğünü ekrana yazdıracak bir program yazınız.

Cevap1:

```
Eğer x, 0'dan küçük bir değerse, s değişkenine, -1 değeri atanır. Eğer x, 0'dan
büyükse,
x'in karesi, s değişkenine atanır. (Gördüğünüz gibi ilk bakışta kodu yorumlamak
biraz
sıkıcı olabiliyor. Sanırım yukarda ne demek istediğimi anlamışsınızdır.)
```

Cevap2:

```
#include<stdio.h>
int main( void )
{
        int sayi_1, sayi_2;
        char operator_simgesi;
        printf("Lütfen işlem simgesi giriniz> ");
        scanf("%c",&operator_simgesi);
        printf("Lütfen birinci sayıyı giriniz> ");
        scanf("%d",&sayi_1);
        printf("Lütfen ikinci sayıyı giriniz> ");
        scanf("%d",&sayi_2);
        switch( operator_simgesi ) {
                case '+':
                        printf("%d %c %d = %d\n",sayi_1,operator_simgesi,
sayi_2, sayi_1 + sayi_2); break;
                case '-':
                        printf("%d %c %d = %d\n",sayi_1,operator_simgesi,
sayi_2, sayi_1 - sayi_2); break;
                case '%':
                        printf("%d %c %d = %d\n",sayi_1,operator_simgesi,
sayi_2, sayi_1 % sayi_2); break;
                case '*':
                        printf("%d %c %d = %d\n",sayi_1,operator_simgesi,
sayi_2, sayi_1 * sayi_2); break;
                case '/':
                        printf("%d %c %d = %.2f\n",sayi_1,operator_simgesi,
sayi_2, (float)sayi_1 / sayi_2);break;
                default:
                        printf("HATA: Tanımsız bir operatör girdiniz!\n");
        }
```

```
        return 0;
}
```

Cevap3:

```
#include<stdio.h>
int main( void )
{
        int sayi_1, sayi_2, sayi_3;
        printf("Üç sayı giriniz> ");
        scanf("%d %d %d", &sayi_1, &sayi_2, &sayi_3);
        if( sayi_1 >= sayi_2 && sayi_1 > sayi_3 )
                printf("%d en büyük sayıdır!\n", sayi_1);
        else if( sayi_2 >= sayi_1 && sayi_2 >= sayi_3 )
                printf("%d en büyük sayıdır!\n", sayi_2);
        else if( sayi_3 >= sayi_1 && sayi_3 >= sayi_1 )
                printf("%d en büyük sayıdır!\n", sayi_3);
        return 0;
}
```

Döngü Kavramı

Programlama konusunda -hangi dil olursa olsun- en kritik yapılardan biri döngülerdir. Döngüler, bir işi, belirlediğiniz sayıda yapan kod blokları olarak düşünülebilir. Ekrana 10 kere "Merhaba Dünya" yazan bir programda, "Merhaba Dünya" yazdıran kodu aslında tek bir defa yazarsınız, döngü burada devreye girip, sizin için bu kodu istediğiniz sayıda tekrarlar.

Döngüleri bu kadar kritik yapan unsur; iyi yazılıp, optimize edilmediği takdirde, bilgisayarınızın işlem gücünü gereksiz yere tüketmesi ve harcanan zamanı arttırmasıdır. Benzer şekilde, iyi yazılmış bir döngü, programınızı hızlı çalıştıracaktır.

Bütün döngüler temelde iki aşamayla özetlenebilir. Aşamalardan biri, döngünün devam edip etmeyeceğine karar verilen mantıksal sorgu kısmıdır. Örneğin, ekrana 10 kere "Merhaba Dünya" yazdıracaksanız, kaçıncı seferde olduğunu, koşul kısmında kontrol edersiniz. Diğer aşama, döngünün ne yapacağını yazdığınız kısımdır. Yani ekrana "Merhaba Dünya" yazılması döngünün yapacağı iştir.

Döngünün devam edip etmeyeceğine karar verilen aşamada, hatalı bir mantık sınaması koyarsanız, ya programınız hiç çalışmaz ya da sonsuza kadar çalışabilir.

C programlama diline ait bazı döngüler; while, do while, for yapılarıdır. Bunlar dışında, goto döngü elemanı olmasına rağmen, kullanılması pek tavsiye edilmemektedir.

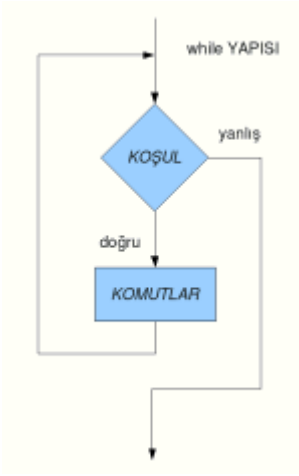
while Döngüsü

while döngüsü, en temel döngü tipimizdir. Bir kontrol ifadesiyle döngünün devam edilip edilmeyeceği kontrol edilirken, scope içinde (yani ayraç işaretleri arasında) kalan bütün alan işleme sokulur. İşleme sokulan kod kısmı döngü yapılacak adet kadar tekrar eder.

while döngüsünün genel yapısını ve akış şemasını aşağıda görebilirsiniz:

while Yapısı ***while Akış Diyagramı***

```
while( koşul ) {
        komut(lar)
}
```



Yukarda 10 kere ekrana "Merhaba Dünya" yazan programdan bahsettik. Gelin bir anlaşma yapalım ve döngülerle alakalı bütün ilk örneklerimiz bu programın nasıl yazılacağını göstersin.

while döngüsü kullanarak, ekrana 10 kere "Merhaba Dünya" yazan program aşağıdaki gibidir:

```
/*
Ekrana 10 kere "Merhaba Dünya"
yazan program
*/
#include<stdio.h>
int main( void )
{
        //i değişkenine bir başlangıç değeri atıyoruz.
        //i'ye ilk deger atanmazsa, döngümüz yanlış çalışır.
        int i = 0;
        //i'nin degeri kontrol işleminden
        //sonra 1 artar.
        while( i++ < 10 ) {
                //2d bir tam sayının yazdırılacağı
                //ancak bu sayı tek rakamdan oluşsa da
                //2 rakamlık yer ayırılmasını belirtir.
                printf("%2d: Merhaba Dünya\n",i);
        }
        return 0;
}
```

Yukardaki program aslında son derece basit. i değişkenine ilk değer olarak 0 atıyoruz. Daha sonra, while döngüsüne başlıyoruz. İfadenin doğruluğu (yani i'nin 10'dan küçük olup olmadığı) kontrol ediliyor. Eğer doğruysa, döngü içindeki kodların çalışması başlatılıyor. Elbette kodların başlamasından bir önceki adımda, i değişkeni arttırılıyor. Bu yapı toplamda 10 kere tekrar ediyor ve en sonunda i'nin degeri 10'a eşit olunca, döngü sonlandırılıyor.

$$\sum_{i=0}^{n} i^2$$

Yandaki işlem basit bir toplama ifadesidir. Yanda gördüğümüz ifade de, n değerini kullanıcıdan alacağımızı düşünerek bir program yazalım. Bu program, alacağı n değerine göre, kendisine kadar olan sayıların karelerinin toplamını gösterecektir. Bu programı yazarsak:

```
#include<stdio.h>
int main( void )
{
        int i = 0, toplam_deger = 0;
        int n;
        printf("Lütfen n değerini giriniz> ");
```

```
        scanf("%d",&n);
        while( i <= n ) {
                toplam_deger += i*i;
                i++;
        }
        printf("Sonuç: %d\n",toplam_deger);
        return 0;
}
```

do while Döngüsü

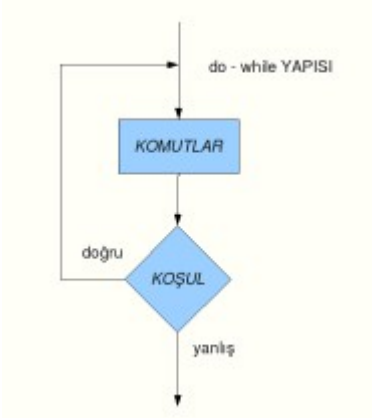
Göreceğimiz ikinci döngü çeşidi, do while döngüsüdür. Yaptığı iş, while ile hemen hemen aynıdır; verilen işi, döngü koşulu bozulana kadar sürdürür. Ancak while'a göre önemli bir farkı vardır.

while döngülerinde, döngü içersindeki işlem yapılmadan önce, sunulan koşul kontrol edilir. Şayet koşul sağlanmıyorsa, o while döngüsünün hiç çalışmama ihtimali de bulunmaktadır. do while döngülerindeyse, durum böyle değildir. İlk çalışmada koşul kontrolü yapılmaz. Dolayısıyla, her ne şartta olursa olsun, döngünüz -en azından bir kere- çalışacaktır.

Bazı durumlarda, döngü bloğu içersindeki kodların en azından bir kere çalışması gerektiğinden, do while yapısı kullanılır. do while ile ilgili genel yapıyı ve akış şemasını aşağıda bulabilirsiniz:

do while Yapısı ***do while Akış Diyagramı***

```
do {
        komut(lar)
} while( koşul );
```



Önce Merhaba Dünya örneğimizi yapalım:

```
#include<stdio.h>
int main( void )
{
        int i = 0;
        do {
                //Önce i'nin değeri arttırılıyor
                //sonra ekrana Merhaba Dünya yazdırılıyor.
                printf("%2d: Merhaba Dünya\n",++i);
        } while( i < 10 );
        return 0;
}
```

Gördüğünüz gibi, bir önceki örneğimize oldukça benzer bir yapıda, yazıldı. Tek fark i'nin değeri 0'da olsa, 1000'de olsa, en azından bir kez Merhaba Dünya'nın yazılacak olmasıdır. Ancak while'de kontrol önce yapıldığı için, hiçbir şey ekrana yazılmaz.

Şimdi do while'in kullanılmasının daha mantıklı olacağı bir program yapalım. Kullanıcıdan iki sayı alınsın. Bu iki sayı toplandıktan sonra, sonucu ekrana yazdırılsın. Yazdırma sonunda "Devam etmek istiyor musunuz?" sorusu sorulsun ve klavyeden 'E' veya 'e' karakterlerinden birisi girilirse, program

devam etsin. Yok farklı birşey girilirse, program sonlandırılsın. Örnek programımızı aşağıda bulabilirsiniz:

```
#include<stdio.h>
int main( void )
{
        int sayi_1, sayi_2;
        char devam_mi;
        do {
                printf("Birinci sayıyı giriniz> ");
                scanf("%d",&sayi_1);
                printf("İkinci sayıyı giriniz> ");
                scanf("%d",&sayi_2);
                printf("%d + %d = %d\n", sayi_1, sayi_2, sayi_1 + sayi_2);
                printf("Devam etmek ister misiniz? ");
                //C'de tek karakter okuma işlemi biraz sıkıntılı
                //olduğundan, burada da bir do while kullandık.
                do {
                        scanf("%c",&devam_mi);
                }while( devam_mi == '\n' );
                printf("\n");
        } while( devam_mi == 'E' || devam_mi == 'e' );

        return 0;
}
```

Program, kullanıcıdan iki sayı alıp, toplamını ekrana bastıktan sonra, yeniden işlem yapıp yapmak istemediğimizi sormaktadır. Bu programı while ile de yazabilirdik. Ancak while ile yazabilmek için, *devam_mi* değişkenine önceden 'E' değerini atamamız gerekmekteydi. do while döngüsündeyse, bu zorunluluğa gerek kalmamıştır.

Not: Yukardaki programda, farketmiş olduğunuz gibi karakter okumayı biraz farklı yaptık. Normalde, scanf( ) fonksiyonunu kullanmak yeterliyken, burada, işin içine bir de, do while girdi. Açıklayacak olursak, C'de karakter okumaları, biraz sıkıntılıdır. Eğer giriş tampon belleğinde (Buffer) veri bulunuyorsa, bu direkt karaktere atanır. Bundan kurtulmak için birçok yöntem olduğu gibi, uygulanabilecek bir yöntem de, yukarda yazılmış olan döngü şeklinde değer almaktır. Çünkü siz daha bir şey girmeden, ilk değer '\n' geleceğinden, döngünün ikinci çalışmasında, doğru değer atanacaktır. İlerki konularda, daha detaylı ele alacağımız bir problem olarak şimdilik önemsemeyelim. Sadece karakter okuyacağınız zaman problem çıkarsa, yukardaki gibi bir yöntem uygulanabileceğini bilmeniz -şimdilik- yeterli.

for Döngüsü

while ve do while dışında, üçüncü bir döngü tipi olarak, for yapısı bulunmaktadır. Diğer iki döngüden farklı olarak, for yapısı, yenilemeli-tekrarlamalı (İngilizce iterative) yapılarda kullanıma daha uygundur. Bunu performans anlamında söylemiyorum. Demek istediğim yazım tekniği olarak, for döngüsünün daha kullanışlı olmasıdır. Örneğin birbirini, sürekli tekrar eden işlemlerin yapıldığı Nümerik Analiz gibi alanlar, for döngüsü için iyi bir örnek olabilir. Ancak bu dediklerim sizi yanıltmasın; for döngüsü sadece soyut alanlarda çalışsın diye yaratılmış bir şey değildir.

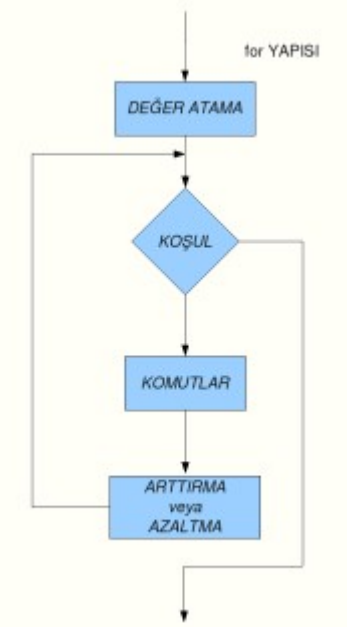
Programlarda, diğer iki döngüden çok daha fazla for kullanırsınız. Çünkü for sadece matematiksel hesaplama işlemlerinde değil, diziler (array) gibi konularda sürekli kullanılan bir yapıdır. Yazımı diğerlerine nazaran daha sade olduğundan, iteratif işlemlerde kullanılması elbette ki tesadüf olarak düşünülemez.

Aşağıda for döngüsünün genel yazımını ve akış diyagramını göreceksiniz:

for Yapısı

```
for( ilk_deger_atama; koşul; arttırma/azaltma ){
        komut(lar)
}
```

for Akış Diyagramı



İlk atacağımız adım; elbette ki ekrana 10 kere "Merhaba Dünya" yazdırmak olacak. (Umarım bu Merhaba Dünya'larla sizi fazla sıkıp, programlama işinden vazgeçirmemişimdir. Programlama mantığını kaptıktan sonra, dünyayı daha farklı görmeye başlayacak ve Merhaba Dünyalar'ın sebebini daha iyi anlayacaksınız. Ve inanın bütün bu eziyete değer...) Buyrun programımız:

```
#include<stdio.h>
int main( void )
{
        int i;
        for( i = 0 ; i < 10; i++ ) {
                printf("%2d: Merhaba Dünya\n",(i+1));
        }
        return 0;
}
```

Gördüğünüz gibi çok daha sade ve açık gözükür bir kod oldu. for altında tek satır komut olduğundan, küme parantezleri koymamız opsiyoneldi ama ne yaptığınızı karıştırmamak için, her zaman koymanızı öneririm.

for döngüleriyle ilgili bazı özel durumlarda vardır. for döngüsü içersine yazdığınız ilk değer atama, kontrol ve arttırma işlemlerini tanımlama esnasında yapmanız gerekmez. Aşağıda verilen kod, yukardakiyle tamamen aynı işi yapar. Farkı, i'nin daha önce tanımlanmış olması ve arttırma/azaltma işinin döngü içinde yapılmasıdır.

```
#include<stdio.h>
int main( void )
{
        int i;
        i = 0;
        for( ; i < 10; ) {
                printf("%2d: Merhaba Dünya\n",(i+1));
                i = i + 1;
        }
        return 0;
}
```

break Komutu

Bazı durumlarda, döngüyü aniden sonlandırmak isteriz. Bunun için 'break' komutunu kullanırız. Döngüyü aniden sonlandırmak veya döngüyü kırmak işlemini, zaten daha önce switch case'lerde kullanmıştık. Bahsetmediğimiz şey, bunun her döngü içersinde kullanılabileceğiydi.

Aşağıdaki programı inceleyelim:

```
/*
0 ile 99 arasında tesadüfi sayılar üreten
bir programın, kaçıncı seferde 61 sayısını
bulacağını yazan program aşağıdadır.
*/
#include<stdio.h>
int main( void )
{
        int i,tesadufi_sayi;
        int deneme_sayisi = 0;
        //while içinde 1 olduğundan sonsuza kadar döngü çalışır.
        while( 1 ){
                //tesadufi_sayi değişkenine, 0 ile 99 arasında
                //her seferinde farklı bir sayı atanır.
                //rand( ) fonksiyonu tesadüfi sayı atamaya yarar.
                //mod 100 işlemiyse, atanacak sayının 0 ile 99
                //arasında olmasını garantiler.
                tesadufi_sayi = rand() % 100;
                //Döngünün kaç defa çalıştığını deneme_sayisi
                //değişkeniyle buluruz.
                deneme_sayisi++;
                //Eğer tesadufi sayı 61'e eşit olursa,
                //döngü kırılıp, sonlandırılır.
                if( tesadufi_sayi == 61 ) break;
        }
        printf("Toplam deneme sayısı: %d\n",deneme_sayisi);
        return 0;
}
```

Program için koyulmuş açıklamalar (comment) zaten neyin n'olduğunu açıklıyor. Kısaca bir şeyler eklemek gerekirse, bitişinin nerede olacağını bilmediğimiz bir döngüyü ancak, break komutuyla sonlandırabiliriz. Şartlar sağlandığında, break komutu devreye girer ve döngü sonlandırılır. Bunun gibi bir çok örnek yaratmak mümkündür.

continue Komutu

break komutunun, döngüyü kırmak için olduğundan bahsetmiştik. Bunun dışında işlem yapmadan döngüyü devam ettirmek gibi durumlara da ihtiyacımız vardır. Bunun içinde continue (Türkçe: devam) komutunu kullanırız.

```
/*
Sadece tek sayıları yazdıran bir
program
*/
#include<stdio.h>
int main( void )
{
        int i;
        for( i = 0; i < 10; i++ ) {
                //i değişkeninin 2'ye göre modu
                //0 sonucunu veriyorsa, bu onun
                //bir çift sayı olduğunu gösterir.
                //Bu durumda ekrana yazdırılmaması
                //için döngü bir sonraki adıma geçer.
```

```
                if( i%2 == 0 ) continue;
                printf("%2d\n",i);
        }
        return 0;
}
```

0 ile 10 arasındaki tek sayıları gösteren program örneğini yukarda görebilirsiniz. Elbette ki bu işi daha farklı ve daha iyi yapan bir program yazabilirdik. Ama şimdilik continue komutunun nasıl kullanıldığını inceleyelim.

Program bir for döngüsü çalıştırmaktadır. Her defasında i değişkenin 2'ye göre modu alınır. Eğer sonuç 0'sa, bu sayının çift olduğunu gösterir. Dolayısıyla, bunun ekrana yazdırılmaması gerekir. Bu yüzden, döngü içersindeki işlemleri sürdürmek yerine, altta kalan kodları atlarız. Burada continue komutu kullanılır ve kullanıldığı noktadan itibaren olan işlemler yapılmaz. Döngü başa döner, aynı işlemleri yapar. Bu sefer i tek sayı olacağından continue komutu çalışmaz ve sayıyı ekrana bastırırız.

goto Yapısı

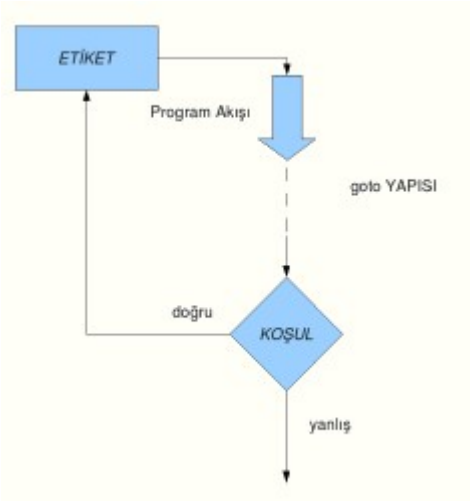
C programlama dilinde bulunan bir başka yapı, goto deyimidir. Koyacağınız etiketler sayesinde, programın bir noktasından bir başka noktasına atlamanızı sağlar. goto, bir döngü değildir ancak döngü olarak kullanılabilir.

goto, çalışabilmek için etiketlere ihtiyaç duyar. Etiketler, vereceğiniz herhangi bir isme sahip olabilir. Etiket oluşturmak için bütün yapmanız gereken; etiket adını belirleyip, sonuna iki nokta üst üste eklemek (:) ve programın herhangi bir yerine bunu yazmaktır. goto deyimi kullanarak bu etiketleri çağırırsanız, etiketin altında bulunan kodlardan devam edilir. goto ve etiketlere dair genel yapıyı, akış diyagramıyla birlikte aşağıda bulabilirsiniz:

goto Yapısı

```
label_name:
        .
        .
        .

if( kosul ) {
        goto label_name
}
        .
        .
        .
```

goto Akış Diyagramı



NOT: goto deyimi tek başına da kullanılabilir. Fakat mantıksal bir sınama olmadan, goto yapısını kullanmanız, sonsuz döngüye neden olacaktır.

Şimdi goto ifadesiyle basit bir döngü örneği oluşturalım. Önceki seferlerde olduğu gibi ekrana 10 defa "*Merhaba Dünya*" yazdıralım:

```
#include<stdio.h>
int main( void )
{
        int i = 0;
        // baslangic_noktasi adinda bir etiket koyuyoruz.
        // i degiskeni 10 degerine ulasmadigi surece,
```

```
        // program buraya donecektir.
        baslangic_noktasi:
        printf( "Merhaba Dünya\n" );
        // i degerini arttiriyoruz.
        i++;
        // i degeri kontrol ediliyor. Sayet 10'dan kucukse,
        // en basa donuyor.
        if( i<10 ) goto baslangic_noktasi;

        return 0;
}
```

İstediğiniz sayıda etiket koyup, goto kullanarak, programın herhangi bir noktasına ulaşabilirsiniz. Programınız, etiket altında kalan kısımdan itibaren çalışır. goto yapısıyla gelen esneklik, ilk bakışta oldukça güzel görünüyor. Ancak goto için birçok kaynak, "ya hiç kullanmayın ya da olabildiğince az kullanın" demektedir.

Okunup, anlaşılması zor ve üzerinde çalışılması güç bir koddan, herkesin uzak durması gerekir. İngilizce'de, karman çorman koda, "*spagetti kod*" adı verilmiştir. goto deyimi, kodunuzun spagetti koda dönüşmesine neden olur. Çünkü program akışının takibini zorlaştırıp, kodun okunabilirliğini azaltır. Diliyorsanız, goto deyimini kullanabilirsiniz. Ama zorunlu kalmadıkça kaçınmak en iyisi...

Sayı Tabanları

Bilgisayar programlamayla, matematik arasında çok güçlü bir ilişki vardır. Geçmişe bakarsanız, bilgisayar alanında önemli adımların, hep matematik kökenli insanlar tarafından atıldığını görürsünüz. Bir bilgisayar programcısı için, matematikten uzak durmak düşünülemez.

Bugün ki dersimizde, biraz matematik içersine gireceğiz ve sayı sistemleriyle, Boole Cebiri (Boolean Algebra) konularını ele alacağız.

Genel kabul görmüş sayı sistemleri vardır ve içlerinde en yaygını, hepimizin gündelik hayatta kullandığı 10'luk sayı sistemidir. Yazması, okunması ve işlem yapması son derece kolay olduğundan bunu daha çocuk yaşta öğrenir ve bu şekilde sürdürürüz. Ancak bilgisayarlar bizim gibi işlem yapabilme yetisine sahip değildir. Onlar için iki ihtimal vardır. Bir şey ya 1'dir ya da 0. Bunu ikilik sayı sistemi olarak adlandırırız. Yani bizim yazdığımız bütün sayılar, bütün harfler ve aklınıza gelen-gelmeyen bütün işaretler, bilgisayar için 0 ve 1'in kombinasyonlarından ibarettir. İşte bu yüzden bizlerin, ikilik sayı sistemine hakim olması gerekir.

Sayı sistemlerini genel olarak aşağıdaki gibi ifade edebiliriz:

$$\sum_{k=0}^{n} d_k . N^k = (d_n . N^k) + (d_{(n-1)} . N^{(k-1)}) + ... + (d_1 . N^1) + (d_0 . N^0)$$

Burada, N sayı tabanına göstermektedir. k sayının hangi hanesinde olduğumuzu ifade ederken, d_k ise, ilgili sayıdaki rakamı temsil eder. Şimdi basit bir örnek yapalım ve ikilik tabandaki 10011 sayısının, 10 tabanındaki eş değerini bulalım:

$$(d_4d_3d_2d_1d_0)_2 = (d_0 . 2^0) + (d_1 . 2^1) + (d_2 . 2^2) + (d_3 . 2^3) + (d_4 . 2^4)$$

$$(10011)_2 = (1 . 2^0) + (1 . 2^1) + (0 . 2^2) + (0 . 2^3) + (1 . 2^4) = 19$$

ikilik sayı sistemi dışında, 16'lık (Hexadecimal) sayı sistemi de oldukça önemli bir başka tabandır. 16'lık sayı sisteminde, rakamları ifade etmek için 16 adet sembole gereksinim duyarız. Bu yüzden 0 ile 9 arasında olan 10 rakamı kullandıktan sonra, A, B, C, D, E ve F harflerini de rakam olarak

değerlendiririz.

Decimal	**:**	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal	**:**	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Hexadecimal/16'lık sayı tabanıyla ilgili aşağıdaki örneklere göz atalım:

$(3FC)_{16} = (3 . 16^2) + (F . 16^1) + (C . 16^0) = 768 + 240 + 12 = 1020$

$(1FA9)_{16} = (1 . 16^3) + (F . 16^2) + (A . 16^1) + (9 . 16^0) = 4096 + 3840 + 160 + 9 = 8105$

$(75)_{16} = (7 . 16^1) + (7 . 16^0) = 112 + 5 = 117$

16'lık sayı sisteminin diğer bir ismi Hexadecimal olduğundan, bazı yerlerde, bunu ifade etmek için 16 yerine 'H' harfi de kullanılabilir:

$(BA3)_{16} = (BA3)_H$; $(A1E)_{16} = (A1E)_H$ gibi...

Tabanlar arasında dönüştürme işlemi, üzerinde duracağımız bir başka konudur. Elinizde 16'lık sayı sisteminde bir sayı varsa ve bunu 2'lik sayı sistemiyle yazmak isterseniz önce 10'luk sayı sistemine çevirip daha sonra 2'lik sayı sistemine dönüştürebilirsiniz. Ancak 16'lık ve 2'lik sayı sistemlerini çok daha kolay birbirine dönüştürmeniz mümkündür. Aşağıdaki tabloda 16'lık sayı sistemindeki rakamlar ve bunun 2'lik sayı sistemindeki karşılığı verilmiştir:

Hexadecimal	**:**	0	1	2	3	4	5	6	7	8	9	A	B	C
Binary (İkilik)	**:**	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100

Bu durumu bir örnekle şöyle gösterebiliriz:

$(A3F1)_H$	**:**	A	3	F	1
	:	1010	0011	1111	0001

16'lık tabandaki her rakamın, 2'lik tabandaki karşılığını koyduğumuzda yukardaki eşitliği elde ediyoruz ve buna göre $(A3F1) = (1010\ 0011\ 1111\ 0001)_2$ eşitliğini kurabiliyoruz. (2'lik tabandaki sayıya ait boşluklar, sayının daha rahat okunması için bırakılmıştır.) Bu tarz dönüşümler, 2 ve 2'nin katında olan sayı tabanlarında rahatlıkla yapılabilir.

Hatırlarsanız, değişken tiplerinde, işaretli ve işaretsiz değişken tanımlamalarından bahsetmiştik. Şimdi olayın biraz daha derinine inelim. Bir char, 1 byte alan kaplar ve 1 byte, 8 bit'ten oluşur. Aşağıdaki kutuların her birini bir bit ve kutuların oluşturduğu bütünü bir byte olarak düşünün:

a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

Yukardaki kutuların toplamı olan bir byte, char değişkeninin kapladığı alanı temsil etmektedir. Pozitif sayıları ifade etmeyi zaten öğrenmiştik. Sayının karşılığını, 2'lik tabanda yazarak, gerekli sonuca ulaşırız. Ancak sayımız Negatif değerliyse, işler biraz farklılaşır. Burada devreye işaret biti (sign bit) devreye girer. Yukardaki şekilde, diğer kutulardan farklı renkte olan a_7 işaret bitidir.

Özetle, a_7 0 ise, sayı pozitiftir. Eğer a_7 1 ise, sayı negatiftir.

İkilik tabandaki işaretli bir sayının, 10'luk tabandaki karşılığını şu şekilde bulabiliriz:

$(a_7a_6a_5a_4a_3a_2a_1a_0)_2 = (a_7 . -2^7) + (a_6 . 2^6) + ... + (a_1 . 2^1) + (a_0 . 2^0)$

İkilik tabanda yazılmış (10000011)$_2$ sayısı, işaretsiz olarak düşünülürse, 131'e eşittir. Ancak işaretli bir sayı olduğu düşünülürse, karşılığı, -125 olacaktır. Konunun pekişmesi açısından aşağıdaki örneklere göz atabilirsiniz:

```
* ( 1011 1011 )2 = -69 (Sayı işaretliyse)
  ( 1011 1011 )2 = 187 (Sayı işaretsizse)
* ( 1100 1101 )2 = -51 (Sayı işaretliyse)
  ( 1100 1101 )2 = 205 (Sayı işaretsizse)
* ( 0110 1101 )2 = 109 (Sayı işaretliyse)
  ( 0110 1101 )2 = 109 (Sayı işaretsizse)
```

Negatif bir sayının 2'lik tabandaki karşılığını bulmak için, önce (*i*) sayıyı pozitif olarak ikilik tabanda yazarız. Daha sonra, (*ii*) ikilik tabanda yazılmış sayının 1 yazan rakamları 0, 0 yazan rakamları 1'e çevrilir. Son olarak (*iii*) çıkan sayıya, 1 eklenir. Bu size istediğiniz sayının ikilik tabanındaki eşini verecektir. Şimdi bir uygulama yapalım ve -7 sayını ikilik tabana çevirmeye çalışalım:

i) -7 ==> (7)$_{10}$ = (0000 0111)$_2$

ii) (0000 0111) ==> (1111 1000)

iii) (1111 1000) + 1 = (1111 1001) ==> (-7)$_{10}$ = (1111 1001)$_2$

Bit Bazında (Bitwise) Operatörler

Bit bazında operatörlerin, İngilizce'deki karşılığı Bitwise Operators (yani Bit bit Operatörler) olarak geçmektedir. Bit bazında operatörler, ikilik sayı tabanında yapabileceğimiz işlemleri temsil eder. Kullanılan operatörleri aşağıda inceleyeceğiz.

AND (&) Operatörü

AND operatörü, kendisine verilen iki değişkenin bütün bitleri 1'e eşit olduğu takdirde, geriye 1 döndürür. Aksi halde -yani en ufak bir fark varsa- 0 değeri dönmektedir.

p	q	p&q
0	0	0
0	1	0
1	0	0
1	1	1
x	0	0
x	1	x

Şimdi, AND (&) operatörünü 25 ve 14 sayılarını karşılaştırmak için kullanalım:

```
  25 ==> ( 0001 1001 )2
  14 ==> ( 0000 1110 )2
&
----------------------------
   8 ==> ( 0000 1000)2
```

OR (|) Operatörü

İki değişkenden herhangi biri 1 içeriyorsa, geriye 1 döner. Eğer her ikisi de 0 içeriyorsa, geriye 0 dönmektedir.

p	q	p|q
0	0	0
0	1	1
1	0	1
1	1	1
x	0	x
x	1	1

Daha önce kullandığımız 25 ve 14 sayıları üzerinde OR (|) işlemi kullanalım:

```
  25 ==> ( 0001 1001 )2
  14 ==> ( 0000 1110 )2
|
----------------------------
  31 ==> ( 0001 1111)2
```

ÖNEMLİ NOT: Bit bazında kullanılan, AND(&) ve OR (|) operatörleri, koşullu ifadelerde kullanılan, AND(&&) ve OR (||) ifadelerinden farklıdır. Şayet, & veya | bir koşullu ifade gibi kullanmaya kalkarsanız, işlem yine yapılacaktır. Ancak bunu yapmanız tavsiye edilmez. İlerki konularımızda neden uygun olmadığı, short-circuit ile ilgili bilgi verilirken açıklanacaktır.

NOT (~) Operatörü

NOT (~) Operatörü, kendisine verilen sayıya ait bit'leri tam tersine çevirir. Yani 0 gördüğü yeri 1; 1 gördüğü yeri 0 yapar.

p	~p
0	1
1	0

```
  25 ==> ( 0001 1001 )2
~
----------------------------
 230 ==> ( 1110 0110 )2
```

XOR (^) Operatörü

XOR (Exclusive OR) Operatörü, sadece ve sadece karşılaştırma yapılan bitlerden biri, 1 değerine sahipse, geriye 1 döndürür. Eğer karşılaştırma yapılan bit'lerden her ikisi de 0 veya 1'se, o zaman sonuç 0 olarak çıkar.

p	q	p^q
0	0	0
0	1	1
1	0	1
1	1	0
x	0	x
x	1	~x

```
  25 ==> ( 0001 1001 )2
  14 ==> ( 0000 1110 )2
^
--------------------------
  23 ==> ( 0001 0111 )2
```

Kaydırma (Shift) Operatörleri

Kaydırma operatörleri, özellikle Assembly ile uğraşanlara tanıdık gelecektir. Bunları kullanarak son derece hızlı çarpma ve bölme yapılabilir. C'deyse benzer amaçlarla kullanmanız elbette mümkündür. İki çeşit kaydırma operatörü vardır:

i) Sola Kaydırma - Shift Left (<<)

ii) Sağa Kaydırma - Shift Right (>>)

Her iki durumda da genel kullanım şekli aşağıdaki gibidir:

[*Tam Sayı*][*Operatör*][*Kaydırma Adım Sayısı*]

Aşağıdaki örnek, sola kaydırma operatörü kullanılarak yapılan bir işlemi göstermektedir. x değişkeni, 10 tabanında 22 sayısını tutmaktadır. 2 adım sola kaydırılması sonucu, sayı 88 olmuş ve y'ye atanmıştır.

```
x = ( 0001 0110 )2 ==> 22
y = x << 2
y = ( 0101 1000 )2 ==> 88
```

Operatör Öncelikleri

Hatırlarsanız, aritmetik işlemlerde önceliklerin olduğunu ve örneğin çarpmanın, toplamadan daha önce yapılacağını anlatmıştık. Benzer bir durum, operatörler içinde geçerlidir. Altta bulunan tabloda, hangi operatörün daha önce işleme alındığını bulabilirsiniz:

OPERATÖR ÖNCELİK SIRASI

DÜŞÜK	|	^	&	<< >>	+ -	* / %	! ~ - ++ --	( )	**YÜKSEK**

Aşağıda bulunan tablo, ilişkisel ve mantıksal operatörlerde ki öncelik sırasını göstermektedir:

İLİŞKİSEL ve MANTIKSAL OPERATÖR ÖNCELİK SIRASI

DÜŞÜK	||	&&	== !=	> >= < <=	!	YÜKSEK

Yukardaki tablolarda, aynı hücrede olan operatörlerin işlem öncelikleri aynıdır. Önce hangisi yazılmışsa, ilk olarak o dikkate alınır. Ama bunun dışında tanınan bir işlem önceliği bulunmamaktadır.

Aşağıdaki örnek, operatör önceliklerini pekiştirmek açısından incelenebilir:

```
7 & 13 ^ 11 % 4 * 2 << 14 / 4
==> 7 & 13 ^ 6 << 3
==> 5 ^ 48 = 53
```

Şimdi de benzer bir örneği, C programı yazarak yapalım:

```
#include<stdio.h>
int main( void )
{
        printf( "İşlem Sonucu: %d\n", 117 & 11 << 2 * 3 );
        return 0;
}
```

Yukardaki program, 64 sonucunu vermektedir. Programı çalıştırdığınızda, ikilik düzeyde işlemler yapılacak ve 64 cevabına ulaşılacaktır. Siz de hesaplayarak, aynı yanıtı bulabilirsiniz.

Bir sonraki dersimizde, konu anlatımı olmayacak. Onun yerine bol bol örnek yapıp, şimdiye kadar işlediğimiz konuların üzerinden geçeceğiz. Görüşmek üzere...

Örnek Sorular

Şimdiye kadar işlemiş olduğumuz dersleri pekiştirmek için, bu yazıda örnek sorular çözeceğiz. Soruların hazırlanmasında, arkadaşım Sunay SÖNMEZ'in büyük emeği geçmiştir. Kendisine teşekkür ederim.

Soru 1.a: Aşağıdaki programın çıktısı nedir?

```
#include<stdio.h>
int main( void )
{
        int i = 0;
        for( ; i < 12; i++ ) {
                if( ( i++ + 1 ) == 5 )
                        printf( "%d\n",i );
                else
                        if( i % 2 == 0 ) break;
                        else if( i % 3 == 0)
                                continue;
                        else
                                printf("%d\n", i );
        }
        return 0;
}
```

```
1


5
7
11
```

Soru 1.b: Aşağıdaki programı doğru olarak y

```
#include<stdio.h>
int main( void )
{
        int sum;
        float average;
        int weight_1, weight_2;
        printf( "1.agirlik> " );
        scanf( "%d", &weight_1 );
        sum += weight_1;
        printf( "2.agirlik> " );
        scanf( "%d", weight_2 );
        sum += weight_2;
        average = sum / 2;
        printf( "Ortalama: %.2f\n",
        return 0;
}
```

```
#include<stdio.h>
int main( void )
{
        int sum = 0;
        float average;
        int weight_1, weight_2;
        printf( "1.agirlik> " );
        scanf( "%d", &weight_1 );
        sum += weight_1;
        printf( "2.agirlik> " );
```

```
                                        scanf( "%d",&weight_2 );
                                        sum += weight_2;
                                        average = sum / 2.0;
                                        printf( "Ortalama: %.2f\n",
                                        return 0;
                                }
```

Soru 2: Bir üçgende, iki kenarın toplam uzunluğu, üçüncü kenardan az olamaz. Ayrıca iki kenarın birbirinden farkının mutlak değeri, üçüncü kenardan büyük olmamalıdır. Bu bilgileri kullanarak, verilen üç kenar uzunluğuna göre bir üçgen çizilip çizilmeyeceğini gösteren programı yazınız. Girilecek kenar uzunlukları tam sayı olacaktır.

Cevap 2:

```
/* Girilen 3 tamsayinin bir ucgenin kenari olup olmadigi kontrol edilir */

#include<stdio.h>
int main( void )
{
        int a, b, c;
        int temp;

        printf("Birinci kenar uzunlugu> ");
        scanf("%d", &a);
        printf("Ikinci kenar uzunlugu> ");
        scanf("%d", &b);
        printf("Ucuncu kenar uzunlugu> ");
        scanf("%d", &c);

        /* a ile b den buyuk olan a ya kucuk olan b ye atanir */
        if(a < b) {
                temp = a;
                a = b;
                b = temp;
        }

        if( ((a + b) < c) || ((a - b) > c) )
                printf("Bu kenar uzunluklarina sahip bir ucgen olamaz.\n");
        else
                printf("Bu kenar uzunluklarina sahip bir ucgen cizilebilir.\n");

        return 0;
}
```

Soru 3: Klavyeden girilecek bir sayının asal sayı olup olmadığını ekrana basan bir program yazınız.

Cevap 3:

```
/*
Bir sayının asal olup olmadığını bulmak için çeşitli metodlar
vardır. Aşağıda bu metodlardan basit bir tanesi yazılmıştır.
Eğer sayının yarısına kadar kontrol etmek yerine, karekköüne
kadar olan sayıları test ederseniz, yine aynı sonuç çıkacaktır.
Ancak anlaşılma konusunda sorun olmaması için soru bu şekilde
çözülmüştür.
*/
#include<stdio.h>
int main( void )
```

```
{
        int sayi, i;
        //Sayıyı ilk başta asal kabul ediyoruz.
        //asal_mi değişkeni 1 ise, sayi asaldir.
        int asal_mi = 1;

        //Klavyeden, test edilmek üzere bir sayı alınıyor.
        printf( "Bir sayı giriniz> " );
        scanf("%d",&sayi);

        //Girilen sayının, başka sayılara göre sırayla modunu
        //alıyoruz. Bir sayının modunu aldığınızda, kalan 0 ise
        //bu sayının bölünebildiğine ve dolayısıyla
        //asal olmadığına dair bilgi verir. Bu işlemi yapabilmek
        //için 2'den başlayarak, sayının yarısına kadar olan
        //bütün değerler deneniyor.
        for( i = 2; i <= sayi/2; i++ ) {
                if( sayi%i == 0 ) {
                //Sayı i değişkenine kalansız bölünmektedir.
                //Dolayısıyla, sayı asal değildir ve döngüyü
                //sürdürmeye gerek yoktur.
                        asal_mi = 0;
                        break;
                }
        }
        //Sayının asal olup olmama durumuna göre, çıktı yazdırılıyor.
        if( asal_mi == 1 )
                printf( "%d sayısı asaldır.\n", sayi );
        else
                printf( "%d sayısı asal değildir.\n", sayi );

        return 0;
}
```

Soru 4: Klavyeden girilecek 0 ile 999 arasında bir tam sayının, yazıyla ifade eden (örneğin 49, 'kırkdokuz' gibi) bir program oluşturunuz.

Cevap 4:

```
/*
Girilen sayının, nasıl okunduğunu ekrana yazdıran programdır.
Fonksiyon kullanımıyla daha efektif olarak yazılabilir.
Ancak henüz işlenmeyen bir konu olduğundan, basit bir şekilde
yazılmıştır.
*/
#include<stdio.h>
int main( void )
{
        int sayi;

        //Klavyeden girilecek sayinin 0 ile 999 sınırlarında
        //olup olmadığı kontrol ediliyor. Eğer değilse,
        //uyarı verilip, yeni bir sayı isteniyor. Bu işlem
        //doğru bir değer girilene kadar devam ediyor.
        do {
                printf( "Sayıyı giriniz> " );
                scanf("%d",&sayi);
                if( sayi>999 || sayi<0 )
                        printf("Girdiğiniz sayı uygun değildir.\n"
                                "0 ile 999 arasında olmalıdır.\n");
                else
                        break;
```

```
	}while( 1 );

	printf( "%d sayısı, '",sayi );

	//Verilen sayıyı, yazıyla ifade edebilmemiz için,
	//yüzler, onlar ve birler basamaklarına ayırmamız
	//gerekmektedir.

	//Sayının yüzler basamağı ayrıştırılıyor.
	//Bunun için sayıyı, 100'e bölmek yeterli.

	//YÜZLER BASAMAĞI:
	switch( sayi/100 ) {
		case 9: printf( "dokuzyüz" );   break;
		case 8: printf( "sekizyüz" );   break;
		case 7: printf( "yediyüz" );    break;
		case 6: printf( "altıyüz" );    break;
		case 5: printf( "beşyüz" );     break;
		case 4: printf( "dörtyüz" );    break;
		case 3: printf( "üçyüz" );      break;
		case 2: printf( "ikiyüz" );     break;
		case 1: printf( "yüz" );        break;
	}

	//Onlar basamağını bulmak için, sayının
	//yüze bölümünden kalan değeri, 10'a
	//bölüyoruz. Yüzler basamağını bir önceki
	//adımda ele aldığımız için, bu adımda,
	//sayının yüze bölümünü değil, bölümünden
	//kalanı kullandık.

	//ONLAR BASAMAĞI:
	switch ( (sayi%100)/10 ) {
		case 9: printf( "doksan" );     break;
		case 8: printf( "seksen" );     break;
		case 7: printf( "yetmiş" );     break;
		case 6: printf( "altmış" );     break;
		case 5: printf( "elli" );       break;
		case 4: printf( "kırk" );       break;
		case 3: printf( "otuz" );       break;
		case 2: printf( "yirmi" );      break;
		case 1: printf( "on" );         break;
	}

	//Birler basamağını bulabilmek için, sayının
	//10'a bölümünden kalana bakıyoruz.

	//BİRLER BASAMAĞI:
	switch ( (sayi%10) ) {
		case 9: printf( "dokuz" );      break;
		case 8: printf( "sekiz" );      break;
		case 7: printf( "yedi" );       break;
		case 6: printf( "altı" );       break;
		case 5: printf( "beş" );        break;
		case 4: printf( "dört" );       break;
		case 3: printf( "üç" );         break;
		case 2: printf( "iki" );        break;
		case 1: printf( "bir" );        break;
	}

	//Eğer sayı 0 ise, yukarda ki düzeneğe
	//uymayacağından, onu ayrıca ele alıyoruz.
	if( sayi == 0 )
		printf( "sıfır" );
```

```
        printf( "' şeklinde okunur.\n" );

        return 0;

}
```

Soru 5: Dört basamaklı bir sayının rakamları ters yazılıp, 4 ile çarpılırsa, çıkan sonuç, kendisiyle aynıdır. Dört basamaklı bu sayıyı bulunuz.

Cevap 5:

```
/*
Dört basamaklı bir sayının rakamları, ters yazılıp, 4 ile
çarpılırsa, sayının kendisine eşit olmaktadır. Bu program,
bahsedilen dört basamaklı bu sayıyı bulmaktadır.
*/
#include<stdio.h>
int main( void )
{
        //Dört basamaklı bir sayının tersinin dört katına
        //eşit olmasını hesaplamamız gerektiğinden, her
        //basamağı ayrı ayrı düşünmemiz gerekmektedir.
        //Bilinmeyen bu sayıyı, abcd olarak düşünebiliriz.
        int a,b,c,d;
        int sayi, sayinin_tersi;
        for( a = 0; a < 10; a++ )
                for( b = 0; b < 10; b++ )
                        for( c = 0; c < 10; c++ )
                                for( d = 0; d < 10; d++ ) {
                                        //Burada sayinin degeri bulunmaktadir.
                                        sayi = a * 1000 + b * 100 + c * 10 + d;

                                        //Burada da sayinin tersi
hesaplanmaktadir.
                                        sayinin_tersi = d * 1000 + c * 100 + b *
10 + a;

                                        //Sayinin tersinin 4 katinin ilk sayiya
                                        //esit olup olmadigi kontrol ediliyor.
                                        //Ayrica sayinin 0 olmamasi
gerekmektedir.
                                        //Bu kontrol de yapilmaktadir.
                                        if( sayi == 4 * sayinin_tersi && sayi !=
0 ) {

                                                //Sayi ve uygun olan tersi
gosterilir:
                                                printf( "Sayı: %d, tersi: %d\n",
sayi, sayinin_tersi );

                                                //return ifadesi programin daha
fazla devam
                                                //etmemesi icin burada
kullanilmistir.
                                                //Program donguye devam etmez ve
burada kesilir.
                                                //return, main fonksiyonunu
sonlandirir.
                                                return 0;
                                        }
                                }
```

```
        //Uygun bir sayi bulunamazsa, program burada sonlanacaktir.
        return 0;
}
```

Soru 6: Fibonacci serisinin ilk iki elemanı 0 ve 1'dir. Bundan sonra gelen sayılar, kendisinden önceki iki sayının toplamıyla bulunur. Yani seri elemanları 0 1 1 2 3 5 8 13 21 ... şeklinde gitmektedir. Verilecek adım sayısına göre, Fibonnaci serisinin elemanlarını gösterecek bir program yazınız.

Cevap 6:

```
/* Fibonacci sayilarini ekrana yazar */

#include<stdio.h>int main( void )
{
        int a = 0; /* a[n]   */
        int b = 1; /* a[n+1] */
        int c;     /* a[n+2] */
        int n;
        int i;

        printf("Fibonacci serisi kacinci elemana kadar yazilsin> ");
        scanf("%d", &n);

        for( i = 1; i <= n ; i++ ) {
                printf( "%4d. Eleman: %d\n", i, a );
                c = a + b;
                a = b;
                b = c;
        }

        return 0;
}
```

Soru 7: Verilecek kenar uzunluğuna göre, yıldız (*) işareti kullanarak kare çizen, bir program yazınız.

Cevap 7:

```
/* '*'ler kullanarak kenar uzunlugu verilen kareyi cizer */

#include<stdio.h>
int main( void )
{
        int kenar;
        int i;
        int j;

        printf("Karenin kenar uzunlugu: ");
        scanf("%d", &kenar);

        /* Gecerli bir deger mi */
        while((kenar < 0) || (kenar > 20)) {

                printf("Lutfen 0 ile 20 arasinda bi deger giriniz.");
                printf("Karenin kenar uzunlugu");
                scanf("%d", &kenar);
        }

        /* karenin cizilmesi */
        for(i = 1; i <= kenar; i++) {
```

```
                if(kenar == 0)
                        break;

                /* alt ve üst kenarlarin cizimi */
                if((i == 1) || (i == kenar)) {

                        for(j = 1; j <= kenar; j++)
                                printf("*");
                        printf("\n");
                        continue;
                } /* if sonu */

                /* sag ve sol kenarlarin cizimi */
                for(j = 1; j <= kenar; j++)
                        if((j == 1) || (j == kenar))
                                printf("*");
                        else
                                printf(" ");

                printf("\n");
        } /* for sonu */

        return 0;
}
```

Soru 8: Aşağıdaki eşkenar dörtgen çıktısını üretecek bir program yazınız:

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

Cevap 8:

```
/* '*'ler yardimiyla eskenar dortgen cizer */

#include<stdio.h>
int main( void )
{
        int i, j;

        for(i = 1; i <= 5; i++) {
                for(j = 1; j <= 9; j++)
                        if((j <= (9 - (2*i - 1))/2) || (j > (i + 4)))
                                printf(" ");
                        else
                                printf("*");

                printf("\n");
        }
        for(i = 4; i >= 1; i--){
                for(j = 1; j <= 9; j++)
                        if((j <= (9 - (2*i - 1))/2) || (j > (i + 4)))
                                printf(" ");
                        else
                                printf("*");
```

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

        return 0;
}
```

Soru 9: Hipotenus'u 500 birime kadar olan dik üçgenlerin, kenar uzunluklarını gösteren bir program yazınız. Kenar uzunlukları, sadece tam sayı olacaktır.

Cevap 9:

```
/* Hipotenusu 500'e kadar olan pisagor uclulerini ekrana yazar. */

#include<stdio.h>
int main( void )
{
        int a; /* birinci dik kenar */
        int b; /* ikinci dik kenar */
        int hipotenus;

        for(a = 1; a < 500; a++)
                for(b = a; b < 500; b++)
                        for(hipotenus = b+1; hipotenus <= 500; hipotenus++)
                                if( (a*a + b*b) == hipotenus*hipotenus )
                                        printf("%5d%5d%5d\n", a, b, hipotenus);
        return 0;
}
```

Kısa Devre Değerlendirme

Kısa devre değerlendirme, ne yazık ki pek iyi bir çeviri olmadı ve bu yüzden hiçbir anlam ifade etmeyebilir. İngilizce'de, Short Circuit Evaluation olarak geçen bu konu, mantıksal ifadelerle ilgilidir.

Hatırlarsanız, daha önce ki derslerimizde iki farklı AND ve OR operatörü görmüştük. Bu yapılardan biri AND için && işaretini kullanıyorken, diğeri sadece tek & simgesini kullanıyordu. Keza, OR ifadesi bir yerde, || şeklinde gösteriliyorken, diğer yerde, tek bir | simgesiyle ifade edilmekteydi. Bu işaretler, aynı sonucu üretiyor gibi görünseler de, farklı şekilde çalışırlar.

Çift şekilde yazılan operatörler, (yani && ve ||) kısa devre operatörleridir. İngilizce, Short Circuit Operator olarak isimlendirilirler. Tek sembolle yazılan operatörlerden farkı, işlemleri kısaltmalarıdır.

Bir koşul içersinde AND (&&) operatörü kullandığınızda, koşulun sol tarafı yanlışsa, sağ tarafı kontrol edilmez. Çünkü artık sağ tarafın doğru veya yanlış olmasının önemi yoktur; sonuç her şekilde yanlış olacaktır.

Benzer bir mantık OR (||) operatörü içinde geçerlidir. Eğer sol taraf doğruysa, sağ tarafın kontrol edilmesine gerek yoktur. Çünkü OR operatöründe taraflardan birinin doğru olması durumunda, diğerinin önemi kalmaz ve sonuç doğru döner.

Aşağıdaki örnekleri inceleyelim:

&& Operatörü

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

& Operatörü

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

```
        int i, j;
        i = 0;
        j = 5;
        if( i == 1 && j++ ) {
                printf( "if içersine girdi\n" );
        }
        else {
                printf( "if içersine girmedi\n" );
                printf( "i: %d, j: %d\n", i, j );
        }
        return 0;
}

if içersine girmedi
i: 0, j: 5
```

```
        int i, j;
        i = 0;
        j = 5;
        if( i == 1 & j++ ) {
                printf( "if içersine
        }
        else {
                printf( "if içersine
                printf( "i: %d, j: %d
        }
        return 0;
}

if içersine girmedi
i: 0, j: 6
```

Gördüğünüz gibi, program çıktıları birbirinden farklıdır. Bunun sebebi, ilk örnekte, *i == 1* koşulu yanlış olduğu için, && operatörünün ifadenin sağ tarafına bakmamasıdır. İkinci örnekteyse, & operatörü, koşulun her iki tarafına da bakar. Bu yüzden, j değişkenine ait değer değişir. Benzer bir uygulamayı, OR için || ve | kullanarak yapabilirsiniz.

ÖNEMLİ NOT: Özetle işlemlerinizi hızlandırmak istiyorsanız; AND kullanacağınız zaman, && operatörüyle çalışın ve yanlış olması muhtemel olan koşulu sol tarafa koyun. Eğer OR operatörü kullanacaksanız, doğru olma ihtimali fazla olan koşulu, ifadenin soluna koyun ve operatör olarak || ile çalışın. Bu şekillde yazılan bir program, kısa devre operatörleri sayesinde, gereksiz kontrolden kaçınarak işlemlerinizi hızlandıracaktır.

Elbette & veya | operatörlerinin kullanılması gereken durumlarda olabilir. Her n'olursa olsun, koşulun iki tarafınında çalışmasını istiyorsanız, o zaman & ve | operatörlerini kullanmanız gerekmektedir.

Önişlemci Komutları

Bir program yazdığınızı düşünün... Bu programda, PI değerini birçok yerde kullanmanız gerekiyor. Siz de PI değeri olması gereken her yere, 3.14 yazıyorsunuz. Oldukça sıkıcı bir iş. İleride PI'yi, 3.141592653 olarak değiştirmek isterseniz daha da sıkıcı hâle dönüşebilir. Veya canınız istedi, printf( ) fonksiyonu yerine ekrana_yazdir( ) kullanmaya niyetlendiniz... İşte bu gibi durumlarda, Önişlemci Komutlarını (Preprocessor) kullanırız. Önişlemci komutlarının amacı, bir şeyi başka bir şekilde ifade etmektir.

Konuya devam etmeden önce ufak bir uyarı da bulunmakta yarar var. Önişlemci komutlarını, değişkenler veya fonksiyonlarla karıştırmamak gerekiyor. Değişkenlerin ve fonksiyonların daha dinamik ve esnek bir yapıları varken, önişlemci komutları statiktir. Programınıza direkt bir kod yazdığınızı düşünün. Bu kod herhangi bir şey (sembol, program parçası, sayı, karakter vs...) olabilir. Örneğin, her yerde PI'nin karşılığı olarak 3.14 girmek yerine, PI diye bir sembol tanımlarız ve bunun görüldüğü her yere 3.14'ü koyulmasını isteriz. Önişlemci komutları bu gibi işlerde, biçilmiş kaftandır.

#define Önişlemci Komutu

#define komutu, adından da anlaşılabileceği gibi tanımlama işlemleri için kullanılır. Tanımlama komutunun kullanım mantığı çok basittir. Bütün yapmamız gereken, neyin yerine neyi yazacağımıza karar vermektir. Bunun için #define yazıp bir boşluk bıraktıkan sonra, önce kullanacağımız bir isim verilir, ardından da yerine geçeceği değer.

Altta ki program, PI sembolü olan her yere 3.14 koyacak ve işlemleri buna göre yapacaktır:

```
/* Çember alanını hesaplar */
```

```
#include<stdio.h>
#define PI 3.14
int main( void )
{
        int yaricap;
        float alan;
        printf( "Çemberin yarı çapını giriniz> " );
        scanf( "%d", &yaricap );
        alan = PI * yaricap * yaricap;
        printf( "Çember alanı: %.2f\n", alan );
        return 0;
}
```

Gördüğünüz gibi, PI bir değişken olarak tanımlanmamıştır. Ancak #define komutu sayesinde, PI'nin aslında 3.14 olduğu derleyici (compiler) tarafından kabul edilmiştir. Sadece #define komutunu kullanarak başka şeylerde yapmak mümkündür. Örneğin, daha önce dediğimizi yapalım ve printf yerine, ekrana_yazdir; scanf yerine de, deger_al isimlerini kullanalım:

```
/* Yarıçapa göre daire alanı hesaplar */

#include<stdio.h>
#define PI              3.14
#define ekrana_yazdir   printf
#define deger_al        scanf
int main( void )
{
        int yaricap;
        float alan;
        ekrana_yazdir( "Çemberin yarı çapını giriniz> " );
        deger_al( "%d", &yaricap );
        alan = PI * yaricap * yaricap;
        ekrana_yazdir( "Çember alanı: %.2f\n", alan );
        return 0;
}
```

#define komutunun başka marifetleri de vardır. İlerki konularımızda göreceğimiz fonksiyon yapısına benzer şekilde kullanımı mümkündür. Elbette ki, fonksiyonlar çok daha gelişmiştir ve sağladıkları esnekliği, #define tutamaz. Bu yüzden #define ile yapılacakların sınırını çok zorlamamak en iyisi. Ancak yine de bilginiz olması açısından aşağıda ki örneğe göz atabilirsiniz:

```
/* Istenen sayida, "Merhaba" yazar */

#include<stdio.h>
#define merhaba_yazdir( x ) int i; for ( i = 0; i < (x); i++ )
printf( "Merhaba\n" );
int main( void )
{
        int yazdirma_adedi;
        printf( "Kaç defa yazdırılsın> " );
        scanf( "%d", &yazdirma_adedi );
        merhaba_yazdir( yazdirma_adedi );
        return 0;
}
```

#undef Önişlemci Komutu

Bazı durumlarda, #define komutuyla tanımladığımız şeyleri, iptal etmek isteriz. Tanımlamayı iptal etmek için, #undef komutu kullanılır. Örneğin #undef PI yazdığınız da, o noktadan itibaren PI tanımsız olacaktır. #define ile oluşturduğunuz sembolleri belirli noktalardan sonra geçerliliğini iptal

etmek veya yeniden tanımlamak için #undef komutunu kullanabilirsiniz.

#ifdef ve #ifndef Önişlemci Komutları

Önişlemci komutlarında bir sembol veya simgenin daha önce tanıtılıp tanıtılmadığını kontrol etmek isteyebiliriz. Tanıtılmışsa, şöyle yapılsın; yok tanıtılmadıysa, böyle olsun gibi farklı durumlarda ne olacağını belirten yapılar gerekebilir. Bu açığı kapatmak için #ifdef (*if defined - şayet tanımlandıysa*) veya #ifndef (*if not defined - şayet tanımlanmadıysa*) operatörleri kullanılır.

```
#include<stdio.h>
#define PI 3.14
int main( void )
{
        // Tanımlı PI değeri, tanımsız hâle getiriliyor.
        #undef PI

        int yaricap;
        float alan;
        printf( "Çemberin yarı çapını giriniz> " );
        scanf( "%d", &yaricap );

        // PI değerinin tanımlı olup olmadığı kontrol ediliyor.
        #ifdef PI
                //PI tanımlıysa, daire alanı hesaplanıyor.
                alan = PI * yaricap * yaricap;
                printf( "Çember alanı: %.2f\n", alan );
        #else
                //PI değeri tanımsızsa, HATA mesajı veriliyor.
                printf("HATA: Alan değeri tanımlı değildir.\n");
        #endif

        return 0;
}
```

Yukardaki örneğe bakacak olursak, önce PI değeri tanımlanıyor ve sonrasında tanım kaldırılıyor. Biz de sürprizlerle karşılaşmak istemediğimizden, PI değerinin tanım durumunu kontrol ediyoruz. Tek başına çalışan biri için gereksiz bir ayrıntı gibi gözükse de, ekip çalışmalarında, bazı şeylerin kontrol edilmesi ve istenmeyen durumlarda, ne yapılacağı belirlenmelidir. Yukarda ki programı şöyle de yazabilirdik:

```
#include<stdio.h>
int main( void )
{
        int yaricap;
        float alan;
        printf( "Çemberin yarı çapını giriniz> " );
        scanf( "%d", &yaricap );

        // Şu noktaya kadar tanımlı bir PI değeri bulunmuyor.
        // #ifndef opertörü bu durumu kontrol ediyor.
        // Eğer tanımsızsa, PI'nin tanımlanması sağlanıyor.
        #ifndef PI
                #define PI 3.14
        #endif

        alan = PI * yaricap * yaricap;
        printf( "Çember alanı: %.2f\n", alan );

        return 0;
}
```

#if, #else, #endif, #elif Önişlemci Komutları

Bazen bir değerin tanımlanıp, tanımlanmadığını bilmek yetmez. Bazı değerler, bayrak (flag) olarak kullanılır. Yani eğer doğruysa, böyle yapılması lâzım, aksi hâlde böyle olacak gibi... Bazı programlar, önişlemci komutlarından yararlanır. Değişken yerine, önişlemcileri kullanarak tanımlanan simgeler, bu programlarda flag görevi görür.

Konumuza dönersek, #if, #else, #endif yapısı daha önce işlemiş olduğumuz if-else yapısıyla hemen hemen aynıdır. if-elif yapısı da if-else if yapısına benzer. Her ikisinin de genel yazım kuralı aşağıda verilmiştir:

#if - #else Yapısı:

```
#if koşul
        komut(lar)
#else
        komut(lar)
#endif
```

#if - #elif Yapısı:

```
#if koşul 1
        komut(lar) 1
#elif koşul 2
        komut(lar) 2
.
.
.
#elif koşul n-1
        komut(lar) n-1
#else
        komut(lar) n
#endif
```

Bir program tasarlayalım. Bu programda, pi sayısının virgülden sonra kaç basamağının hesaba katılacağına karar veren bir mekanizma olsun. Soruyu, şu ana kadar gördüğümüz, if - else gibi yapılarla rahatça yapabiliriz. Önişlemci komutuyla ise, aşağıdakine benzer bir sistem oluşturulabilir:

```
/* Daire alanını hesaplar */

#include<stdio.h>
#define HASSASLIK_DERECESI 2
int main( void )
{
        int yaricap;
        float alan;
        printf( "Çemberin yarı çapını giriniz> " );
        scanf( "%d", &yaricap );

        // Hassaslık derecesi, pi sayısının virgülden kaç
        // basamak sonrasının hesaba katılacağını belirtir.
        // Eğer hassaslık derecesi bunlara uymuyorsa, alan
        // değeri -1 yapılır.

        #if ( HASSASLIK_DERECESI == 0 )
                alan = 3 * yaricap * yaricap;
        #elif ( HASSASLIK_DERECESI == 1 )
                alan = 3.1 * yaricap * yaricap;
        #elif ( HASSASLIK_DERECESI == 2 )
                alan = 3.14 * yaricap * yaricap;
        #else
                alan = -1;
        #endif

        printf( "Çember alanı: %.2f\n", alan );
        return 0;
}
```

#include Önişlemci Komutu

#include oldukça tanıdık bir operatördür. Her programımızda, #include önişlemci komutunu kullanırız. Şayet kullanmasak, printf( ) veya scanf( ) gibi fonksiyonları tekrar tekrar yazmamız gerekirdi. #include komutu, programımıza bir başlık dosyasının (header file) dâhil edileceğini belirtir. Bu başlık dosyası, standart giriş çıkış işlemlerini içeren bir kütüphane olabileceği gibi, kendimize ait fonksiyonların bulunduğu bir dosya da olabilir.

Eğer sistem kütüphanelerine ait bir başlık dosyasını programınıza dâhil edeceksek, küçüktür (<) ve büyüktür (>) işaretlerini kullanırız. Örneğin stdio.h sisteme ait bir kütüphane dosyasıdır ve Linux'ta /usr/include/stdio.h adresinde bulunur. Dolayısıyla stdio.h kütüphanesini programımıza eklerken, #include<stdio.h> şeklinde yazarız. Kendi oluşturduğumuz başlık dosyaları içinse, durum biraz daha farklıdır.

Çalışma ortamımızla aynı klasörde olan bir başlık dosyasını, programımıza eklemek için #include "benim.h" şeklinde yazarız. İlerki derslerimizde, kendi başlık dosyalarımızı oluşturmayı göreceğiz. Ancak şimdilik burada keselim...

Önemli Noktalar...

Konuyu noktalarken, söylemek istediğim bazı şeyler bulunuyor. Olabildiğince, önişlemci komutlarından - *#include komutu hariç* - uzak durun. Çünkü bu komutlar, esnek bir yapıya sahip değiller ve bu durum, bir noktadan sonra başınızı ağrıtacaktır. Önişlemci komutlarıyla yazılmış kodları takip etmek oldukça zordur ve debug edilemezler. Java gibi gelişmiş dillerde, #define komutu bulunmaz. Modern dillerde, bu yapıdan uzaklaşılmaya başlanmıştır.

Yukarda saydıklarıma rağmen, bazı durumlarda, önişlemci komutlarını kullanmak uygun olabilir. Kaldı ki bu komutların kullanıldığı birçok yer bulunuyor ve biz kullanmasak bile, bilmeye mecbur durumdayız. Sözün özü; bu konuyu es geçmek uygun değil. Ancak üzerine düşmek oldukça gereksiz.

Fonksiyonlar

C gibi prosedürel dillerin önemli konularından birisi fonksiyonlardır. Java veya C# gibi dillerde metot (method) ismini alırlar. Adı n'olursa olsun, görevi aynıdır. Bir işlemi birden çok yaptığınızı düşünün. Her seferinde aynı işlemi yapan kodu yazmak oldukça zahmetli olurdu. Fonksiyonlar, bu soruna yönelik yaratılmıştır. Sadece bir kereye mahsus yapılacak işlem tanımlanır. Ardından dilediğiniz kadar, bu fonksiyonu çağırırsınız. Üstelik fonksiyonların yararı bununla da sınırlı değildir.

Fonksiyonlar, modülerlik sağlar. Sayının asallığını test eden bir fonksiyon yazıp, bunun yanlış olduğunu farkederseniz, bütün programı değiştirmeniz gerekmez. Yanlış fonksiyonu düzeltirsiniz ve artık programınız doğru çalışacaktır. Üstelik yazdığınız fonksiyonlara ait kodu, başka programlara taşımanız oldukça basittir.

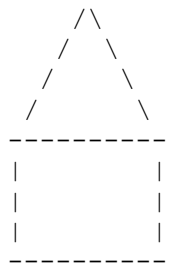
Fonksiyonlar, çalışmayı kolaylaştırır. Diskten veri okuyup, işleyen; ardından kullanıcıya gösterilmek üzere sonuçları grafik hâline dönüştüren; ve işlem sonucunu diske yazan bir programı baştan aşağı yazarsanız, okuması çok güç olur. Yorum koyarak kodun anlaşılabilirliğini, artırabilirsiniz. Ancak yine de yeterli değildir. İzlenecek en iyi yöntem, programı fonksiyon parçalarına bölmektir. Örneğin, diskten okuma işlemini *disten_oku( )* isimli bir fonksiyon yaparken; grafik çizdirme işini *grafik_ciz( )* fonksiyonu ve diske yazdırma görevini de *diske_yaz( )* fonksiyonu yapabilir. Yarın öbür gün, yazdığınız kodu birileri incelediğinde, sadece ilgilendiği yapıya göz atarak, aradığını çok daha rahat bulabilir. Binlerce satır içinde çalışmaktansa, parçalara ayrılmış bir yapı herkesin işine gelecektir.

Bu yazımızda, fonksiyonları açıklayacağız.

main( ) Fonksiyonu

Şimdiye kadar yazdığımız bütün kodlarda, main( ) şeklinde bir notasyon kullandık. Bu kullandığımız ifade, aslında main( ) fonksiyonudur. C programlama dilinde, bir kodun çalışması main( ) fonksiyonun içersinde olup olmamasına bağlıdır. Bir nevi başlangıç noktası olarak düşünebiliriz. Her programda sadece bir tane main( ) fonksiyonu bulunur. Başka fonksiyonların, kütüphanelerin, kod parçalarının çalıştırılması main( ) içersinde direkt veya dolaylı refere edilmesiyle alakalıdır.

main( ) fonksiyonuna dair bilgimizi pekiştirmek için bir program yazalım. Aşağıdaki çizimi inceleyip, C programlama diliyle bunu çizen programı oluşturalım.



Ev veya kule benzeri bu şekli aşağıdaki, kod yardımıyla gösterebiliriz:

```
/* Ev sekli cizen program */
#include<stdio.h>
int main( void )
{
        printf( "    /\\   \n" );
        printf( "   /  \\  \n" );
        printf( "  /    \\ \n" );
        printf( " /      \\\n" );
        printf( "----------\n" );
        printf( "|        |\n" );
        printf( "|        |\n" );
        printf( "|        |\n" );
        printf( "----------\n" );

        return 0;
}
```

Programın özel bir yanı yok. '\' simgesi özel olduğu için bundan iki tane yazmamız gerekti. Bunu önceki derslerimizde işlemiştik. Bunun dışında kodun herhangi bir zorluğu olmadığı için açıklamaya girmiyorum. Dikkat etmeniz gereken tek şey, kodun main( ) fonksiyonuyla çalışması.

Bilgilerimizi özetleyecek olursak; main( ) fonksiyonu özel bir yapıdır. Hazırladığımız program, main( ) fonksiyonuyla çalışmaya başlar. main( ) fonksiyonu içersinde yer almayan kodlar çalışmaz.

Fonksiyon Oluşturma

Kendinize ait fonksiyonlar oluşturabilirsiniz. Oluşturacağınız fonksiyonlar, vereceğiniz işlemi yapmakla görevlidir ve çağrıldıkça tekrar tekrar çalışır.

Yukardaki ev örneğine geri dönelim. Her şeyi main( ) içinde, tek bir yerde yazacağımıza, çatıyı çizen ayrı, katı çizen ayrı birer fonksiyon yazsaydık daha rahat olmaz mıydı? Ya da birden çok kat çizmemiz gerekirse, tek tek kat çizmekle uğraşmaktansa, fonksiyon adını çağırmak daha akıllıca değil mi? Bu soruların yanıtı, bizi fonksiyon kullanmaya götürüyor. Şimdi yukarda yazdığımız

kodu, iki adet fonksiyon kullanarak yapalım:

```
/* Ev sekli cizen program */
#include<stdio.h>
// Evin catisini cizen fonksiyon.
void catiyi_ciz( void )
{
        printf( "    /\\   \n" );
        printf( "   /  \\  \n" );
        printf( "  /    \\ \n" );
        printf( " /      \\\n" );
        printf( "----------\n" );
}

// Evin katini cizen fonksiyon.
void kat_ciz( void )
{
        printf( "|        |\n" );
        printf( "|        |\n" );
        printf( "|        |\n" );
        printf( "----------\n" );
}

// Programin calismasini saglayan
// ana fonksiyon.
int main( void )
{
        catiyi_ciz( );
        kat_ciz( );

        return 0;
}
```

Yazdığımız bu kod, ilk başta elde ettiğimiz çıktının aynısını verir. Ama önemli bir fark içerir:. Bu programla birlikte ilk defa fonksiyon kullanmış olduk!

Fonksiyon kullanmanın, aynı şeyleri baştan yazma zahmetinden kurtaracağından bahsetmiştik. Diyelim ki bize birden çok kat gerekiyor. O zaman *kat_ciz( )* fonksiyonunu gereken sayıda çağırmamız yeterlidir.

```
/* Ev sekli cizen program */
#include<stdio.h>
// Evin catisini cizen fonksiyon.
void catiyi_ciz( void )
{
        printf( "    /\\   \n" );
        printf( "   /  \\  \n" );
        printf( "  /    \\ \n" );
        printf( " /      \\\n" );
        printf( "----------\n" );
}

// Evin katini cizen fonksiyon.
void kat_ciz( void )
{
        printf( "|        |\n" );
        printf( "|        |\n" );
        printf( "|        |\n" );
        printf( "----------\n" );
}

// Programin calismasini saglayan
// ana fonksiyon.
int main( void )
```

```
{
        catiyi_ciz( );
        // 3 adet kat ciziliyor.
        kat_ciz( );
        kat_ciz( );
        kat_ciz( );

        return 0;
}
```

Yukarda yazılı kod, bir üstekinden pek farklı durmasa bile, bu sefer üç katlı bir evin çıktısını elde etmiş olacaksınız.

Yaptığımız örneklerde, kullanılan *void* ifadesi dikkatinizi çekmiş olabilir. İngilizce bir kelime olan void, boş/geçersiz anlamındadır. C programlama dilinde de buna benzer bir anlam taşır. *kat_ciz( );* fonksiyonuna bakalım. Yapacağı iş için herhangi bir değer alması gerekmiyor. Örneğin verilen sayının asallığını test eden bir fonksiyon yazsaydık, bir değişken almamız gerekirdi. Ancak bu örnekte gördüğümüz *kat_ciz( );* fonksiyonu, dışardan bir değere gerek duymaz. Eğer bir fonksiyon, çalışmak için dışardan gelecek bir değere ihtiyaç duymuyorsa, fonksiyon adını yazdıktan sonra parantez içini boş bırakabiliriz. Ya da *void* yazarak, fonksiyonun bir değer almayacağını belirtiriz. (Sürekli olarak *main( )* fonksiyonuna *void* koymamızın sebebi de bundandır; fonksiyon argüman almaz.) İkinci yöntem daha uygun olmakla birlikte, birinci yöntemi kullanmanın bir mahsuru yok. Aşağıda bulunan iki fonksiyon aynı şekilde çalışır:

```
// Evin katini cizen fonksiyon.
// void var

void kat_ciz( void )
{
        printf( "|         |\n" );
        printf( "|         |\n" );
        printf( "|         |\n" );
        printf( "----------\n" );
}
```

```
// Evin katini cizen fonksiyon.
// void yok

void kat_ciz( )
{
        printf( "|         |\n" );
        printf( "|         |\n" );
        printf( "|         |\n" );
        printf( "----------\n" );
}
```

void ifadesinin, değer alınmayacağını göstermek için kullanıldığını gördünüz. Bir de fonksiyonun değer döndürme durumu vardır. Yazdığınız fonksiyon yapacağı işlemler sonucunda, çağrıldığı noktaya bir değer gönderebilir. Değer döndürme konusunu, daha sonra işleyeceğiz. Şimdilik değer döndürmeme durumuna bakalım.

Yukarda kullanılan fonksiyonlar, geriye bir değer döndürmemektedir. Bir fonksiyonun geriye değer döndürmeyeceğini belirtmek için, *void* ifadesini fonksiyon adından önce yazarız. Böyleyece geriye bir değer dönmeyeceği belirtilir.

Argüman Aktarımı

Daha önce ki örneklerimiz de, fonksiyonlar dışardan değer almıyordu. Bu yüzden parantez içlerini boş bırakmayı ya da void ifadesini kullanmayı görmüştük. Her zaman böyle olması gerekmez; fonksiyonlar dışardan değer alabilirler.

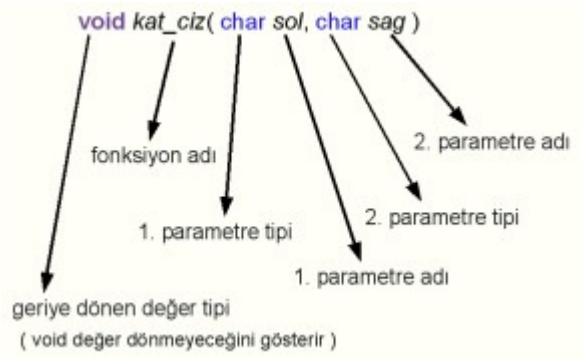
Fonksiyonu tanımlarken, fonksiyona nasıl bir değerin gönderileceğini belirtiriz. Gönderilecek değerin hangi değişken tipinde olduğunu ve değişken adını yazarız. Fonksiyonu tanımlarken, yazdığımız bu değişkenlere 'parametre' (parameter) denir. Argüman (argument) ise, parametrelere değer atamasında kullandığımız değerlerdir. Biraz karmaşık mı geldi? O zaman bir örnekle açıklayalım.

Daha önce çizdiğimiz ev örneğini biraz geliştirelim. Bu sefer, evin duvarları düz çizgi olmasın; kullanıcı istediği karakterlerle, evin duvarlarını çizdirsin.

```
/* Ev sekli cizen program */
#include<stdio.h>
// Evin catisini cizen fonksiyon.
void catiyi_ciz( void )
{
        printf( "    /\\   \n" );
        printf( "   /  \\  \n" );
        printf( "  /    \\ \n" );
        printf( " /      \\\n" );
        printf( "----------\n" );
}

// Evin katini cizen fonksiyon.
// sol ve sag degiskenleri fonksiyon
// parametreleridir.
void kat_ciz( char sol, char sag )
{
        printf( "%c        %c\n", sol, sag );
        printf( "%c        %c\n", sol, sag );
        printf( "%c        %c\n", sol, sag );
        printf( "----------\n" );
}

// Programin calismasini saglayan
// ana fonksiyon.
int main( void )
{
        char sol_duvar, sag_duvar;
        printf( "Kullanılacak karakterler> " );
        scanf( "%c%c",&sol_duvar, &sag_duvar );
        catiyi_ciz( );

        // sol_duvar ve sag_duvar, fonksiyona
        // giden argumanlardir.
        kat_ciz( sol_duvar, sag_duvar );
        kat_ciz( sol_duvar, sag_duvar );
        kat_ciz( sol_duvar, sag_duvar );

        return 0;
}
```

Argümanların değer olduğunu unutmamak gerekiyor. Yukardaki örneğimizden, değişken olması gerektiği yanılgısına düşebilirsiniz. Ancak bir fonksiyona değer aktarırken, direkt olarak değeri de yazabilirsiniz. Programı değiştirip, *sol_duvar* ve *sag_duvar* değişkenleri yerine, '*' simgesini koyun. Şeklin duvarları, yıldız işaretinden oluşacaktır.

Yazdığımız *kat_ciz( )* fonksiyonunu incelemek için, aşağıda bulunan grafiğe göz atabilirsiniz:



Şimdi de başka bir örnek yapalım ve verilen herhangi bir sayının tek mi yoksa çift mi olduğuna karar veren bir fonksiyon oluşturalım:

```
/* Sayının tek veya çift olmasını
   kontrol eder. */
#include<stdio.h>
void tek_mi_cift_mi( int sayi )
{
        if( sayi%2 == 0 )
                printf( "%d, çift bir sayıdır.\n", sayi );
        else
                printf( "%d, tek bir sayıdır.\n", sayi );
}
int main( void )
{
        int girilen_sayi;
        printf( "Lütfen bir sayı giriniz> " );
        scanf( "%d",&girilen_sayi );
        tek_mi_cift_mi( girilen_sayi );

        return 0;
}
```

Yerel (Local) ve Global Değişkenler

Kendi oluşturacağınız fonksiyon içersinde, main( ) fonksiyonunda ki her şeyi yapabilirsiniz. Değişken tanımlayabilir, fonksiyon içinden başka fonksiyonları çağırabilir veya dilediğiniz operatörü kullanabilirsiniz. Ancak değişken tanımlamalarıyla ilgili göz ardı etmememiz gereken bir konu bulunuyor. Bir fonksiyon içersinde tanımladığınız değişkenler, sadece o fonksiyon içersinde tanımlıdır. main( ) veya kendinize ait fonksiyonlardan bu değişkenlere ulaşmamız mümkün değildir. main( ) içinde tanımladığınız a isimli değişkenle, kendinize özgü tanımladığınız *kup_hesapla( )* içersinde tanımlanmış a isimli değişken, bellekte farklı adresleri işaret eder. Dolayısıyla değişkenlerin arasında hiçbir ilişki yoktur. *kup_hesapla( )* içersinde geçen a değişkeninde yapacağınız değişiklik, main( ) fonksiyonundakini etkilemez. Keza, tersi de geçerlidir. Şu ana kadar yaptığımız bütün örneklerde, değişkenleri yerel olarak tanımladığımızı belirtelim.

Yerel değişken dışında, bir de global değişken tipi bulunur. Programın herhangi bir noktasından erişebileceğiniz ve nerede olursa olsun aynı bellek adresini işaret eden değişkenler, global değişkenlerdir. Hep aynı bellek adresi söz konusu olduğun için, programın herhangi bir noktasında yapacağınız değişiklik, global değişkenin geçtiği bütün yerleri etkiler. Aşağıdaki örneği inceleyelim:

```
#include<stdio.h>
// Verilen sayinin karesini hesaplar
void kare_hesapla( int sayi )
{
        // kare_hesapla fonksiyonunda
        // a degiskeni tanimliyoruz.
        int a;
        a = sayi * sayi;
        printf( "Sayının karesi\t: %d\n", a );
}

// Verilen sayinin kupunu hesaplar
void kup_hesapla( int sayi )
{
        // kup_hesapla fonksiyonunda
        // a degiskeni tanimliyoruz.
        int a;
        a = sayi * sayi * sayi;
        printf( "Sayının küpü\t: %d\n", a );
}
```

```
int main( void )
{
        // main( ) fonksiyonunda
        // a degiskeni tanimliyoruz.
        int a;
        printf( "Sayı giriniz> ");
        scanf( "%d",&a );
        printf( "Girdiğiniz sayı\t: %d\n", a );
        kare_hesapla( a );
        // Eger a degiskeni lokal olmasaydi,
        // kare_hesapla fonksiyonundan sonra,
        // a'nin degeri bozulur ve kup yanlis
        // hesaplanirdi.
        kup_hesapla( a );
        return 0;
}
```

Kod arasına konulan yorumlarda görebileceğiniz gibi, değişkenler lokal olarak tanımlanmasa, a'nin değeri farklı olurdu. Sayının karesini bulduktan sonra, küpünü yanlış hesaplardık. Değişkenler lokal olduğu için, her aşamada farklı bir değişken tanımlandı ve sorun çıkartacak bir durum olmadı. Benzer bir programı global değişkenler için inceleyelim:

```
#include<stdio.h>
int sonuc = 0;

// Verilen sayinin karesini hesaplayip,
// global 'sonuc' degiskenine yazar.
void kare_hesapla( int sayi )
{
        sonuc = sayi * sayi;
}

int main( void )
{
        // main( ) fonksiyonunda
        // a degiskeni tanimliyoruz.
        int a;
        printf( "Sayı giriniz> ");
        scanf( "%d",&a );
        printf( "Girdiğiniz sayı\t: %d\n", a );
        kare_hesapla( a );
        printf("Sayının karesi\t: %d\n", sonuc );
        return 0;
}
```

Gördüğünüz gibi, *sonuc* isimli değişken her iki fonksiyonun dışında bir alanda, programın en başında tanımlanıyor. Bu sayede, fonksiyon bağımsız bir değişken elde ediyoruz.

Global değişkenlerle ilgili dikkat etmemiz gereken bir iki ufak nokta bulunuyor: Global bir değişkeni fonksiyonların dışında bir alanda tanımlarız. Tanımladığımız noktanın altında kalan bütün fonksiyonlar, bu değişkeni tanır. Fakat tanımlanma noktasının üstünde kalan fonksiyonlar, değişkeni görmez. Bu yüzden, bütün programda geçerli olacak gerçek anlamda global bir değişken istiyorsanız, #include ifadelerinin ardından tanımlamayı yapmanız gerekir. Aynı ismi taşıyan yerel ve global değişkenleri aynı anda kullanıyorsak, iş birazcık farklılaşır.

Bir fonksiyon içersinde, Global değişkenle aynı isimde, yerel bir değişken bulunduruyorsanız, bu durumda lokal değişkenle işlem yapılır. Açıkcası, sınırsız sayıda değişken ismi vermek mümkünken, global değişkenle aynı adı vermenin uygun olduğunu düşünmüyorum. Program akışını takip etmeyi zorlaştıracağından, ortak isimlerden kaçınmak daha akıllıca.

Lokal ve global değişkenlere dair son bir not; lokal değişkenlerin sadece fonksiyona özgü olması

gerekmez. Bir fonksiyon içersinde 'daha lokal' değişkenleri tanımlayabilirsiniz. Internet'te bulduğum aşağıdaki programı incelerseniz, konuyu anlamanız açısından yardımcı olacaktır.

```
#include<stdio.h>
int main( void )
{
        int i = 4;
        int j = 10;

        i++;

        if( j > 0 ){
          printf("i : %d\n",i);          /* 'main' icinde tanımlanmis 'i'
degiskeni */
        }

        if (j > 0){
                int i=100;               /* 'i' sadece bu if blogunda gecerli
                                            olmak uzere tanimlaniyor. */
                printf("i : %d\n",i);
        }                                /* if blogunda tanimlanan ve 100
degerini
                                         tasiyan 'i' degiskeni burada sonlaniyor.
*/

        printf("i : %d\n",i);            /* En basta tanimlanan ve 5 degerini
tasiyan
                                         'i' degiskenine donuyoruz */
}
```

return İfadesi

Yazımızın üst kısımlarında fonksiyonların geriye değer döndürebileceğinden bahsetmiştik. Bir fonksiyonun geriye değer döndürüp döndürmemesi, o fonksiyonu genel yapı içersinde nasıl kullanacağınıza bağlıdır. Eğer hazırlayacağınız fonksiyonun, çalışıp, üreteceği sonuçları başka yerlerde kullanmayacaksanız, fonksiyondan geriye değer dönmesi gerekmez. Ancak fonksiyonun ürettiği sonuçları, bir değişkene atayıp kullanacaksanız, o zaman fonksiyonun geriye değer döndürmesi gerekir. Bunun için 'return' ifadesini kullanırız.

Daha önce gördüğümüz geriye değer döndürmeyen fonksiyonları tanımlarken, başına *void* koyuyorduk. Geriye değer döndüren fonksiyonlar içinse, hangi tipte değer dönecekse, onu fonksiyon adının başına koyuyoruz. Diyelim ki fonksiyonumuz bir tamsayı döndürecekse, *int*; bir karakter döndürecekse *char* diye belirtiyoruz. Fonksiyon içersinden neyin döneceğine gelince, burada da *return* ifadesi devreye giriyor.

Fonksiyonun neresinde olduğu farketmez, return sonuç döndürmek üzere kullanılır. Döndüreceği sonuç, elle girilmiş veya değişkene ait bir değer olabilir. Önemli olan döndürülecek değişken tipiyle, döndürülmesi vaad edilen değişken tipinin birbirinden farklı olmamasıdır. Yani *int kup_hesapla( )* şeklinde bir tanımlama yaptıysanız, double tipinde bir sonucu döndüremezsiniz. Daha doğrusu döndürebilirsiniz ama program yanlış çalışır. Tip uyuşmazlığı genel hatalardan biri olduğu için, titiz davranmanızı öğütlerim.

Şimdi return ifadesini kullanabileceğimiz, bir program yapalım. Kullanıcıdan bir sayı girilmesi istensin; girilen sayı asal değilse, tekrar ve tekrar sayı girmesi gereksin:

```
#include<stdio.h>

// Verilen sayinin asal olup olmadigina
// bakar. Sayi asalsa, geriye 1 aksi hâlde
// 0 degeri doner.
```

```
int sayi_asal_mi( int sayi )
{
        int i;
        for( i = 2; i <= sayi/2; i++ ) {
        // Sayi asal degilse, i'ye tam olarak
        // bolunur.
                if( sayi%i == 0 ) return 0;
        }
        // Verilen sayi simdiye kadar hicbir
        // sayiya bolunmediyse, asaldir ve
        // geriye 1 doner.
        return 1;
}

// main fonksiyonu
int main( void )
{
        int girilen_sayi;
        int test_sonucu;
        do{
                printf( "Lütfen bir sayı giriniz> " );
                scanf( "%d",&girilen_sayi );
                test_sonucu = sayi_asal_mi( girilen_sayi );
                if( !test_sonucu )
                        printf("Girilen sayı asal değildir!\n");
        } while( !test_sonucu );
        printf( "Girilen sayı asaldır!\n" );

        return 0;
}
```

Dikkat edilmesi gereken bir diğer konu; return koyduğunuz yerde, fonksiyonun derhâl sonlanmasıdır. Fonksiyonun kalan kısmı çalışmaz. Geriye değer döndürmeye fonksiyonlar için de aynı durum geçerlidir, onlarda da return ifadesini kullanabilirsiniz. Değer döndürsün, döndürmesin yazdığınız fonksiyonda herhangi bir yere '*return;*' yazın. Fonksiyonun bu noktadan itibaren çalışmayı kestiğini farkedeceksiniz. Bu fonksiyonu çalıştırmanın uygun olmadığı şartlarda, kullanabileceğiniz bir yöntemdir. Bir kontrol ekranında, kullanıcı adı ve/veya şifresini yanlış girildiğinde, programın çalışmasını anında kesmek isteyebilirsiniz. Böyle bir durumda '*return;*' kullanılabilir.

Dersimizi burada tamamlayıp örnek sorulara geçmeden önce, fonksiyonlara ait genel yapıyı incelemenizi öneririm.

```
donus_tipi fonksiyon_adi( alacagi_arguman[lar] )
{
        .
        .
        FONKSİYON İÇERİĞİ
        ( YAPILACAK İŞLEMLER )
        .
        .
        [return deger]
}
```

NOT: Köşeli parantez gördüğünüz yerler opsiyoneldir. Her fonksiyonda yazılması gerekmez. Ancak oluşturacağınız fonksiyon yapısına bağlı olarak yazılması şartta olabilir. Mesela dönüş tipi olarak void dışında bir değişken tipi belirlediyseniz, return koymanız gerekir.

Bazı Aritmetik Fonksiyonlar

Geçen dersimizde, fonksiyonları ve bunları nasıl kullanılacağını görmüştük. Ayrıca kütüphanelerin hazır fonksiyonlar içerdiğinden bahsetmiştik. Bazı matematiksel işlemlerin kullanımı sıkça gerekebileceği için bunları bir liste hâlinde vermenin uygun olduğuna inanıyorum. Böylece var olan aritmetik fonksiyonları tekrar tekrar tanımlayarak zaman kaybetmezsiniz.

- double **ceil**(*double n*) : Virgüllü n sayısını, kendisinden büyük olan ilk tam sayıya tamamlar. Örneğin ceil(51.4) işlemi, 52 sonucunu verir.
- double **floor**(*double n*) : Virgüllü n sayısının, virgülden sonrasını atarak, bir tam sayıya çevirir. floor(51.4) işlemi, 51 sayısını döndürür.
- double **fabs**(*double n*) : Verilen n sayısının mutlak değerini döndürür. fabs(-23.5), 23.5 değerini verir.
- double **fmod**(*double a*, *double b*) : a sayısının b sayısına bölümünden kalanı verir. (Daha önce gördüğümüz modül (%) operatörü, sadece tam sayılarda kullanılırken, fmod fonksiyonu virgüllü sayılarda da çalışır.)
- double **pow**(*double a*, *double b*) : Üstel değer hesaplamak için kullanılır; a^b değerini verir.
- double **sqrt**(*double a*) : a'nın karekökünü hesaplar.

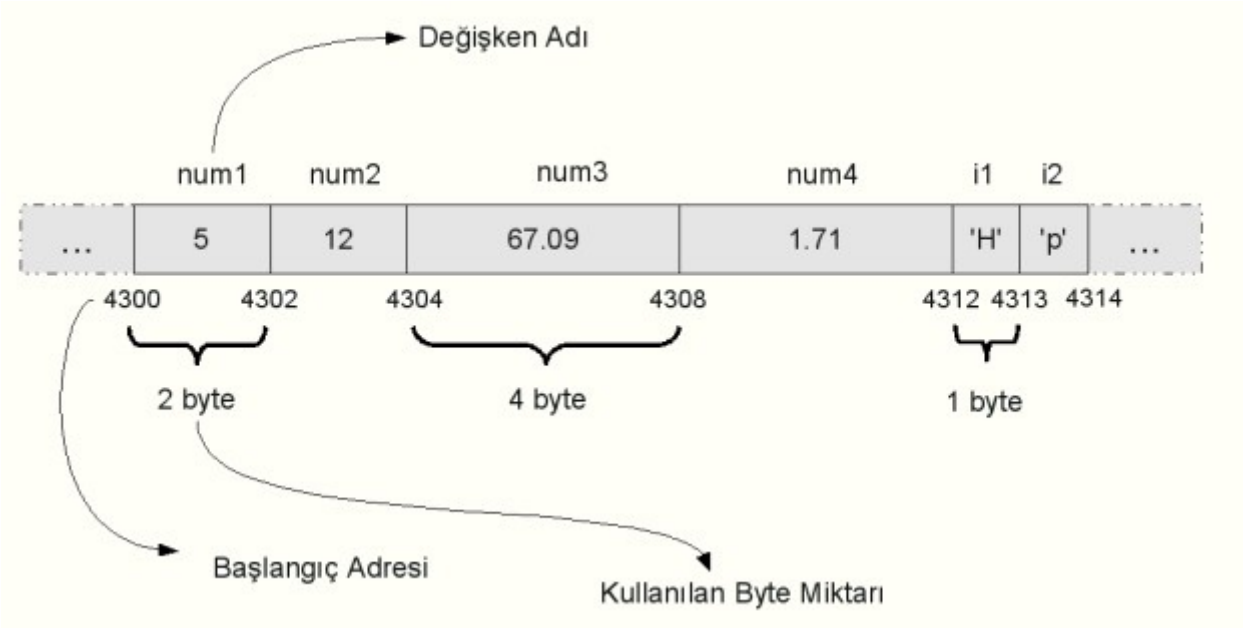
Yukarda verilmiş fonksiyonlar, matematik kütüphanesi (math.h) altındadır. Bu fonksiyonlardan herhangi birini kullacağınız zaman, program kodununun başına #include<math.h> yazmalısınız. Ayrıca derleyici olarak gcc'yle çalışıyorsanız, derlemek için -lm parametresini eklemeniz gerekir. (Örneğin: "*gcc test.c -lm*" gibi...)

Bellek Yapısı ve Adresler

Şimdiye kadar değişken tanımlamayı gördük. Bir değişken tanımlandığında, arka plânda gerçekleşen olaylara ise değinmedik. Hafızayı küçük hücrelerden oluşmuş bir blok olarak düşünebilirsiniz. Bir değişken tanımladığınızda, bellek bloğundan gerekli miktarda hücre, ilgili değişkene ayrılır. Gereken hücre adedi, değişken tipine göre değişir. Şimdi aşağıdaki kod parçasına bakalım:

```
#include<stdio.h>
int main( void )
{
        // Degiskenler tanımlanıyor:
        int num1, num2;
        float num3, num4;
        char i1, i2;

        // Degiskenlere atama yapiliyor:
        num1 = 5;
        num2 = 12;
        num3 = 67.09;
        num4 = 1.71;
        i1 = 'H';
        i2 = 'p';

        return 0;
}
```

Yukarda bahsettiğimiz hücrelerden oluşan bellek yapısını, bu kod parçası için uygulayalım. Değişken tiplerinden int'in 2 byte, float'un 4 byte ve char'ın 1 byte yer kapladığını kabul edelim. Her bir hücre 1 byte'lık alanı temsil etsin. Değişkenler için ayrılan hafıza alanı, 4300 adresinden başlasın. Şimdi bunları temsili bir şekle dökelim:



Bir değişken tanımladığımızda, bellekte gereken alan onun adına rezerve edilir. Örneğin 'int num1' yazılması, bellekte uygun bir yer bulunup, 2 byte'ın, *num1* değişkeni adına tutulmasını sağlıyor. Daha sonra *num1* değişkenine değer atarsak, ayrılan hafıza alanına 5 sayısı yazılıyor. Aslında, *num1* ile ilgili yapacağınız bütün işlemler, 4300 adresiyle 4302 adresi arasındaki bellek hücrelerinin değişmesiyle alakalıdır. Değişken dediğimiz; uygun bir bellek alanının, bir isme revize edilip, kullanılmasından ibarettir.

Bir parantez açıp, küçük bir uyarı da bulunalım. Şeklimizin temsili olduğunu unutmamak gerekiyor. Değişkenlerin bellekteki yerleşimi bu kadar 'uniform' olmayabilir. Ayrıca başlangıç adresini 4300 olarak belirlememiz keyfiydi. Sayılar ve tutulan alanlar değişebilir. Ancak belleğin yapısının, aşağı yukarı böyle olduğunu kabul edebilirsiniz.

Pointer Mekanizması

Bir değişkene değer atadığımızda, aslında bellek hücrelerini değiştirdiğimizi söylemiştik. Bu doğru bir tanım ama eksik bir noktası var. Bellek hücrelerini değiştermemize rağmen, bunu direkt yapamaz; değişkenleri kullanırız. Bellek hücrelerine direkt müdahâle Pointer'lar sayesinde gerçekleşir.

Pointer, birçok Türkçe kaynakta 'işaretçi' olarak geçiyor. Direkt çevirirseniz mantıklı. Ancak terimlerin özünde olduğu gibi öğrenilmesinin daha yararlı olduğunu düşünüyorum ve ben Pointer olarak anlatacağım. Bazı yerlerde işaretçi tanımı görürseniz, bunun pointer ile aynı olduğunu bilin. Şimdi gelelim Pointer'in ne olduğuna...

Değişkenler bildiğiniz gibi değer (sayı, karakter, vs...) tutar. Pointer'lar ise adres tutan değişkenlerdir. Bellekten bahsetmiştik; küçük hücrelerin oluşturduğu hafıza bloğunun adreslere ayrıldığını ve değişkenlerin bellek hücrelerine yerleştiğini gördük. İşte pointer'lar bu bellek adreslerini tutarlar.

Pointer tanımlamak oldukça basittir. Sadece değişken adının önüne '*' işareti getiririz. Dikkat edilmesi gereken tek nokta; pointer'ı işaret edeceği değişken tipine uygun tanımlamaktır. Yani float bir değişkeni, int bir pointer ile işaretlemeğe çalışmak yanlıştır! Aşağıdaki örneğe bakalım:

```
#include<stdio.h>
int main( void )
{
        // int tipinde değişken
        // tanımlıyoruz:
        int xyz = 10, k;
```

```
        // int tipinde pointer
        // tanımlıyoruz:
        int *p;

        // xyz değişkeninin adresini
        // pointer'a atıyoruz.
        // Bir değişken adresini '&'
        // işaretiyle alırız.
        p = &xyz;

        // k değişkenine xyz'nin değeri
        // atanır. Pointer'lar değer tutmaz.
        // değer tutan değişkenleri işaret
        // eder. Başına '*' koyulduğunda,
        // işaret ettiği değişkenin değerini
        // gösterir.
        k = *p;

        return 0;
}
```

Kod parçasındaki yorumları okuduğunuzda, pointer ile ilgili fikriniz olacaktır. Pointer adres tutan değişkenlerdir. Şimdiye kadar gördüğümüz değişkeninlerin saklayabildiği değerleri tutamazlar. Sadece değişkenleri işaret edebilirler. Herhangi bir değişkenin adresini pointer içersine atamak isterseniz, değişken adının önüne '&' getirmeniz gerekir. Bundan sonra o pointer, ilgili değişkeni işaret eder. Eğer bahsettiğimiz değişkenin sahip olduğu değeri pointer ile göstermek veya değişken değerini değiştirmek isterseniz, pointer başına '*' getirerek işlemlerinizi yapabilirsiniz. Pointer başına '*' getirerek yapacağınız her atama işlemi, değişkeni de etkileyecektir. Daha kapsamlı bir örnek yapalım:

```
#include<stdio.h>
int main( void )
{
        int x, y, z;
        int *int_addr;
        x = 41;
        y = 12;
        // int_addr x degiskenini
        // isaret ediyor.
        int_addr = &x;
        // int_addr'in isaret ettigi
        // degiskenin sakladigi deger
        // aliniyor. (yani x'in degeri)
        z = *int_addr;
        printf( "z: %d\n", z );
        // int_addr, artik y degiskenini
        // isaret ediyor.
        int_addr = &y;
        // int_addr'in isaret ettigi
        // degiskenin sakladigi deger
        // aliniyor. (yani y'nin degeri)
        z = *int_addr;
        printf( "z: %d\n" ,z );

        return 0;
}
```

Bir pointer'in işaret ettiği değişkeni program boyunca sürekli değiştirebilirsiniz. Yukardaki örnekte, int_addr pointer'i, önce x'i ve ardından y'yi işaret etmiştir. Bu yüzden, z değişkenine int_addr kullanarak yaptığımız atamalar, her seferinde farklı sonuçlar doğurmuştur. Pointer kullanarak, değişkenlerin sakladığı değerleri de değiştirebiliriz. Şimdi bununla ilgili bir örnek inceleyelim:

```
#include<stdio.h>
int main( void )
{
        int x, y;
        int *int_addr;
        x = 41;
        y = 12;
        // int_addr x degiskenini
        // isaret ediyor
        int_addr = &x;
        // int_addr'in isaret ettigi
        // degiskenin degerini
        // degistiriyoruz
        *int_addr = 479;
        printf( "x: %d y: %d\n", x, y );
        int_addr = &y;

        return 0;
}
```

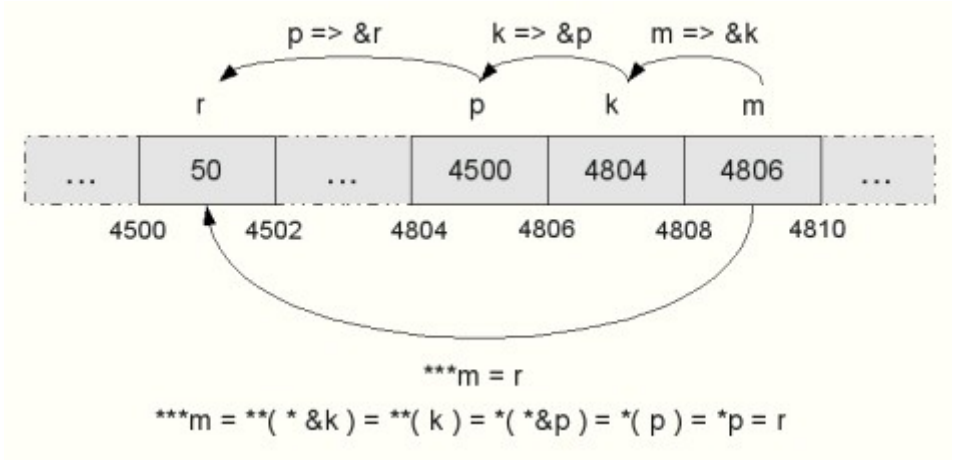
Kodu derleyip, çalıştırdığınızda, x'in değerinin değiştiğini göreceksiniz. Pointer başına '*' getirip, pointer'a bir değer atarsanız; aslında işaret ettiği değişkene değer atamış olursunuz. Pointer ise hiç değişmeden, aynı adres bilgisini tutmaya devam edecektir.

Pointer tutan Pointer'lar

Pointer'lar, gördüğümüz gibi değişkenleri işaret ederler. Pointer'da bir değişkendir ve onu da işaret edecek bir pointer yapısı kullanılabilir. Geçen sefer ki bildirimden farkı, pointer değişkenini işaret edecek bir değişken tanımlıyorsanız; başına '**' getirmeniz gerekmesidir. Buradaki yıldız sayısı değişebilir. Eğer, pointer işaret eden bir pointer'i işaret edecek bir pointer tanımlamak istiyorsanız, üç defa yıldız (***) yazmanız gerekir. Evet, cümle biraz karmaşık, ama kullanım oldukça basit! Pointer işaret eden pointer'ları aşağıdaki örnekte bulabilirsiniz:

```
#include<stdio.h>
int main( void )
{
        int r = 50;
        int *p;
        int **k;
        int ***m;
        printf( "r: %d\n", r );
        p = &r;
        k = &p;
        m = &k;
        ***m = 100;
        printf( "r: %d\n", r );

        return 0;
}
```

Yazmış olduğumuz kod içersinde kimin neyi gösterdiğini grafikle daha iyi anlayabiliriz:



Birbirini gösteren Pointer'ları ilerki derslerimizde, özellikle dinamik bellek tahsis ederken çok ihtiyaç duyacağımız bir yapı. O yüzden iyice öğrenmek gerekiyor.

Referansla Argüman Aktarımı

Fonksiyonlara nasıl argüman aktaracağımızı biliyoruz. Hatırlayacağınız gibi parametrelere değer atıyorduk. Bu yöntemde, kullandığınız argümanların değeri değişmiyordu. Fonksiyona parametre olarak yollanan argüman hep aynı kalıyordu. Fonksiyon içinde yapılan işlemlerin hiçbiri argüman değişkeni etkilemiyordu. Sadece değişken değerinin aktarıldığı ve argümanın etkilenmediği bu duruma, "*call by value*" veya "*pass by value*" adı verilir. Bu isimleri bilmiyor olsanız dahi, şu ana kadar ki fonksiyon çalışmaları böyleydi.

Geriye birden çok değer dönmesi gereken veya fonksiyonun içersinde yapacağınız değişikliklerin, argüman değişkene yansıması gereken durumlar olabilir. İşte bu gibi zamanlarda, "*call by reference*" veya "*pass by reference*" olarak isimlendirilen yöntem kullanılır. Argüman değer olarak aktarılmaz; argüman olan değişkenin adres bilgisi fonksiyona aktarılır. Bu sayede fonksiyon içersinde yapacağınız her türlü değişiklik argüman değişkene de yansır.

Söylediklerimizi uygulamaya dökelim ve kendisine verilen iki sayının yerlerini değiştiren bir fonksiyon yazalım. Yani kendisine a ve b adında iki değişken yollanıyorsa, a'nın değerini b; b'nin değeriniyse a yapsın.

```
#include<stdio.h>
// Kendisine verilen iki degiskenin
// degerlerini degistirir.
// Parametreleri tanimlarken baslarina
// '*' koyuyoruz.
void swap( int *x, int *y )
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
int main( void )
{
        int a, b;
        a = 12;
        b = 27;
        printf( "a: %d b: %d\n", a, b );
        // Argumanları aktarırken, baslarina
        // '&' koyuyoruz.
        swap(&a, &b);
        printf( "a: %d b: %d\n", a, b );
```

```
        return 0;
}
```

Referans yoluyla aktarım olmasaydı, iki değişkenin değerlerini fonksiyon kullanarak değiştiremezdik. Eğer yazdığınız fonksiyon birden çok değer döndürmek zorundaysa, referans yoluyla aktarım zorunlu hâle geliyor. Çünkü daha önce işlediğimiz return ifadesiyle sadece tek bir değer döndürebiliriz. Örneğin bir bölme işlemi yapıp, bölüm sonucunu ve kalanı söyleyen bir fonksiyon yazacağımızı düşünelim. Bu durumda, bölünen ve bölen fonksiyona gidecek argümanlar olurken; kalan ve bölüm geriye dönmelidir. return ifadesi geriye tek bir değer vereceğinden, ikinci değeri alabilmek için referans yöntemi kullanmamız gerekir.

```
#include<stdio.h>
int bolme_islemi( int bolunen, int bolen, int *kalan )
{
        *kalan = bolunen % bolen;
        return bolunen / bolen;
}
int main( void )
{
        int bolunen, bolen;
        int bolum, kalan;
        bolunen = 13;
        bolen = 4;
        bolum = bolme_islemi( bolunen, bolen, &kalan );
        printf( "Bölüm: %d Kalan: %d\n", bolum, kalan );

        return 0;
}
```

Fonksiyon Prototipleri

Bildiğiniz gibi fonksiyonlarımızı, main( ) üzerine yazıyoruz. Tek kısa bir fonksiyon için bu durum rahatsız etmez; ama uzun uzun 20 adet fonksiyon olduğunu düşünün. main( ) fonksiyonu sayfalar dolusu kodun altında kalacak ve okunması güçleşecektir. Fonksiyon prototipleri burada devreye girer.

Bir üstte yazdığımız programı tekrar yazalım. Ama bu sefer, fonksiyon prototipi yapısına uygun olarak bunu yapalım:

```
#include<stdio.h>
int bolme_islemi( int, int, int * );
int main( void )
{
        int bolunen, bolen;
        int bolum, kalan;
        bolunen = 13;
        bolen = 4;
        bolum = bolme_islemi( bolunen, bolen, &kalan );
        printf( "Bölüm: %d Kalan: %d\n", bolum, kalan );

        return 0;
}
int bolme_islemi( int bolunen, int bolen, int *kalan )
{
        *kalan = bolunen % bolen;
        return bolunen / bolen;
}
```

bolme_islemi( ) fonksiyonunu, main( ) fonksiyonundan önce yazmadık. Sadece böyle bir fonksiyon olduğunu ve alacağı parametre tiplerini bildirdik. (İsteseydik parametre adlarını da

yazabilirdik ama buna gerek yok.) Daha sonra main( ) fonksiyonu altına inip, fonksiyonu yazdık.

Öğrendiklerimizi pekiştirmek için yeni bir program yazalım. Fonksiyonumuz, kendisine argüman olarak gönderilen bir pointer'i alıp; bu pointer'in bellekteki adresini, işaret ettiği değişkenin değerini ve bu değişkenin adresini göstersin.

```
#include<stdio.h>
void pointer_detayi_goster( int * );
int main( void )
{
        int sayi = 15;
        int *pointer;
        // Degisken isaret ediliyor.
        pointer = &sayi;
        // Zaten pointer oldugu icin '&'
        // isaretine gerek yoktur. Eger
        // bir degisken olsaydi, basina '&'
        // koymamiz gerekirdi.
        pointer_detayi_goster( pointer );

        return 0;
}
void pointer_detayi_goster( int *p )
{
        // %p, bellek adreslerini gostermek icindir.
        // 16 tabaninda (Hexadecimal) sayilar icin kullanilir.
        // %p yerine, %x kullanmaniz mumkundur.
        printf( "Pointer adresi\t\t\t: %p\n", &p );
        printf( "İşaret ettiği değişkenin adresi\t: %p\n", p );
        printf( "İşaret ettiği değişkenin değeri\t: %d\n", *p );
}
```

Fonksiyon prototipi, "*Function Prototype*"dan geliyor. Bunun güzel bir çeviri olduğunu düşünmüyorum. Ama aklıma daha uygun bir şey gelmedi. Öneriniz varsa değiştirebiliriz.

Rekürsif Fonksiyonlar

Bir fonksiyon içersinden, bir diğerini çağırabiliriz. Rekürsif fonksiyonlar, fonksiyon içersinden fonksiyon çağırmanın özel bir hâlidir. Rekürsif fonksiyon bir başka fonksiyon yerine kendisini çağırır ve şartlar uygun olduğu sürece bu tekrarlanır. Rekürsif, Recursive kelimesinden geliyor ve tekrarlamalı, yinelemeli anlamını taşıyor. Kelimenin anlamıyla, yaptığı iş örtüşmekte.

Rekürsif fonksiyonları aklımızdan çıkartıp, bildiğimiz yöntemle 1, 5, 9, 13 serisini oluşturan bir fonksiyon yazalım:

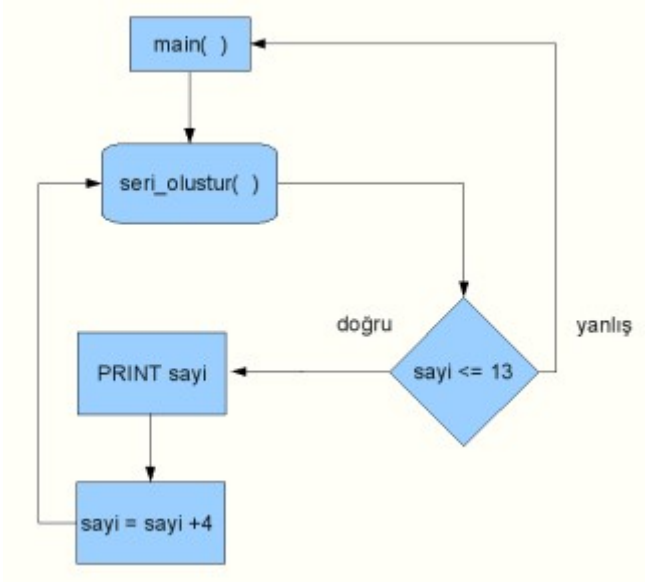
```
#include<stdio.h>
void seri_olustur( int );
int main( void )
{
        seri_olustur( 1 );
}
void seri_olustur( int sayi )
{
        while( sayi <= 13 ) {
                printf("%d ", sayi );
                sayi += 4;
        }
}
```

Bu fonksiyonu yazmak oldukça basitti. Şimdi aynı işi yapan rekürsif bir fonksiyon yazalım:

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

```
void seri_olustur( int );
int main( void )
{
        seri_olustur( 1 );
}
void seri_olustur( int sayi )
{
        if( sayi <= 13 ) {
                printf("%d ", sayi );
                sayi += 4;
                seri_olustur( sayi );
        }
}
```

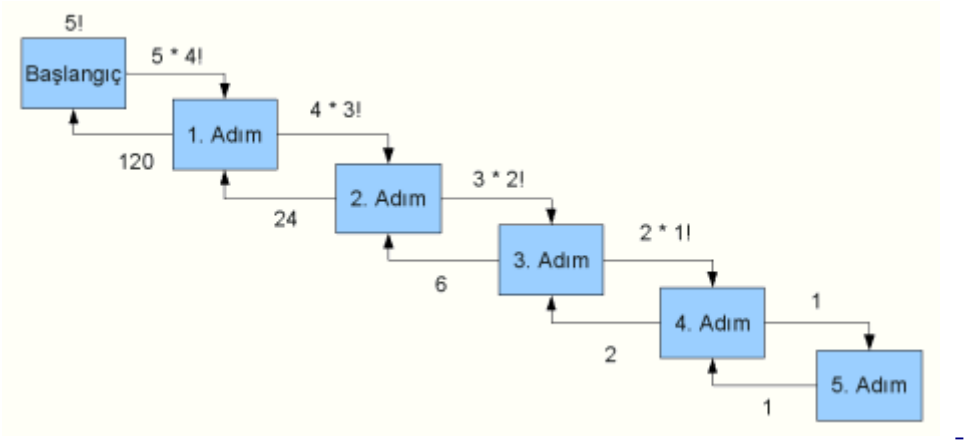
Son yazdığımız programla, bir önce yazdığımız program aynı çıktıları üretir. Ama birbirlerinden farklı çalışırlar. İkinci programın farkını akış diyagramına bakarak sizler de görebilirsiniz. Rekürsif kullanım, fonksiyonun tekrar tekrar çağrılmasını sağlamıştır.



Daha önce faktöriyel hesabı yapan program yazmıştık. Şimdi faktöriyel hesaplayan fonksiyonu, rekürsif olarak yazalım:

```
#include<stdio.h>
int faktoriyel( int );
int main( void )
{
        printf( "%d\n", faktoriyel(5) );
}
int faktoriyel( int sayi )
{
        if( sayi > 1 )
                return sayi * faktoriyel( sayi-1 );
        return 1;
}
```

Yukardaki programın detaylı bir şekilde akış diyagramını vermeyeceğim. Ancak faktöriyel hesaplaması yapılırken, adımları görmenizi istiyorum. Adım olarak geçen her kutu, fonksiyonun bir kez çağrılmasını temsil ediyor. Başlangıç kısmını geçerseniz fonksiyon toplamda 5 kere çağrılıyor.



Rekürsif yapılar, oldukça karmaşık olabilir. Fakat kullanışlı oldukları kesin. Örneğin silme komutları rekürsif yapılardan yararlanır. Bir klasörü altında bulunan her şeyle birlikte silmeniz gerekiyorsa, rekürsif fonksiyon kaçınılmazdır. Ya da bazı matematiksel işlemlerde veya arama (search) yöntemlerinde yine rekürsif fonksiyonlara başvururuz. Bunların dışında rekürsif fonksiyonlar, normal fonksiyonlara göre daha az kod kullanılarak yazılır. Bunlar rekürsif fonksiyonların olumlu yönleri... Ancak hiçbir şey mükemmel değildir.

Rekürsif fonksiyon kullanmanın bilgisayarınıza bindereceği yük daha fazladır. Faktoriyel örneğine bakın; tam 5 kez aynı fonksiyonu çağırıyoruz ve bu sırada bütün değerler bellekte tutuluyor. Eğer çok sayıda iterasyondan söz ediyorsak, belleğiniz hızla tükenecektir. Rekürsif yapılar, bellekte ekstra yer kapladığı gibi, normal fonksiyonlara göre daha yavaştır. Üstelik kısa kod yazımına karşın, rekürsif fonksiyonların daha karmaşık olduklarını söyleyebiliriz. Anlamak zaman zaman sorun olabiliyor. Kısacası bir programda gerçekten rekürsif yapıya ihtiyacınız olmadığı sürece, ondan kaçınmanız daha iyi!

Örnek Sorular

Soru 1: Aşağıdaki programa göre, a, b ve c'nin değerleri nedir?

```
#include<stdio.h>
int main( void )
{
        float a, b, c;
        float *p;
        a = 15.4;
        b = 48.6;
        p = &a;
        c = b = *p = 151.52;
        printf( "a: %f, b: %f, c: %f\n", a, b, c );
        return 0;
}
```

Cevap 1:

```
a: 151.52, b: 151.52, c: 151.52
```

Soru 2: Fibonnacci serisinde herhangi bir seri elemanın değerini bulmak için f(n) = f(n - 1) + f(n - 2) fonksiyonu kullanılır. Başlangıç değeri olarak f(0) = 0 ve f(1) = 1'dir. Bu bilgiler ışığında, verilen n sayısına göre, seride karşılık düşen değeri bulan fonksiyonu rekürsif olarak yazınız.

Cevap 2:

```
#include<stdio.h>
int fibonacci( int );
int main( void )
```

```
{
        int i;
        // Fibonacci serisinin ilk 10 elemani
        // yazilacaktir.
        for( i = 0; i < 10; i++ ) {
                printf( "f(%d)= %d\n", i, fibonacci( i ) );
        }
        return 0;
}
int fibonacci( int eleman_no )
{
        if( eleman_no > 1 ) {
        return fibonacci( eleman_no - 1 ) +
                fibonacci( eleman_no - 2 ) ;
        }
        else if( eleman_no == 1 )
                return 1;
        else
                return 0;
}
```

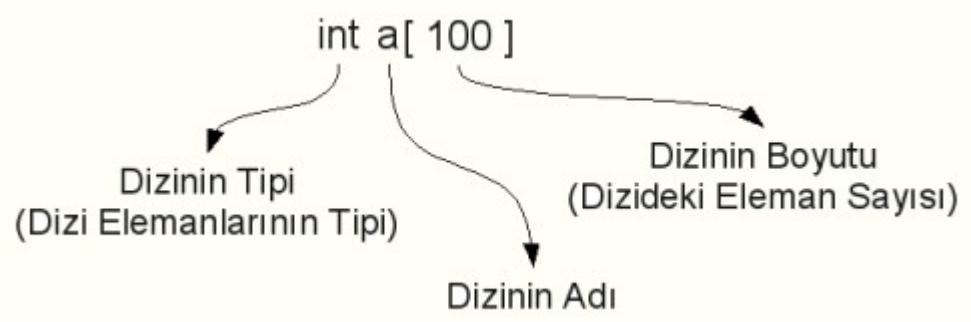
Diziler

Bir bilgisayar programı yaptığınızı düşünün. Kullanıcının 100 değer girmesi isteniyor. Girilen bütün bu sayıların farklı aşamalardan geçeceğini ve bu yüzden hepsini ayrı bir değişkende tutmamız gerektiğini varsayalım. Bu durumda ne yapardınız? a0, a1, a2, ..., a99 şeklinde 100 tane değişken tanımlamak elbette mümkün; ama oldukça zahmetli olurdu. Sırf değişkenleri tanımlarken kaybedeceğiniz zamanı düşünürseniz ne demek istediğimi anlarsınız. Bunun için alternatif bir çözümün gerektiği mutlak!

Çok sayıda değişkenin gerektiği durumlarda, diziler imdadımıza yetişir. (Dizi, İngilizce kaynaklarda array olarak geçer.) 100 değişken tanımlamamızın gerektiği yukardaki örneğe dönelim. Tek tek a0, ..., a100 yaparak bunu nasıl yazacağınızı zaten biliyorsunuz. Şimdi tek satırda dizi tanımlayarak, bunu nasıl yapacağımızı görelim:

```
int a[100];
```

Yukardaki tek satır, bellek bloğunda 100 adet int değişken yeri ayırır. Tek satır kod elbette açıklayıcı değil, o yüzden bunu aşağıdaki şekille açıklayalım:



Her şeyin başında dizideki elemanların değişken tipini yazıyoruz; buradaki örneğimizde tam sayı gerektiği için 'int' yazdık. Daha sonra diziye 'a' yazarak bir isim veriyoruz. Değişkenleri nasıl isimlendiriyorsak, aynı kurallar diziler için de geçerli... Son aşamada bu dizinin kaç eleman içereceğini belirtiyoruz. Köşeli parantezler ([ ]) içinde yazdığımız 100 sayısı, 100 adet int tipinde değişkenin oluşturulmasını sağlıyor.

Bir değişkene ulaşırken, onun adını yazarız. Dizilerde de aşağı yukarı böyle sayılır. Fakat ufak farklar vardır. Bir dizi, birden çok elemandan oluşur. Bu yüzden sadece dizi adını yazmaz, ulaşmak

istediğimiz elemanı da yer numarasını yazarak belirtiriz. Örneğin a dizisinin, 25.elemanı gerekiyorsa, a[24] ile çağrılır. Sanırım 25 yerine 24 yazıldığını fark etmişsinizdir. C programlama dilinde, dizilerin ilk elemanı 0'dır. Diziler 0'dan başladığı için, ulaşmak istenilen dizi elemanı hangisiyse bir eksiğini yazarız. Yani a dizisinin 25.elemanı, a[24]; 100.elemanı a[99] ile çağırırız.

Şimdi basit bir örnek yapalım. Bu örnekte, her aya ait güneşli gün sayısı sorulsun ve sonunda yıllık güneşli gün sayısı yazılsın.

```
#include<stdio.h>
int main( void )
{
        // Aylari temsil etmesi icin
        // aylar adinda 12 elemanli
        // bir dizi olusturuyoruz.
        int aylar[ 12 ];
        int toplam = 0;
        int i;

        // Birinci dongu, deger atamak icindir
        for( i = 0; i < 12; i++ ) {
                printf( "%2d.Ay: ", (i+1) );
                // aylara deger atıyoruz:
                scanf( "%d", &aylar[ i ] );
        }

        // Az evvel girilen degerleri gostermek icin
        // ikinci bir dongu kurduk
        printf( "\nGİRDİĞİNİZ DEĞERLER\n\n" );
        for( i = 0; i < 12; i++ ) {
                printf( "%2d.Ay için %d girdiniz\n", (i+1), aylar[i] );
                toplam += aylar[ i ];
        }

        printf( "Toplam güneşli gün sayısı: %d\n", toplam );
        return 0;
}
```

Örneğimizi inceleyelim. En başta 12 elemanlı *aylar* dizisini, "*int aylar[12]*" yazarak tanımlıyoruz. Her ay için değer girilmesini gerekiyor. Klavyeden girilen sayıların okunması için elbette scanf( ) fonksiyonunu kullanacağız ama ufak bir farkla! Eğer 'a' isimde bir değişkene atama yapıyor olsaydık, hepinizin bileceği şekilde "*scanf("%d", &a)*" yazardık. Fakat dizi elemanlarına atama yapılacağından komutu, "*scanf("%d", &aylar[i])*" şeklinde yazmamız gerekiyor. Döngü içindeki *i* değişkeni, 0'dan 11'e kadar sırasıyla artıyor. Bu sayede, döngünün her adımında farklı bir dizi elemanına değer atıyabiliyoruz. (*i* değişkeni, bir nevi indis olarak düşünülebilir.) Klavyeden okunan değerlerin dizi elemanlarına atanmasından sonra, ikinci döngü başlıyor. Bu döngüde girmiş olduğunuz değerler listelenip, aynı esnada toplam güneşli gün sayısı bulunuyor. Son aşamada, hesaplanan toplam değerini yazdırıp, programı bitiriyoruz.

Dikkat ederseniz, değerlerin alınması veya okunması gibi işlemler döngüler aracılığıyla yapıldı. Bunları döngüler aracılığı ile yapmak zorunda değildik. Mesela "*scanf("%d", &aylar[5])*" yazıp, 6.ayın değerini; "*scanf("%d", &aylar[9])*" yazıp, 10.ayın değerini klavyeden alabilirdik. Ancak böyle yapmak, döngü kullanmaktan daha zahmetlidir. Yanılgıya düşmemeniz için döngüleri kullanmanın kural olmadığını, sadece işleri kolaylaştırdığını hatırlatmak istedim. Gerek tek tek, gerekse örnekte yaptığımız gibi döngülerle çözüm üretebilirsiniz.

Başka bir örnek yapalım. Kullanıcımız, float tipinde 10 adet değer girsin. Önce bu değerlerin ortalaması bulunsun, ardından kaç adet elemanın ortalamanın altında kaldığı ve kaç adet elemanın ortalamanın üstünde olduğu gösterilsin.

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

```
int main( void )
{
        // Degerleri tutacagimiz 'dizi'
        // adinda bir dizi olusturuyoruz.
        float dizi[ 10 ];
        float ortalama, toplam = 0;
        int ortalama_ustu_adedi = 0;
        int ortalama_alti_adedi = 0;
        int i;

        // Kullanici dizinin elemanlarini giriyor:
        for( i = 0; i < 10; i++ ) {
                printf( "%2d. elemanı giriniz> ", (i+1) );
                scanf( "%f", &dizi[ i ] );
                toplam += dizi[ i ];
        }

        // dizinin ortalamasi hesaplaniyor.
        ortalama = toplam / 10.0;

        // ortalamadan kucuk ve buyuk elemanlarin
        // kac adet oldugu belirleniyor.
        for( i = 0; i < 10; i++ ) {
                if( dizi[ i ] < ortalama )
                        ortalama_alti_adedi++;
                else if( dizi[ i ] > ortalama )
                        ortalama_ustu_adedi++;
        }

        // raporlama yapiliyor.
        printf( "Ortalama: %.2f\n", ortalama );
        printf( "Ortalamadan düşük %d eleman vardır.\n", ortalama_alti_adedi );
        printf( "Ortalamadan yüksek %d eleman vardır.\n", ortalama_ustu_adedi );

        return 0;
}
```

Program pek karmaşık değil. Dizi elemanlarını alıyor, ortalamalarını hesaplıyor, elemanları ortalamayla karşılaştırıp, ortalamadan büyük ve küçük elemanların adedini veriyoruz. Anlaşılması güç bir şey bulacağınızı sanmıyorum. Tek karmaşık gelecek nokta, ikinci döngüde neden bir *else* olmadığı olabilir. Oldukça geçerli bir sebebi var ve *if else-if* yapısını iyice öğrenenler böyle bırakılmasını anlayacaklardır. Bilmeyenlere gelince... her şeyi ben söylersem, işin tadı tuzu kalmaz; eski konuları gözden geçirmelisiniz.

Dizilere İlk Değer Atama

Değişken tanımı yaparken, ilk değer atamayı biliyoruz. Örneğin "*int a = 5;*" şeklinde yazacağınız bir kod, *a* değişkenini oluşturacağı gibi, içine 5 değerini de atayacaktır. (Bu değişkene, tanımladıktan sonra farklı değerler atayabilirsiniz.) Benzer şekilde, bir diziyi tanımlarken, dizinin elemanlarına değer atayabilirsiniz. Aşağıda bunu nasıl yapabileceğinizi görebilirsiniz:

```
int dizi1[ 6 ] = { 4, 8, 15, 16, 23, 42 };
float dizi2[ 5 ] = { 11.5, -1.6, 46.3, 5, 21.56 };
```

Küme parantezleri içinde gördüğünüz her değer, sırasıyla bir elemana atanmıştır. Örneğin dizi1'in ilk elemanı 4 olurken, son elemanı 42'dir.

Yukardaki tanımlamalarda farkedeceğiniz gibi dizi boyutlarını 6 ve 5 şeklinde belirttik. İlk değer ataması yapacağımız durumlarda, dizinin boyutunu belirtmeniz gerekmez; dizi boyutunu yazıp yazmamak size bağlıdır. Dilerseniz dizi boyutunu belirtmeden aşağıdaki gibi de yazabilirdiniz:

```
int dizi1[ ] = { 4, 8, 15, 16, 23, 42 };
float dizi2[ ] = { 11.5, -1.6, 46.3, 5, 21.56 };
```

Derleyici, atanmak istenen değer sayısına bakar ve *dizi1* ile *dizi2*'nin boyutunu buna göre belirler. *dizi1* için 6, *dizi2* için 5 tane değer belirtmişiz. Bu *dizi1* dizisinin 6, *dizi2* dizisinin 5 elemanlı olacağını işaret eder.

Değer atamayla ilgili ufak bir bilgi daha aktarmak istiyorum. Aşağıda iki farklı ilk değer atama yöntemi bulunuyor. Yazım farkı olmasına rağmen, ikisi de aynı işi yapar.

```
int dizi[ 7 ] = { 0, 0, 0, 0, 0, 0, 0 };

int dizi[ 7 ] = { 0 };
```

Bir diziyi tanımlayın ve hiçbir değer atamadan, dizinin elemanlarını printf( ) fonksiyonuyla yazdırın. Ortaya anlamsız sayılar çıktığını göreceksiniz. Bir dizi tanımlandığında, hafızada gerekli olan yer ayrılır. Fakat daha önce bu hafıza alanında ne olup olmadığıyla ilgilenilmez. Ortaya çıkan anlamsız sayılar bundan kaynaklanır. Önceden hafızada bulunan değerlerin yansımasını görürsünüz. Modern programlama dillerinin bir çoğunda, dizi tanımladığınızda, dizinin bütün elemanları 0 değeriyle başlar; sizin herhangi bir atama yapmanıza gerek yoktur. C programlama dilindeyse, kendiliğinden bir başlangıç değeri atanmaz. Bunu yapıp yapmamak size kalmıştır. Kısacası işlerin daha kontrolü gitmesini istiyorsanız, dizileri tanımlarken "*dizi[7] = { 0 };*" şeklinde tanımlamalar yapabilirsiniz.

İlk değer atanmasıyla ilgili bir örnek yapalım. 10 elemanlı bir diziye atadığımız ilk değerin maksimum ve minimum değerleri gösterilsin:

```
#include<stdio.h>
int main( void )
{
        // dizi'yi tanitirken, ilk deger
        // atiyoruz
        int dizi[ ] = { 15, 54, 1, 44, 55,
                        40, 60, 4, 77, 45 };
        int i, max, min;

        // Dizinin ilk elemanini, en kucuk
        // ve en buyuk deger kabul ediyoruz.
        // Bunun yanlis olmasi onemli degil;
        // sadece bir noktadan kontrole baslamamiz
        // gerektiginden boyle bir secim yaptik.
        min = dizi[ 0 ];
        max = dizi[ 0 ];

        for( i = 1; i < 10; i++ ) {
                // min'in degeri, dizi elemanindan
                // buyukse, min'in degerini degistiririz.
                // Kendisinden daha kucuk sayi oldugunu
                // gosterir.
                if( min > dizi[i] )
                        min = dizi[i];

                // max'in degeri, dizi elemanindan
                // kucukse, max'in degerini degistiririz.
                // Kendisinden daha buyuk sayi oldugunu
                // gosterir.
                if( max < dizi[i] )
                        max = dizi[i];
        }

        printf( "En Küçük Değer: %d\n", min );
```

```
        printf( "En Büyük Değer: %d\n", max );

        return 0;
}
```

Dizilerin fonksiyonlara aktarımı

Dizileri fonksiyonlara aktarmak, tıpkı değişkenleri aktarmaya benzemektedir. Uzun uzadıya anlatmak yerine, örnek üstünden gitmenin daha fayda olacağını düşünüyorum. Bir fonksiyon yazalım ve bu fonksiyon kendisine gönderilen dizinin elemanlarını ekrana yazsın.

```
#include<stdio.h>
void elemanlari_goster( int [ 5 ] );
int main( void )
{
        int dizi[ 5 ] = { 55, 414, 7, 210, 15 };
        elemanlari_goster( dizi );
        return 0;
}
void elemanlari_goster( int gosterilecek_dizi[ 5 ] )
{
        int i;
        for( i = 0; i < 5; i++)
                printf( "%d\n", gosterilecek_dizi[ i ] );
}
```

Fonksiyon prototipini yazarken, dizinin tipini ve boyutunu belirttiğimizi fark etmişsinizdir. Fonksiyonu tanımlama aşamasında, bunlara ilaveten parametre olarak dizinin adını da yazıyoruz. (Bu isim herhangi bir şey olabilir, kendisine gönderilecek dizinin adıyla aynı olması gerekmez.) Bir dizi yerine sıradan bir değişken kullansaydık, benzer şeyleri yapacaktık. Farklı olan tek nokta; dizi eleman sayısını belirtmemiz. Şimdi main( ) fonksiyonuna dönelim ve elemanlari_goster( ); fonksiyonuna bakalım. Anlayacağınız gibi, "*dizi*" ismindeki dizinin fonksiyona argüman olarak gönderilmesi için sadece adını yazmamız yeterli.

Fonksiyonlarla ilgili bir başka örnek yapalım. Bu sefer üç fonksiyon oluşturalım. Birinci fonksiyon kendisine gönderilen dizideki en büyük değeri bulsun; ikinci fonksiyon, dizinin en küçük değerini bulsun; üçüncü fonksiyon ise elemanların ortalamasını döndürsün.

```
#include<stdio.h>
float maksimum_bul( float [ 8 ] );
float minimum_bul( float [ 8 ] );
float ortalama_bul( float [ 8 ] );
int main( void )
{
        // 8 boyutlu bir dizi olusturup buna
        // keyfi degerler atiyoruz.
        float sayilar[ 8 ] = {  12.36, 4.715, 6.41, 13,
                                1.414, 1.732, 2.236, 2.645 };
        float max, min, ortalama;
        // Ornek olmasi acisindan fonksiyonlar
        // kullaniliyor.
        max = maksimum_bul( sayilar );
        min = minimum_bul( sayilar );
        ortalama = ortalama_bul( sayilar );
        printf( "Maksimum: %.2f\n", max );
        printf( "Minimum: %.2f\n", min );
        printf( "Ortalama: %.2f\n", ortalama );

        return 0;
}
/* Dizi icindeki maksimum degeri bulur */
```

```
float maksimum_bul( float dizi[ 8 ] )
{
        int i, max;
        max = dizi[0];
        for( i = 1; i < 8; i++ ) {
                if( max < dizi[ i ] )
                        max = dizi[ i ];
        }
        return max;
}
/* Dizi icindeki minimum degeri bulur */
float minimum_bul( float dizi[ 8 ] )
{
        int i, min;
        min = dizi[ 0 ];
        for( i = 1; i < 8; i++ ) {
                if( min > dizi[ i ] ) {
                        min = dizi[ i ];
                }
        }
        return min;
}
/* Dizi elemanlarinin ortalamasini bulur */
float ortalama_bul( float dizi[ 8 ] )
{
        int i, ortalama = 0;
        for( i = 0; i < 8; i++ )
                ortalama += dizi[ i ];

        return ortalama / 8.0;
}
```

Yukardaki örneklerimizde, dizilerin boyutlarını bilerek fonksiyonlarımızı yazdık. Ancak gerçek hayat böyle değildir; bir fonksiyona farklı farklı boyutlarda diziler göndermeniz gerekir. Mesela *ortalama_bul( )* fonksiyonu hem 8 elemanlı bir diziye hizmet edecek, hem de 800 elemanlı bir başka diziye uyacak şekilde yazılmalıdır. Son örneğimizi her boyutta dizi için kullanılabilecek hâle getirelim ve *ortalama_bul( )*, *minimum_bul( )* ve *maksimum_bul( )* fonksiyonlarını biraz değiştirelim.

```
#include<stdio.h>
float maksimum_bul( float [ ], int );
float minimum_bul( float [ ], int );
float ortalama_bul( float [ ], int );
int main( void )
{
        // 8 boyutlu bir dizi olusturup buna
        // keyfi degerler atiyoruz.
        float sayilar[ 8 ] = {  12.36, 4.715, 6.41, 13,
                                1.414, 1.732, 2.236, 2.645 };
        float max, min, ortalama;
        // Ornek olmasi acisindan fonksiyonlar
        // kullaniliyor.
        max = maksimum_bul( sayilar, 8 );
        min = minimum_bul( sayilar, 8 );
        ortalama = ortalama_bul( sayilar, 8 );
        printf( "Maksimum: %.2f\n", max );
        printf( "Minimum: %.2f\n", min );
        printf( "Ortalama: %.2f\n", ortalama );

        return 0;
}
/* Dizi icindeki maksimum degeri bulur */
float maksimum_bul( float dizi[ ], int eleman_sayisi )
```

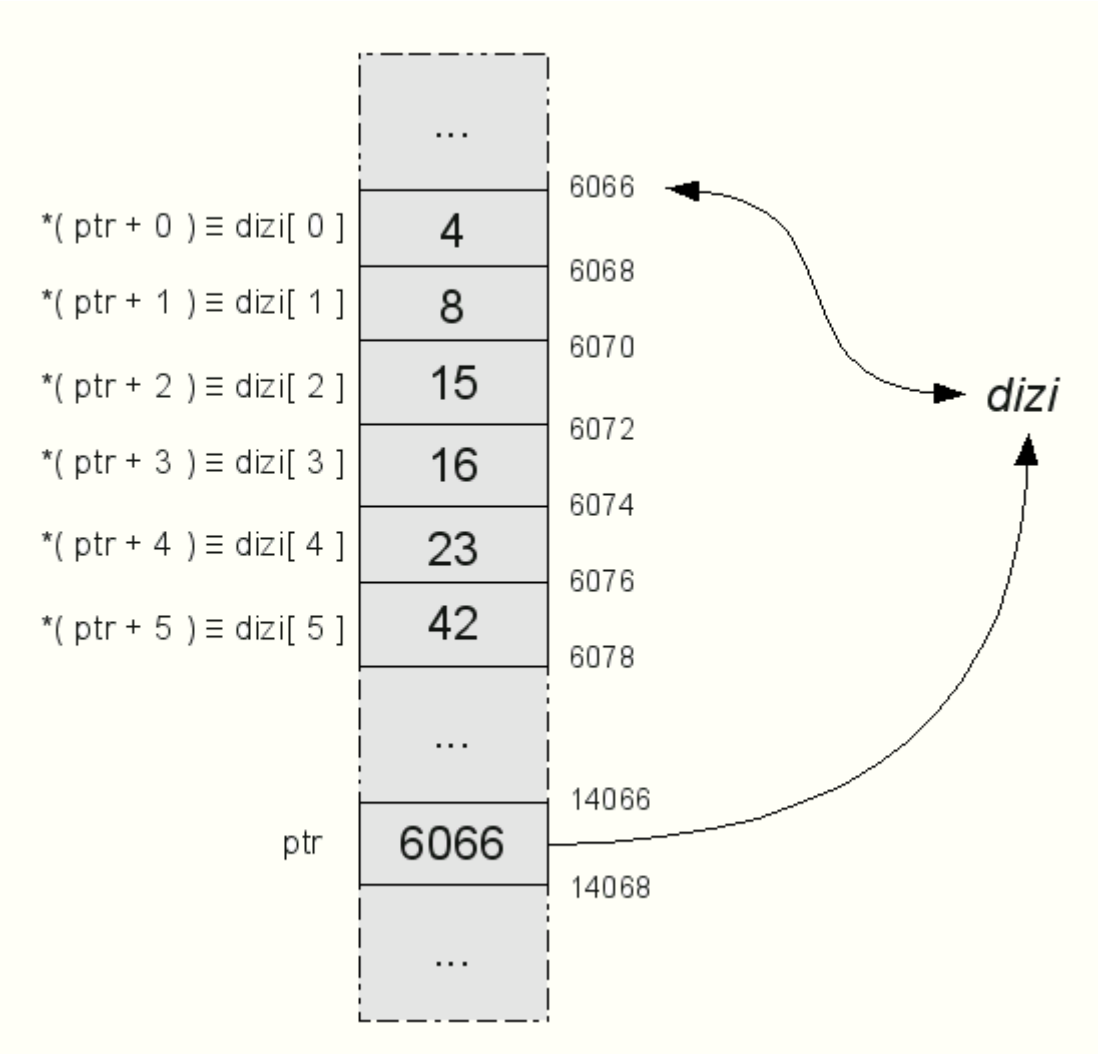
```
{
        int i, max;
        max = dizi[0];
        for( i = 1; i < eleman_sayisi; i++ ) {
                if( max < dizi[ i ] )
                        max = dizi[ i ];
        }
        return max;
}
/* Dizi icindeki minimum degeri bulur */
float minimum_bul( float dizi[ ], int eleman_sayisi )
{
        int i, min;
        min = dizi[ 0 ];
        for( i = 1; i < eleman_sayisi; i++ ) {
                if( min > dizi[ i ] ) {
                        min = dizi[ i ];
                }
        }
        return min;
}
/* Dizi elemanlarinin ortalamasini bulur */
float ortalama_bul( float dizi[ ], int eleman_sayisi )
{
        int i, ortalama = 0;
        for( i = 0; i < eleman_sayisi; i++ )
                ortalama += dizi[ i ];

        return ortalama / 8.0;
}
```

Fonksiyonlara dikkatlice bakın. Geçen sefer sadece dizi adını gönderirken, artık dizi adıyla birlikte boyutunu da yolluyoruz. Böylece dizinin boyutu n'olursa olsun, fark etmiyor. Yeni bir parametre açıp dizinin eleman sayısını isterseniz, fonksiyon her dizi için çalışabilir.

Dizilere Pointer ile Erişim

Pointer'ların değişkenleri işaret etmesini geçen dersimizde işlemiştik. Benzer şekilde dizileri de işaret edebilirler. Örneğin, *int dizi[6];* şeklinde tanımlanmış bir diziyi, pointer ile işaret etmek istersek, *ptr = dizi;* yazmamız yeterlidir. Değişkenlerde, değişken adının başına '&' işareti getiriyorduk, fakat dizilerde buna gerek yoktur. Çünkü dizilerin kendisi de bir pointer'dır. Dizilerin hepsi hafıza alanında bir başlangıç noktası işaret eder. Örnek olması açısından bu noktaya 6066 diyelim. Siz "*dizi[0]*" dediğiniz zaman 6066 ile 6068 arasında kalan bölgeyi kullanırsınız. Ya da "*dizi[4]*" dediğiniz zaman 6074 ile 6076 hafıza bölgesi işleme alınır.

Bir diziyi işaret eden pointer aynen dizi gibi kullanılabilir. Yani *ptr = dizi;* komutunu vermenizden sonra, ptr[0] ile dizi[0] birbirinin aynısıdır. Eğer *ptr yazarsanız, yine dizinin ilk elemanı dizi[0]'ı işaret etmiş olursunuz. Ancak dizi işaret eden pointer'lar genellikle, *(ptr + 0) şeklinde kullanılır. Burada 0 yerine ne yazarsanız, dizinin o elemanını elde ederseniz. Diyelim ki, 5.elemanı (yani dizi[4]) kullanmak istiyorsunuz, o zaman *(ptr + 4) yazarsanız. Bir resim, bin sözden iyidir... Aşağıdaki resmi incelerseniz, durumu daha net anlayacağınızı düşünüyorum.



Gördüğünüz gibi *dizi*, 6066 numaralı hafıza adresini işaret ediyor. *ptr* isimli pointer ise, dizi üzerinden 6066 numaralı adresi gösteriyor. Kısacası ikisi de aynı noktayı işaret ediyorlar. Şimdi bunu koda dökelim:

```
#include<stdio.h>
int main( void )
{
        int i;
        // dizi'yi tanimliyoruz.
        int dizi[ 6 ] = { 4, 8, 15, 16, 23, 42 };
        // ptr adinda bir pointer tanimliyoruz.
        int *ptr;
        // ptr'nin dizi'yi isaret etmesini soyluyoruz.
        ptr = dizi;
        // ptr'in degerini artirip, dizi'nin butun
        // elemanlarini yazdiriyoruz.
        for( i = 0; i < 6; i++ )
                printf( "%d\n", *( ptr + i ) );

        return 0;
}
```

Pointer'lar farklı şekillerde kullanılabilir. Her defasında, dizinin başlangıç elemanını atamanız gerekmez. Örneğin, *ptr = &dizi[2]* şeklinde bir komut kullanarak, dizinin 3.elemanının adresini pointer'a atayabilirsiniz. Pointer'larin değişik kullanım çeşitlerini aşağıda görebilirsiniz:

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

```
{
        int elm;
        int month[ 12 ];
        int *ptr;
        ptr = month; // month[0] adresini atar
        elm = ptr[ 3 ]; // elm = month[ 3 ] değerini atar
        ptr = month + 3; // month[ 3 ] adresini atar
        ptr = &month[ 3 ]; // month[ 3 ] adresini atar
        elm = ( ptr+2 )[ 2 ]; // elm = ptr[ 4 ] ( = month[ 7 ] )değeri atanır.
        elm = *( month + 3 );
        ptr = month;
        elm = *( ptr + 2 ); // elm = month[ 2 ] değerini atar
        elm = *( month + 1 ); // elm = month[ 1 ] değerini atar

        return 0;
}
```

Dizilerin fonksiyonlara gönderilmesini görmüştük. Parametre kısmına dizinin tipini ve boyutunu yazıyorduk. Ancak bunun yerine pointer da kullanabiliriz. Örneğin aşağıdaki iki komut satırı birbirinin aynısıdır.

```
int toplam_bul( int dizi[ ], int boyut );

int toplam_bul( int *dizi, int boyut );
```

Fonksiyondan Dizi Döndürmek

Fonksiyondan bir diziyi döndürmeden önce önemli bir konuyla başlayalım. Hatırlarsanız fonksiyonlara iki şekilde argüman yolluyorduk. Birinci yöntemde, sadece değer gidiyordu ve değişken üzerinde bir değişiklik olmuyordu. (Call by Value) İkinci yöntemdeyse, yollanılan değişken, fonksiyon içersinde yapacağınız işlemlerden etkileniyordu. (Call by Reference) Dizilerin aktarılması, referans yoluyla olur. Fonksiyon içersinde yapacağınız bir değişiklik, dizinin aslında da değişikliğe sebep olur. Aşağıdaki örneğe bakalım:

```
#include<stdio.h>
/* Kendisine verilen dizinin butun
   elemanlarinin karesini alir */
void karesine_donustur( int [ ], int );
int main( void )
{
        int i;
        int liste[ ] = { 1,2,3,4,5,6,7 };
        for( i = 0; i < 7; i++ ) {
                printf( "%d ", liste[ i ] );
        }
        printf("\n");

        // Fonksiyonu cagiriyoruz. Fonksiyon geriye
        // herhangi bir deger dondurmuyor. Diziler
        // referans yontemiyle aktarildigi icin dizinin
        // degeri bu sekilde degismis oluyor.
        karesine_donustur( liste, 7 );
        for( i = 0; i < 7; i++ ) {
                printf( "%d ", liste[ i ] );
        }
        printf("\n");
        return 0;
}
void karesine_donustur( int dizi[ ], int boyut )
{
        int i;
```

```
        for( i = 0; i < boyut; i++ ) {
                dizi[ i ] = dizi[ i ] * dizi[ i ];
        }
}
```

Gördüğünüz gibi fonksiyon içersinde diziyle ilgili yaptığımız değişikler, dizinin aslına da yansımıştır. Sırada, fonksiyondan dizi döndürmek var.

Aslında fonksiyondan dizi pek doğru bir isimlendirme değil. Gerçekte döndürdüğümüz şey, dizinin kendisi değil, sadece başlangıç adresi oluyor. Dolayısıyla bir dizi döndürdüğümüzü söylemek yerine, Pointer döndürdüğümüzü söyleyebiliriz. Basit bir fonksiyon hazırlayalım; bu fonksiyon kendisine gönderilen iki diziyi birleştirip, tek bir dizi hâline getirsin.

```
#include<stdio.h>
/* Kendisine verilen iki diziyi birlestirip
   sonuclari ucuncu bir diziye atar */
int *dizileri_birlestir( int [], int,
                         int [], int,
                         int []);
int main( void )
{
        // liste_1, 5 elemanli bir dizidir.
        int liste_1[5] = { 6, 7, 8, 9, 10 };
        // liste_2, 7 elemanli bir dizidir.
        int liste_2[7] = {13, 7, 12, 9, 7, 1, 14 };
        // sonuclarin toplanacagi toplam_sonuc dizisi
        int toplam_sonuc[13];
        // sonucun dondurulmesi icin pointer tanimliyoruz
        int *ptr;
        int i;

        // fonksiyonu calistiriyoruz.
        ptr = dizileri_birlestir( liste_1, 5, liste_2, 7,
                                  toplam_sonuc );

        // pointer uzerinden sonuclari yazdiriyoruz.
        for( i = 0; i < 12; i++ )
                printf("%d ", *(ptr+i) );
        printf("\n");

        return 0;
}
int *dizileri_birlestir( int dizi_1[], int boyut_1,
                         int dizi_2[], int boyut_2,
                         int sonuc[] )
{
        int i, k;
        // Birinci dizinin degerleri ataniyor.
        for( i = 0; i < boyut_1; i++ )
                sonuc[i] = dizi_1[i];

        // Ikinci dizinin degerleri ataniyor.
        for( k = 0; k < boyut_2; i++, k++ ) {
                sonuc[i] = dizi_2[k];
        }

        // Geriye sonuc dizisi gonderiliyor.
        return sonuc;
}
```

Neyin nasıl olduğunu sanırım anlamışsınızdır. Diziler referans yoluyla gönderilirken ve gönderdiğimiz dizilerin boyutları belliyken, neden bir de işin içine pointer'ları soktuğumuzu

sorabilirsiniz. İlerki konumuzda, dinamik yer ayırma konusunu işleyeceğiz. Şimdilik çok lüzumlu gözükmese de, ön hazırlık olarak olarak bunları öğrenmeniz önemli!

Sıralama

Sıralama oldukça önemli bir konudur. Çeşit çeşit algoritmalar (QuickSort, Insertion, Shell Sort, vs...) bulunmaktadır. Ben sizlere en basit sıralama yöntemlerinden biri olan, "*Bubble Sort*" ("*Kabarcık Sıralaması*") metodundan bahsedeceğim.

Elinizde, {7, 3, 66, 3, -5, 22, -77, 2} elemanlarından oluşan bir dizi olduğunu varsayın. Dizinin en sonuna gidiyorsunuz ve 8.elemanla (*dizi[7]*), 7.elemanı (*dizi[6]*) karşılaştırıyorsunuz. Eğer 8.eleman, 7.elemandan küçükse bu ikisi yer değiştiriyor; değilse, bir değişiklik yapmıyorsunuz. Sonra 7.elemanla (*dizi[6]*) 6.eleman için aynı işlemler yapılıyor. Bu böyle dizinin son elemanına (*dizi[0]*) kadar gidiyor. Buraya kadar yaptığımız işlemlere birinci aşama diyelim. İkinci aşamada da tamamen aynı işlemleri yapıyorsunuz. Sadece süreç dizinin son elemanına (*dizi[0]*) kadar değil, ondan bir önceki elemana kadar sürüyor. Kısacası her aşamada, kontrol ettiğiniz eleman sayısını bir azaltıyorsunuz. Aşama sayısı da, dizi eleman sayısının bir eksiği oluyor. Yani bu örneğimizde 7 aşama gerekiyor

Konu biraz karmaşık; tek seferde anlaşılmayabilir. Bu dediklerimizi algoritmaya dökelim:

```
#include<stdio.h>
void dizi_goster( int [ ], int );
void kabarcik_siralamasi( int [ ], int );
int main( void )
{
        int i, j;
        int dizi[ 8 ] = { 7, 3, 66, 3,
                          -5, 22, -77, 2 };

        // Siralama islemi icin fonksiyonu
        // cagriyoruz.
        kabarcik_siralamasi( dizi, 8 );
        // Sonucu gostermesi icin dizi_gosteri
        // calistiriyoruz.
        dizi_goster( dizi, 8 );
        return 0;
}
// Dizi elemanlarini gostermek icin yazilmis
// bir fonksiyondur.
void dizi_goster( int dizi[ ], int boyut )
{
        int i;
        for( i = 0; i < boyut; i++ ) {
                printf("%d ",dizi[i]);
        }
        printf("\n");
}
// Bubble Sort algoritmasina gore, siralama islemi
// yapar.
void kabarcik_siralamasi( int dizi[ ], int boyut )
{
        int i, j, temp;
        // Ilk dongu asama sayisini temsil ediyor.
        // Bu donguye gore, ornegin boyutu 8 olan
        // bir dizi icin 7 asama gerceklesir.
        for( i = 0; i < boyut-1; i++ ) {
                // Ikinci dongu, her asamada yapilan
                // islemi temsil eder. Dizinin elemanlari
                // en sondan baslayarak kontrol edilir.
                // Eger kendisinden once gelen elemandan
```

```
                // kucuk bir degeri varsa, elemanlarin
                // degerleri yer degistirir.
                for( j = boyut - 1; j > i; j-- )  {
                        if( dizi[ j ] < dizi[ j - 1 ] ) {
                                temp = dizi[ j -1 ];
                                dizi[ j - 1 ] = dizi[ j ];
                                dizi[ j ] = temp;
                        }

                }
        }
}
```

Örnek Sorular

Soru 1: Kendisine parametre olarak gelen bir diziyi, yine parametre olarak bir başka diziye ters çevirip atayacak bir fonksiyon yazınız.

Cevap 1:

```
#include<stdio.h>
void diziyi_ters_cevir( int[], int[], int );
int main( void )
{
        int i;
        int liste_1[] = { 6, 7, 8, 9, 10 };
        int liste_2[5];

        diziyi_ters_cevir( liste_1, liste_2, 5 );

        for( i = 0; i < 5; i++ ) {
                printf("%d ", liste_2[i]);
        }
        printf("\n");
}
void diziyi_ters_cevir( int dizi_1[], int dizi_2[], int boyut )
{
        int i, k;
        for( i = 0, k = boyut - 1; i < boyut; i++, k-- )
                dizi_2[k] = dizi_1[i];
}
```

Soru 2: Kendisine parametre olarak gelen bir dizinin bütün elemanlarını, mutlak değeriyle değiştiren programı yazınız.

Cevap 2:

```
#include<stdio.h>
void dizi_mutlak_deger( int[], int );
int main( void )
{
        int i;
        int liste[] = { -16, 71, -18, -4, 10, 0 };

        dizi_mutlak_deger( liste, 6 );

        for( i = 0; i < 6; i++ ) {
                printf("%d ", liste[i]);
        }
```

```
        printf("\n");
}
void dizi_mutlak_deger( int dizi[], int boyut )
{
        int i;
        for( i = 0; i < boyut; i++ ) {
                if( dizi[i] < 0 )
                        dizi[i] *= -1;
        }
}
```

Soru 3: int *ptr = &month[3] şeklinde bir atama yapılıyor. Buna göre, aşağıdakilerden hangileri *ptr değerine eşittir?

a) month;
b) ptr[0];
c) ptr[1];
d) *(month + 3)
e) (month + 3)[0];
f) (month + 1)[2];
g) *(month + 3)[0];

Cevap3:

b, d, e ve f şıkları, *ptr'ye eşittir.

Çok Boyutlu Diziler

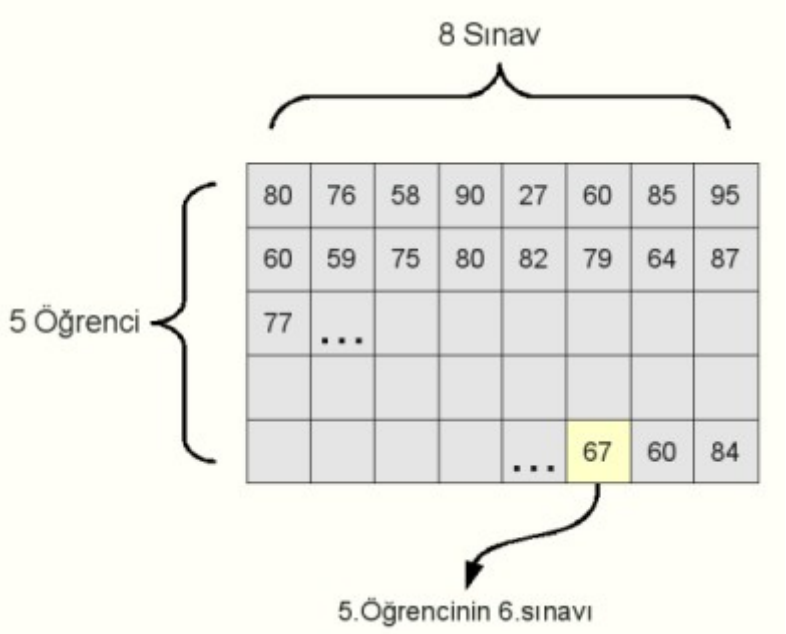
Önceki derslerimizde dizileri görmüştük. Kısaca özetleyecek olursak, belirlediğimiz sayıda değişkeni bir sıra içinde tutmamız, diziler sayesinde gerçekleşiyordu. Bu dersimizde, çok boyutlu dizileri inceleyip, ardından dinamik bellek konularına gireceğiz.

Şimdiye kadar gördüğümüz diziler, tek boyutluydu. Bütün elemanları tek boyutlu bir yapıda saklıyorduk. Ancak dizilerin tek boyutlu olması gerekmez; istediğiniz boyutta tanımlayabilirsiniz. Örneğin 3x4 bir matris için 2 boyutlu bir dizi kullanırız. Ya da üç boyutlu Öklid uzayındaki x, y, z noktalarını saklamak için 3 boyutlu bir diziyi tercih ederiz.

Hemen bir başka örnek verelim. 5 kişilik bir öğrenci grubu için 8 adet test uygulansın. Bunların sonuçlarını saklamak için 2 boyutlu bir dizi kullanalım:

```
#include<stdio.h>
int main( void )
{
        // 5 adet ogrenci icin 8 adet sinavi
        // temsil etmesi icin bir ogrenci tablosu
        // olusturuyoruz. Bunun icin 5x8 bir matris
        // yaratilmasi gerekiyor.
        int ogrenci_tablosu[ 5 ][ 8 ];
        int i, j;
        for( i = 0; i < 5; i++ ) {
                for( j = 0; j < 8; j++ ) {
                        printf( "%d no.'lu ogrencinin ", ( i + 1 ) );
                        printf( "%d no.'lu sınavı> ", ( j + 1 ) );
                        // Tek boyutlu dizilerdeki gibi deger
                        // atiyoruz
                        scanf( "%d", &ogrenci_tablosu[ i ][ j ] );
                }
        }

        return 0;
}
```

Bu programı çalıştırıp, öğrencilere çeşitli değerler atadığımızı düşünelim. Bunu görsel bir şekle sokarsak, aşağıdaki gibi bir çizelge oluşur:



Tabloya bakarsak, 1.öğrenci sınavlardan, 80, 76, 58, 90, 27, 60, 85 ve 95 puan almış gözüküyor. Ya da 5.öğrencinin, 6.sınavından 67 aldığını anlıyoruz. Benzer şekilde diğer hücrelere gerekli değerler atanıp, ilgili öğrencinin sınav notları hafızada tutuluyor.

Çok Boyutlu Dizilere İlk Değer Atama

Çok boyutlu bir diziyi tanımlarken, eleman değerlerini atamak mümkündür. Aşağıdaki örneği inceleyelim:

```
int tablo[3][4] = { 8, 16, 9, 52, 3, 15, 27, 6, 14, 25, 2, 10 };
```

Diziyi tanımlarken, yukardaki gibi bir ilk değer atama yaparsanız, elemanların değeri aşağıdaki gibi olur:

Satır 0	**:**	8	16	9	52
Satır 1	**:**	3	15	27	6
Satır 2	**:**	14	25	2	10

Çok boyutlu dizilerde ilk değer atama, tek boyutlu dizilerdekiyle aynıdır. Girdiğiniz değerler sırasıyla hücrelere atanır. Bunun nedeni de basittir. Bilgisayar, çok boyutlu dizileri sizin gibi düşünmez; dizi elemanlarını hafızada arka arkaya gelen bellek hücreleri olarak değerlendirir.

Çok boyutlu dizilerde ilk değer atama yapacaksanız, değerleri kümelendirmek iyi bir yöntemdir; karmaşıklığı önler. Örneğin yukarıda yazmış olduğumuz ilk değer atama kodunu, aşağıdaki gibi de yazabiliriz:

```
int tablo[3][4] = { {8, 16, 9, 52}, {3, 15, 27, 6}, {14, 25, 2, 10} };
```

Farkedeceğiniz gibi elemanları dörderli üç gruba ayırdık. Bilgisayar açısından bir şey değişmemiş olsa da, kodu okuyacak kişi açısından daha yararlı oldu. Peki ya dört adet olması gereken grubun elemanlarını, üç adet yazsaydık ya da bir-iki grubu hiç yazmasaydık n'olurdu? Deneyelim...

```
int tablo[3][4] = { {8, 16}, {3, 15, 27} };
```

Tek boyutlu dizilerde ilk değer ataması yaparken, eleman sayısından az değer girerseniz, kalan

değerler 0 olarak kabul edilir. Aynı şey çok boyutlu diziler için de geçerlidir; olması gerektiği sayıda eleman ya da grup girilmezse, bu değerlerin hepsi 0 olarak kabul edilir. Yani üstte yazdığımız kodun yaratacağı sonuç, şöyle olacaktır:

Satır 0 : 8 16 0 0

Satır 1 : 3 15 27 0

Satır 2 : 0 0 0 0

Belirtmediğimiz bütün elemanlar 0 değerini almıştır. Satır 2'ninse bütün elemanları direkt 0 olmuştur; çünkü grup tanımı hiç yapılmamıştır.

Fonksiyonlara 2 Boyutlu Dizileri Aktarmak

İki boyutlu bir diziyi fonksiyona parametre göndermek, tek boyutlu diziyi göndermekten farklı sayılmaz. Tek farkı dizinin iki boyutlu olduğunu belirtmemiz ve ikinci boyutun elemanını mutlaka yazmamızdır. Basit bir örnek yapalım; kendisine gönderilen iki boyutlu bir diziyi matris şeklinde yazan bir fonksiyon oluşturalım:

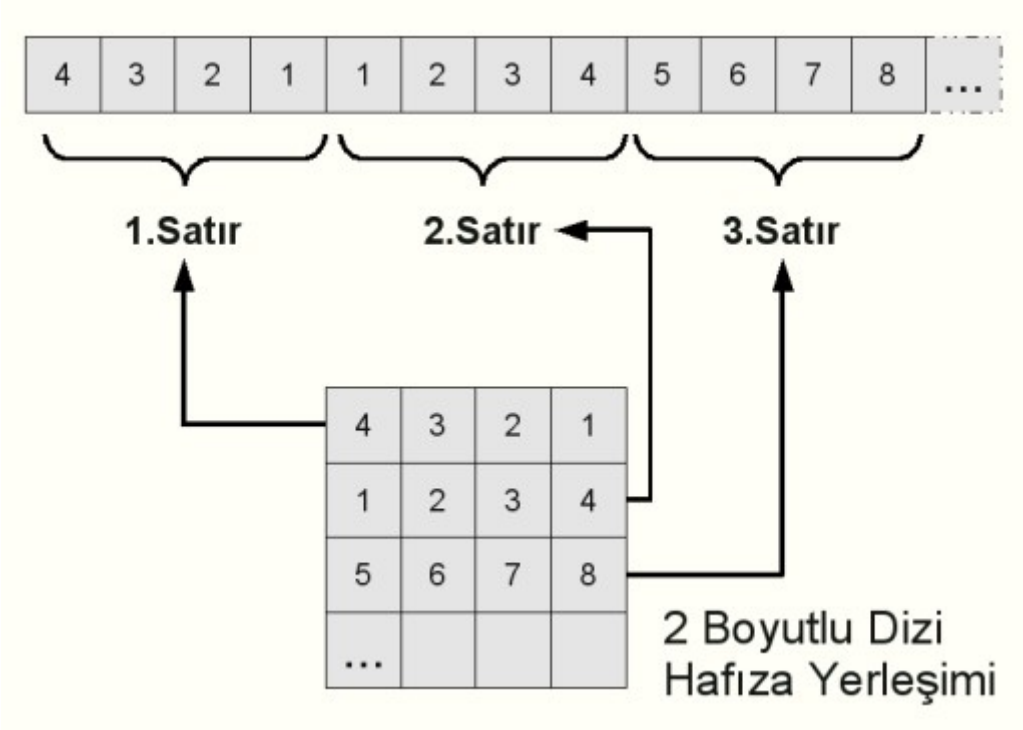
```
#include<stdio.h>
/* Parametre tanimlamasi yaparken, iki boyutlu dizinin
   satir boyutunu girmemize gerek yoktur. Ancak sutun
   boyutunu girmek gerekir.
*/
void matris_yazdir( int [ ][ 4 ], int );
int main( void )
{
        // Ornek olmasi acisindan matrise keyfi
        // degerler atiyoruz. Matrisimiz 3 satir
        // ve 4 sutundan ( 3 x 4 ) olusuyor.
        int matris[ 3 ][ 4 ] = {
                        {10, 15, 20, 25},
                        {30, 35, 40, 45},
                        {50, 55, 60, 65} };

        // Matris elemanlarini yazdiran fonksiyonu
        // cagriyoruz.
        matris_yazdir( matris, 3 );

        return 0;
}
void matris_yazdir( int dizi[ ][ 4 ], int satir_sayisi )
{
        int i, j;
        for( i = 0; i < satir_sayisi; i++ ) {
                for( j = 0; j < 4; j++ ) {
                        printf( "%d ", dizi[ i ][ j ] );
                }
                printf( "\n" );
        }
}
```

Kod içersinde bulunan yorumlar, iki boyutlu dizilerin fonksiyonlara nasıl aktarıldığını göstermeye yetecektir. Yine de bir kez daha tekrar edelim... Fonksiyonu tanımlarken, çok boyutlu dizinin ilk boyutunu yazmak zorunda değilsiniz. Bizim örneğimizde *int dizi[ ][4]* şeklinde belirtmemiz bundan kaynaklanıyor. Şayet 7 x 6 x 4 boyutlarında dizilerin kullanılacağı bir fonksiyon yazsaydık tanımlamamızı *int dizi[ ][6][4]* olarak değiştirmemiz gerekirdi. Kısacası fonksiyonu tanımlarken dizi boyutlarına dair ilk değeri yazmamakta serbestsiniz; ancak diğer boyutların yazılması zorunlu! Bunun yararını merak ederseniz, sütun sayısı 4 olan her türlü matrisi bu fonksiyona gönderebileceğinizi hatırlatmak isterim. Yani fonksiyon her boyutta matrisi alabilir, tabii sütun

sayısı 4 olduğu sürece...

2 Boyutlu Dizilerin Hafıza Yerleşimi

Dizilerin çok boyutlu olması sizi yanıltmasın, bilgisayar hafızası tek boyutludur. İster tek boyutlu bir dizi, ister iki boyut ya da isterseniz 10 boyutlu bir dizi içersinde bulunan elemanlar, birbiri peşi sıra gelen bellek hücrelerinde tutulur. İki boyutlu bir dizide bulunan elemanların, hafizada nasıl yerleştirildiğini aşağıdaki grafikte görebilirsiniz.



Görüldüğü gibi elemanların hepsi sırayla yerleştirilmiştir. Bir satırın bittiği noktada ikinci satırın elemanları devreye girer. Kapsamlı bir örnekle hafıza yerleşimini ele alalım:

```
#include<stdio.h>
void satir_goster( int satir[ ] );
int main( void )
{
        int tablo[5][4] = {
                        {4, 3, 2, 1},
                        {1, 2, 3, 4},
                        {5, 6, 7, 8},
                        {2, 5, 7, 9},
                        {0, 5, 9, 0} };
        int i, j;

        // Dizinin baslangic adresini yazdiriyoruz
        printf( "2 boyutlu tablo %p adresinden başlar\n\n", tablo );

        // Tablo icersinde bulunan dizi elemanlarinin adreslerini
        // ve degerlerini yazdiriyoruz.
        printf( "Tablo elemanları ve hafıza adresleri:\n");
        for( i = 0; i < 5; i++ ) {
                for( j = 0; j < 4; j++ ) {
                        printf( "%d (%p) ", tablo[i][j], &tablo[i][j] );
                }
                printf( "\n" );
        }

        // Cok boyutlu diziler birden fazla dizinin toplami olarak
```

```
        // dusunulebilir ve her satir tek boyutlu bir dizi olarak
        // ele alinabilir. Once her satirin baslangic adresini
        // gosteriyoruz. Sonra satirlari tek boyutlu dizi seklinde
        // satir_goster( ) fonksiyonuna gonderiyoruz.
        printf( "\nTablo satırlarının başlangıç adresleri: \n");
        for( i = 0; i < 5; i++ )
                printf( "tablo[%d]'nin başlangıç adresi %p\n", i, tablo[i] );

        printf( "\nsatir_goster( ) fonksiyonuyla, "
                "tablo elemanları ve hafıza adresleri:\n");
        for( i = 0; i < 5; i++ )
                satir_goster( tablo[i] );
}
// Kendisine gonderilen tek boyutlu bir dizinin
// elemanlarini yazdirir.
void satir_goster( int satir[ ] )
{
        int i;
        for( i = 0; i < 4; i++ ) {
                printf( "%d (%p) ", satir[i], &satir[i] );
        }
        printf( "\n" );
}
```

Örnekle ilgili en çok dikkat edilmesi gereken nokta, çok boyutlu dizilerin esasında, tek boyutlu dizilerden oluşmuş bir bütün olduğudur. Tablo isimli 2 boyutlu dizimiz 5 adet satırdan oluşur ve bu satırların her biri kendi başına bir dizidir. Eğer tablo[2] derseniz bu üçüncü satırı temsil eden bir diziyi ifade eder. satir_goster( ) fonksiyonunu ele alalım. Esasında fonksiyon içersinde satır diye bir kavramın olmadığını söyleyebiliriz. Bütün olan biten fonksiyona tek boyutlu bir dizi gönderilmesidir ve fonksiyon bu dizinin elemanlarını yazar.

Dizi elemanlarının hafızadaki ardışık yerleşimi bize başka imkanlar da sunar. İki boyutlu bir diziyi bir hamlede, tek boyutlu bir diziye dönüştürmek bunlardan biridir.

```
#include<stdio.h>
int main( void )
{
        int i;
        int tablo[5][4] = {
                        {4, 3, 2, 1},
                        {1, 2, 3, 4},
                        {5, 6, 7, 8},
                        {2, 5, 7, 9},
                        {0, 5, 9, 0} };

        // Cok boyutlu dizinin baslangic
        // adresini bir pointer'a atiyoruz.
        int *p = tablo[0];

        // p isimli pointer'i tek boyutlu
        // bir dizi gibi kullanabiliriz.
        // Ayni zamanda p uzerinde yapacagimiz
        // degisikler, tablo'yu da etkiler.
        for( i = 0; i < 5*4; i++ )
                printf( "%d\n", p[i] );

        return 0;
}
```

Daha önce sıralama konusunu işlemiştik. Ancak bunu iki boyutlu dizilerde nasıl yapacağımızı henüz görmedik. Aslında görmemize de gerek yok! İki boyutlu bir diziyi yukardaki gibi tek boyuta indirin ve sonrasında sıralayın. Çok boyutlu dizileri, tek boyuta indirmemizin ufak bir faydası...

Pointer Dizileri

Çok boyutlu dizilerin tek boyutlu dizilerin bir bileşimi olduğundan bahsetmiştik. Şimdi anlatacağımız konuda çok farklı değil. Dizilerin, adresi göstermeye yarayan Pointer'lardan pek farklı olmadığını zaten biliyorsunuz. Şimdi de pointer dizilerini göreceğiz. Yani adres gösteren işaretçi saklayan dizileri...

```
#include<stdio.h>
int main( void )
{
        int i, j;

        // Dizi isimleri keyfi secilmistir.
        // alfa, beta, gama gibi baska isimler de
        // verebilirdik.
        int Kanada[8];
        int ABD[8];
        int Meksika[8];
        int Rusya[8];
        int Japonya[8];

        // Bir pointer dizisi tanimliyoruz.
        int *tablo[5];
        // Yukarda tanimlanan dizilerin adreslerini
        // tablo'ya aktiriyoruz.
        tablo[0] = Kanada;
        tablo[1] = ABD;
        tablo[2] = Meksika;
        tablo[3] = Rusya;
        tablo[4] = Japonya;

        // Tablo elemanlarinin adreslerini gosteriyor
        // gibi gozukse de, gosterilen adresler Kanada,
        // ABD, Meksika, Rusya ve Japonya dizilerinin
        // eleman adresleridir.
        for( i = 0; i < 5; i++ ) {
                for( j = 0 ; j < 8; j++ )
                        printf( "%p\n", &tablo[i][j] );
        }
        return 0;
}
```

Ülke isimlerini verdiğimiz 5 adet dizi tanımladık. Bu dizileri daha sonra tabloya sırayla atadık. Artık her diziyle tek tek uğraşmak yerine tek bir diziden bütün ülkelere ulaşmak mümkün hâle gelmiştir. İki boyutlu *tablo* isimli matrise atamasını yaptığımız şey değer veya bir eleman değildir; dizilerin başlangıç adresleridir. Bu yüzden *tablo* dizisi içersinde yapacağımız herhangi bir değişiklik orijinal diziyi de (örneğin Meksika) değiştirir.

Atama işlemini aşağıdaki gibi tek seferde de yapabilirdik:

```
int *tablo[ ] = { Kanada, ABD, Meksika, Rusya, Japonya };
```

Şimdi de bir pointer dizisini fonksiyonlara nasıl argüman olarak göndereceğimize bakalım.

```
#include<stdio.h>
void adresleri_goster( int *[ ] );
int main( void )
{
        int Kanada[8];
        int ABD[8];
        int Meksika[8];
        int Rusya[8];
        int Japonya[8];
```

```
        int *tablo[ ] = { Kanada, ABD, Meksika, Rusya, Japonya };

        // Adresleri gostermesi icin adresleri_goster( )
        // fonksiyonunu cagriyoruz.
        adresleri_goster( tablo );

        return 0;
}
void adresleri_goster( int *dizi[ ] )
{
        int i, j;
        for( i = 0; i < 5; i++ ) {
                for( j = 0 ; j < 8; j++ )
                        printf( "%p\n", &dizi[ i ][ j ] );
        }
}
```

Dinamik Bellek Yönetimi

Dizileri etkin bir biçimde kullanmayı öğrendiğinizi ya da öğreneceğinizi umuyorum. Ancak dizilerle ilgili işlememiz gereken son bir konu var: Dinamik Bellek Yönetimi...

Şimdiye kadar yazdığımız programlarda kaç eleman olacağı önceden belliydi. Yani sınıf listesiyle ilgili bir program yazacaksak, sınıfın kaç kişi olduğunu biliyormuşuz gibi davranıyorduk. Programın en başında kaç elemanlık alana ihtiyacımız varsa, o kadar yer ayırıyorduk. Ama bu gerçek dünyada karşımıza çıkacak problemler için yeterli bir yaklaşım değildir. Örneğin bir sınıfta 100 öğrenci varken, diğer bir sınıfta 50 öğrenci olabilir ve siz her ortamda çalışsın diye 200 kişilik bir üst sınır koyamazsınız. Bu, hem hafızanın verimsiz kullanılmasına yol açar; hem de karma eğitimlerin yapıldığı bazı fakültelerde sayı yetmeyebilir. Statik bir şekilde dizi tanımlayarak bu sorunların üstesinden gelemezsiniz. Çözüm dinamik bellek yönetimindedir.

Dinamik bellek yönetiminde, dizilerin boyutları önceden belirlenmez. Program akışında dizi boyutunu ayarlarız ve gereken bellek miktarı, program çalışırken tahsis edilir. Dinamik bellek tahsisi için *calloc( )* ve *malloc( )* olmak üzere iki önemli fonksiyonumuz vardır. Bellekte yer ayrılmasını bu fonksiyonlarla sağlarız. Her iki fonksiyon da *stdlib* kütüphanesinde bulunur. Bu yüzden fonksiyonlardan herhangi birini kullanacağınız zaman, programın başına *#include<stdlib.h>* yazılması gerekir.

calloc( ) fonksiyonu aşağıdaki gibi kullanılır:

```
isaretci_adi = calloc( eleman_sayisi, her_elemanin_boyutu );
```

calloc( ) fonksiyonu eleman sayısını, eleman boyutuyla çarparak hafızada gereken bellek alanını ayırır. Dinamik oluşturduğunuz dizi içersindeki her elemana, otomatik olarak ilk değer 0 atanır.

malloc( ) fonksiyonu, *calloc( )* gibi dinamik bellek ayrımı için kullanılır. *calloc( )* fonksiyonundan farklı olarak ilk değer ataması yapmaz. Kullanımıysa aşağıdaki gibidir:

```
isaretci_adi = malloc( eleman_sayisi * her_elemanin_boyutu );
```

Bu kadar konuşmadan sonra işi pratiğe dökelim ve dinamik bellekle ilgili ilk programımızı yazalım:

```
#include<stdio.h>
#include<stdlib.h>
int main( void )
{
        // Dinamik bir dizi yaratmak icin
        // pointer kullaniriz.
        int *dizi;
```

```
        // Dizimizin kac elemanli olacagini
        // eleman_sayisi isimli degiskende
        // tutuyoruz.
        int eleman_sayisi;
        int i;

        // Kullanicidan eleman sayisini girmesini
        // istiyoruz.
        printf( "Eleman sayısını giriniz> ");
        scanf( "%d", &eleman_sayisi );

        // calloc( ) fonksiyonuyla dinamik olarak
        // dizimizi istedigimiz boyutta yaratiyoruz.
        dizi = calloc( eleman_sayisi, sizeof( int ) );

        // Ornek olmasi acisindan dizinin elemanlarini
        // ekrana yazdiriliyor. Dizilerde yapabildiginiz
        // her seyi hicbir fark olmaksizin yapabilirsiniz.
        for( i = 0; i < eleman_sayisi; i++ )
                printf( "%d\n", dizi[i] );

        // Dinamik olan diziyi kullandiktan ve isinizi
        // tamamladiktan sonra free fonksiyonunu kullanip
        // hafizadan temizlemelisiniz.
        free( dizi );

        return 0;
}
```

Yazdığınız programların bir süre sonra bilgisayar belleğini korkunç bir şekilde işgal etmesini istemiyorsanız, free( ) fonksiyonunu kullanmanız gerekmektedir. Gelişmiş programlama dillerinde (örneğin, Java, C#, vb...) kullanılmayan nesnelerin temizlenmesi otomatik olarak çöp toplayıcılarla (Garbage Collector) yapılmaktadır. Ne yazık ki C programlama dili için bir çöp toplayıcı yoktur ve iyi programcıyla, kötü programcı burada kendisini belli eder.

Programınızı bir kereliğine çalıştırıyorsanız ya da yazdığınız program çok ufaksa, boş yere tüketilen bellek miktarını farketmeyebilirsiniz. Ancak büyük boyutta ve kapsamlı bir program söz konusuysa, efektif bellek yönetiminin ne kadar önemli olduğunu daha iyi anlarsınız. Gereksiz tüketilen bellekten kaçınmak gerekmektedir. Bunun için fazla bir şey yapmanız gerekmez; *calloc( )* fonksiyonuyla tahsis ettiğiniz alanı, işiniz bittikten sonra *free( )* fonksiyonuyla boşaltmanız yeterlidir. Konu önemli olduğu için tekrar ediyorum; artık kullanmadığınız bir dinamik dizi söz konusuysa onu *free( )* fonksiyonuyla kaldırılabilir hâle getirmelisiniz!

Az evvel *calloc( )* ile yazdığımız programın aynısını şimdi de *malloc( )* fonksiyonunu kullanarak yazalım:

```
#include<stdio.h>
#include<stdlib.h>
int main( void )
{
        // Dinamik bir dizi yaratmak icin
        // pointer kullaniriz.
        int *dizi;
        // Dizimizin kac elemanli olacagini
        // eleman_sayisi isimli degiskende
        // tutuyoruz.
        int eleman_sayisi;
        int i;

        printf( "Eleman sayısını giriniz> ");
        scanf( "%d", &eleman_sayisi );
```

```
        // malloc( ) fonksiyonuyla dinamik olarak
        // dizimizi istedigimiz boyutta yaratiyoruz.
        dizi = malloc( eleman_sayisi * sizeof( int ) );

        for( i = 0; i < eleman_sayisi; i++ )
                printf( "%d\n", dizi[i] );

        // Dinamik olan diziyi kullandiktan ve isinizi
        // tamamladiktan sonra free fonksiyonunu kullanip
        // hafizadan temizlemelisiniz.
        free( dizi );

        return 0;
}
```

Hafıza alanı ayırırken bazen bir problem çıkabilir. Örneğin bellekte yeterli alan olmayabilir ya da benzeri bir sıkıntı olmuştur. Bu tarz problemlerin sık olacağını düşünmeyin. Ancak hafızanın gerçekten ayrılıp ayrılmadığını kontrol edip, işinizi garantiye almak isterseniz, aşağıdaki yöntemi kullanabilirsiniz:

```
        dizi = calloc( eleman_sayisi, sizeof( int ) );
        // Eger hafiza dolmussa dizi pointer'i NULL'a
        // esit olacak ve asagidaki hata mesaji cikacaktir.
        if( dizi == NULL )
                printf( "Yetersiz bellek!\n" );
```

Dinamik hafıza kullanarak dizi yaratmayı gördük. Ancak bu diziler tek boyutlu dizilerdi. Daha önce pointer işaret eden pointer'ları görmüştük. Şimdi onları kullanarak dinamik çok boyutlu dizi oluşturacağız:

```
#include<stdio.h>
#include<stdlib.h>
int main( void )
{
        int **matris;
        int satir_sayisi, sutun_sayisi;
        int i, j;
        printf( "Satır sayısı giriniz> " );
        scanf( "%d", &satir_sayisi );
        printf( "Sütun sayısı giriniz> " );
        scanf( "%d", &sutun_sayisi );

        // Once satir sayisina gore hafizada yer ayiriyoruz.
        // Eger gerekli miktar yoksa, uyari veriliyor.
        matris = (int **)malloc( satir_sayisi * sizeof(int) );
        if( matris == NULL )
                printf( "Yetersiz bellek!" );

        // Daha sonra her satirda, sutun sayisi kadar hucrenin
        // ayrilmasini sagliyoruz.
        for( i = 0; i < satir_sayisi; i++ ) {
                matris[i] = malloc( sutun_sayisi * sizeof(int) );
                if( matris[i] == NULL )
                        printf( "Yetersiz bellek!" );
        }

        // Ornek olmasi acisindan matris degerleri
        // gosteriliyor. Dizilerde yaptiginiz butun
        // islemleri burada da yapabilirsiniz.
        for( i = 0; i < satir_sayisi; i++ ) {
                for( j = 0; j < sutun_sayisi; j++ )
                        printf( "%d ", matris[i][j] );
```

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

        // Bu noktada matris ile isimiz bittiginden
        // hafizayi bosaltmamiz gerekiyor. Oncelikle
        // satirlari bosaltiyoruz.
        for( i = 0; i < satir_sayisi; i++ ) {
                free( matris[i] );
        }
        // Satirlar bosaldiktan sonra, matrisin
        // bos oldugunu isaretliyoruz.
        free( matris );

        return 0;
}
```

Yukardaki örnek karmaşık gelebilir; tek seferde çözemeyebilirsiniz. Ancak bir iki kez üzerinden geçerseniz, temel yapının aklınıza yatacağını düşünüyorum. Kodun koyu yazılmış yerlerini öğrendiğiniz takdirde, sorun kalmayacaktır.

Örnek Sorular

Soru 1: Kendisine gönderilen iki diziyi birleştirip geriye tek bir dizi döndüren fonksiyonu yazınız.

Cevap 1:

```
#include<stdio.h>
#include<stdlib.h>
/* Kendisine verilen iki diziyi birlestirip
   sonuc dizisini geriye dondurur */
int *dizileri_birlestir( int [], int,
                         int [], int );
int main( void )
{
        int i;
        // liste_1, 5 elemanli bir dizidir.
        int liste_1[5] = { 6, 7, 8, 9, 10 };
        // liste_2, 7 elemanli bir dizidir.
        int liste_2[7] = {13, 7, 12, 9, 7, 1, 14 };
        // sonuclarin toplanacagi toplam_sonuc dizisi
        // sonucun dondurulmesi icin pointer tanimliyoruz
        int *ptr;

        // fonksiyonu calistiriyoruz.
        ptr = dizileri_birlestir( liste_1, 5, liste_2, 7 );

        // ptr isimli pointer'i bir dizi olarak dusunebiliriz
        for( i = 0; i < 12; i++ )
                printf("%d ", ptr[i] );
        printf("\n");

        return 0;
}
int *dizileri_birlestir( int dizi_1[], int boyut_1,
                         int dizi_2[], int boyut_2 )
{
        int *sonuc = calloc( boyut_1+boyut_2, sizeof(int) );
        int i, k;
        // Birinci dizinin degerleri ataniyor.
        for( i = 0; i < boyut_1; i++ )
                sonuc[i] = dizi_1[i];

        // Ikinci dizinin degerleri ataniyor.
```

```
        for( k = 0; k < boyut_2; i++, k++ ) {
                sonuc[i] = dizi_2[k];
        }

        // Geriye sonuc dizisi gonderiliyor.
        return sonuc;
}
```

Soru 2: Sol aşağıda bulunan 4x4 boyutundaki matrisi saat yönünde 90^o döndürecek fonksiyonu yazınız. (Sol matris döndürüldüğü zaman sağ matrise eşit olmalıdır.)

```
 12   34   22   98         38   90   88   12
 88   54   67   11         39   91   54   34
 90   91   92   93   >>>   40   92   67   22
 38   39   40   41         41   93   11   98
```

Cevap 2:

```
#include<stdio.h>
void elemanlari_goster( int [][4] );
void saat_yonunde_cevir( int [][4] );
int main( void )
{
        int matris[4][4] = {
                {12, 34, 22, 98},
                {88, 54, 67, 11},
                {90, 91, 92, 93},
                {38, 39, 40, 41} };
        elemanlari_goster( matris );
        printf("\n");
        saat_yonunde_cevir( matris );
}
void elemanlari_goster( int dizi[][4] )
{
        int i, j;
        for( i = 0; i < 4; i++ ) {
                for( j = 0; j < 4; j++ )
                        printf( "%d ", dizi[i][j] );
                printf( "\n" );
        }
}
void saat_yonunde_cevir( int dizi[][4] )
{
        int i, j;
        for( i = 0; i < 4; i++ ) {
                for( j = 0; j < 4 ; j++ )
                        printf( "%d ", dizi[3-j][i] );
                printf( "\n" );
        }
}
```

Katarlar (String)

Dizileri ve çok boyutlu dizileri gördük. Katar dediğimiz şey de aslında bir dizidir. Değişken tipi *char* yani karakter olan diziler, '*katar*' ya da İngilizce adıyla '*string*' olarak isimlendirilirler.

Katarları, şimdiye kadar gördüğümüz dizilerden ayıran, onları farklı kılan özellikleri yoktur. Örneğin bir tam sayı (*int*) dizisinde, tam sayıları saklarken; bir karakter dizisinde -yani katarda- karakterleri (*char*) saklarız. Bunun dışında bir fark bulunmaz. Ancak sık kullanılmalarına paralel olarak, katarlara ayrı bir önem vermek gerekir. Yaptığınız işlemler bilimsel ve hesaplama ağırlıklı

değilse, hangi dili kullanırsanız kullanın, en çok içli dışlı olacağınız dizi tipi, karakter dizileridir. İsimler, adresler, kullanıcı adları, telefonlar vs... sözle ifade edilebilecek her şey için karakter dizilerini kullanırız. Katarlar işte bu yüzden önemlidir!

Karakter dizilerine İngilizce'de *String* dendiğini belirtmiştik. String; ip, bağ, kordon gibi anlamlar taşıyor. İlk olarak *katar* adını kim münasip gördü bilmiyorum. Muhtemelen bellek hücrelerine peşi sıra dizilen karakterlerin, vagonlara benzetilmesiyle, String değişken tipi Türkçe'ye katar olarak çevrildi. (*Arapça kökenli Türkçe bir kelime olan katar, 'tren' anlamına gelmektedir.*) Daha uygun bir isim verilebilirdi ya da sadece '*karakter dizisi*' de diyebilirdik. Fakat madem genel kabul görmüş bir terim var; yazımız içersinde biz de buna uyacağız. String, katar ya da karakter dizisi hiç farketmez; hepsi aynı kapıya çıkıyor: Değişken tipi karakter olan dizi...

Katarlarda printf( ) ve scanf( ) Kullanımı

Katarlarla, daha önce gördüğümüz diziler arasında bir farkın olmadığını söylemtiştik. Bu sözümüz, teorik olarak doğru olsa da, pratikte ufak tefek farkları kapsam dışı bırakıyor. Hatırlayacaksınız, dizilerde elemanlara değer atama ya da onlardan değer okuma adım adım yapılan bir işlemdi. Genellikle bir döngü içersinde, her dizi elemanı için scanf( ) veya printf( ) fonksiyonunu çağırmamız gerekiyordu. Katarlar için böyle bir mecburiyet bulunmuyor. Tek bir kelimeyi, tek bir scanf( ) fonksiyonuyla okutabilir ve elemanlara otomatik değer atayabilirsiniz. Yani "Merhaba" şeklinde bir girdi-input gelirse, 3.dizi elemanı 'r' olurken; 6.dizi elemanı 'b' olur. Önceki dizilerde gördüğümüzün aksine, eleman atamaları kendiliğinden gerçekleşir. Aşağıdaki örneği inceleyelim:

```
#include<stdio.h>
int main( void )
{
        char isim[30];
        printf( "İsim giriniz> ");
        scanf( "%s", isim );
        printf( "Girdiğiniz isim: %s\n", isim );
        return 0;
}
```

Örneğimizde 30 karakterlik bir karakter dizisi tanımlayarak işe başladık. Bunun anlamı girdileri saklayacağımız '*isim*' katarının 30 karakter boyutunda olacağıdır. Ancak bu katara en fazla 29 karakterlik bir kelime atanabilir. Çünkü katarlarda, kelime bitiminden sonra en az bir hücre boş bırakılmalıdır. Bu hücre '*Boş Karakter*' (*NULL Character*) tutmak içindir. Boş karakter "**\0**" şeklinde ifade edilir. C programlama dilinde, kelimelerin bittiğini boş karakterlerle anlarız. Herhangi bir katarı boş karakterle sonlandırmaya, '*null-terminated*' denmektedir.

Bu arada katarlara değer atarken ya da katarlardan değer okurken, sadece katar adını yazmamızın yettiğini farketmişsinizdir. Yani scanf( ) fonksiyonu içersine & işareti koymamız gerekmiyor. Çünkü scanf( ), katarın ilk adresinden başlayarak aşağıya doğru harfleri tek tek ataması gerektiğini biliyor. (Aslında biliyor demek yerine, fonksiyonun o şekilde yazıldığını söylememiz daha doğru olur.)

Katarların, esasında bir dizi olduğundan bahsetmiştik. Şimdi bunun uygulamasını yapalım. Katara değer atamak için yine aynı kodu kullanırken; katardan değer okumak için kodumuzu biraz değiştirelim:

```
#include<stdio.h>
int main( void )
{
        char isim[30];
        int i;
        printf( "İsim giriniz> ");
        scanf( "%s", isim );
```

```
        printf( "Girdiğiniz isim: ");
        for( i = 0; isim[i]!='\0'; i++ )
                printf( "%c", isim[i] );
        printf("\n");

        return 0;
}
```

Daha önce tek bir printf( ) fonksiyonuyla bütün katarı yazdırabilirken, bu sefer katar elemanlarını tek tek, karakter karakter yazdırmayı tercih ettik. Çıkan sonuç aynı olacaktır fakat gidiş yolu biraz farklılaştı. Özellikle *for* döngüsü içersinde bulunan " *isim[i]!='\0'* " koşuluna dikkat etmek gerekiyor. İsteseydik, " *i < 30* " yazar ve katarın bütün hücrelerini birer birer yazdırabilirdik. Fakat bu mantıklı değil! 30 karakterlik bir dizi olsa bile, kullanıcı 10 harften oluşan bir isim girebilir. Dolayısıyla kalan 20 karakteri yazdırmaya gerek yoktur. Kelimenin nerede sonlandığını belirlemek için "*isim[i]!='\0'*" koşulunu kullanıyoruz. Bunun anlamı; *isim* katarının elemanları, "\0" yani boş karakterere (NULL Character) eşit olmadığı sürece yazdırmaya devam edilmesidir. Ne zaman ki kelime biter, sıradaki elemanın değeri "**\0**" olur; işte o vakit döngüyü sonlandırmamız gerektiğini biliriz.

Yukardaki örneğimize birden çok kelime girdiyseniz, sadece ilk kelimenin alındığını farketmişsinizidir. Yani "*Bugün hava çok güzel.*" şeklinde bir cümle girdiğiniz zaman, katara sadece "*Bugün*" kelimesi atanır. Eğer aynı anda birden fazla kelime almak istiyorsanız, ayrı ayrı belirtilmesi gerekir.

```
#include<stdio.h>
int main( void )
{
        char isim[25], soyad[30];
        printf( "Ad ve soyad giriniz> ");
        scanf( "%s%s", isim, soyad );
        printf( "Sayın %s %s, hoş geldiniz!\n", isim, soyad );
        return 0;
}
```

gets( ) ve puts( ) Fonksiyonları

Gördüğünüz gibi aynı anda iki farklı kelime alıp, ikisini birden yazdırdık. Fakat scanf( ) fonksiyonu "*Bugün hava çok güzel.*" cümlesini tek bir katara alıp, atamak için hâlen yetersizdir. Çünkü boşluk gördüğü noktada, veriyi almayı keser ve sadece "*Bugün*" kelimesinin atamasını yapar. Boşluk içeren bu tarz cümleler için puts( ) ve gets( ) fonksiyonları kullanılmaktadır. Aşağıdaki örnek program, 40 harfi geçmeyecek her cümleyi kabul edecektir:

```
#include<stdio.h>
int main( void )
{
        char cumle[40];
        printf( "Cümle giriniz> ");
        gets( cumle );
        printf( "Girdiğiniz cümle:\n" );
        puts( cumle );
        return 0;
}
```

gets( ) isminden anlayacağınız (*get string*) gibi katara değer atamak için kullanılır. puts( ) (*put string*) ise, bir katarın içeriğini ekrana yazdırmaya yarar. gets( ) atayacağı değerin ayrımını yapabilmek için '**\n**' aramaktadır. Yani klavyeden Enter'a basılana kadar girilen her şeyi, tek bir katara atayacaktır. puts( ) fonksiyonuysa, printf( ) ile benzer çalışır. Boş karakter (NULL Character) yani '**\0**' ulaşana kadar katarı yazdırır; printf( ) fonksiyonundan farklı olarak sonuna '**\n**'

koyarak bir alt satıra geçer. Oldukça açık ve basit kullanımlara sahip olduklarından, kendiniz de başka örnekler deneyebilirsiniz.

Katarlara İlk Değer Atama

Bir katar tanımı yaptığınız anda, katarın bütün elemanları otomatik olarak '\0' ile doldurulur. Yani katarın bütün elemanlarına boş karakter (NULL Character) atanır. Dilerseniz, katarı yaratırken içine farklı değerler atayabilirsiniz. Katarlarda ilk değer ataması iki şekilde yapılır.

Birinci yöntemle değer ataması yapacaksanız, istediğiniz kelimeyi bir bütün olarak yazarsınız:

```
#include<stdio.h>
int main( void )
{
        // Her iki katarada ilk deger
        // ataması yapiliyor. Ancak
        // isim katarinda, boyut
        // belirtilmezken, soyad katarinda
        // boyutu ayrica belirtiyoruz.
        char isim[] = "CAGATAY";
        char soyad[5] = "CEBI";
        printf( "%s %s\n", isim, soyad );

        return 0;
}
```

İkinci yöntemdeyse, kelime bütün olarak yazılmaz. Bunun yerine harf harf yazılır ve sonlandırmak için en sonuna boş karakter (NULL) eklenir:

```
#include<stdio.h>
int main( void )
{
        char isim[] = { 'C', 'A', 'G', 'A',
                        'T', 'A', 'Y', '\0' };
        char soyad[5] = { 'C', 'E', 'B', 'I', '\0' };
        printf( "%s %s\n", isim, soyad );
        return 0;
}
```

Ben ilk değer ataması yapacağım durumlarda, ilk yolu tercih ediyorum. İkinci yöntem, daha uzun ve zahmeti...

Biçimlendirilmiş (Formatlı) Gösterim

Daha önce float tipindeki bir sayının, noktadan sonra iki basamağını göstermek türünden şeyler yapmıştık. Örneğin printf( ) fonksiyonu içersinde, sayıyı %.2f şeklinde ifade ederseniz, sayının virgülden sonra sadece iki basamağı gösterilir. Yada %5d yazarak tam sayıları gösterdiğiniz bir durumda, sayı tek bir rakamdan dahi oluşsa, onun için 5 rakamlık gösterim yeri ayrılır. Aynı şekilde biçimlendirilmiş (formatlı) gösterim, katarlarda da yapılmaktadır.

Katarları biçimlendirilmiş şekilde göstermeyi, örnek üzerinden anlatmak daha uygun olacaktır:

```
#include<stdio.h>
int main( void )
{
        char cumle[20] = "Denemeler";

        // Cumleyi aynen yazar:
        printf( "%s\n", cumle );

        // 20 karakterlik alan ayirir
```

```
        // ve en saga dayali sekilde yazar.
        printf( "%20s\n", cumle );

        // 20 karakterlik alan ayirir
        // ve en saga dayali sekilde,
        // katarin ilk bes kelimesini
        // yazar
        printf( "%20.5s\n", cumle );

        // 5 karakterlik alan ayirir
        // ve en saga dayali sekilde yazar.
        // Eger girilen kelime 5 karakterden
        // buyukse, kelimenin hepsi yazilir.
        printf( "%5s\n", cumle );

        // 20 karakterlik alan ayirir
        // ve sola dayali sekilde yazar.
        // Sola dayali yazilmasi icin
        // yuzde isaretinden sonra, -
        // (eksi) isareti konulur.
        printf( "%-20s\n", cumle );

        return 0;
}
```

Örneğimizde bulunan formatlama biçimlerini gözden geçirirsek:

- %20s, ekranda 20 karakter alan ayrılacağı anlamına gelir. Katar, en sağa dayanır ve "*Denemeler*" yazılır.
- %.5s olursa 5 karakterlik boşluk ayrılır. Yüzde işaretinden sonra nokta olduğu için katarın sadece ilk beş harfi yazdırılır. Yani sonuç "*Denem*" olacaktır. %20.5s yazıldığında, 20 karakterlik boşluk ayrılması istenmiş ancak katarın sadece ilk 5 harfi bu boşluklara yazılmıştır.
- %5s kullanırsanız, yine 5 karakterlik boşluk ayrılacaktır. Ancak yüzdeden sonra nokta olmadığı için, katarın hepsi yazılır. Belirtilen boyutu aşan durumlarda, eğer noktayla sınır konmamışsa, katar tamamen gösterilir. Dolayısıyla çıktı, "Denemeler" şeklinde olacaktır.
- Anlattıklarımızın hepsi, sağa dayalı şekilde çıktı üretir. Eğer sola dayalı bir çıktı isterseniz, yüzde işaretinden sonra '-' (eksi) işareti koymanız gerekir. Örneğin %-20.5s şeklinde bir format belirlerseniz, 20 karakterlik boşluk ayarlandıktan sonra, sola dayalı olarak katarın ilk 5 harfi yazdırılacaktır. İmleç (cursor), sağ yönde 20 karakter sonrasına düşecektir.

Standart Katar Fonksiyonları

Katarlarla daha kolay çalışabilmek için, bazı hazır kütüphane fonksiyonlarından bahsedeceğiz. Bu fonkisyonlar, string kütüphanesinde bulunuyor. Bu yüzden, programınızın başına, #include<string.h> eklemeniz gerekiyor.

* strlen( ) fonksiyonuyla katar boyutu bulma

Dizi boyutuyla, katar uzunluğunun farklı şeyler olduğundan bahsetmiştik. Dizi boyutu, 40 karakter olacak şekilde ayarlanmışken, dizi içinde sadece 7 karakterlik "*Merhaba*" kelimesi tutulabilir. Bu durumda, dizi boyutu 40 olmasına rağmen, katar boyutu yalnızca 7'dir. Katarların boyutunu saptamak için, boş karakter (NULL Character) işaretinin yani "*\0*" simgesinin konumuna bakılır. Her seferinde arama yapmanıza gerek kalmasın diye *strlen( )* fonksiyonu geliştirilmiştir. *strlen( )* kendisine argüman olarak gönderilen bir katarın boyutunu geri döndürür. Aşağıdaki gibi kullanılmaktadır:

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

```
#include<string.h>
int main( void )
{
        printf( "Katar Uzunluğu: %d\n", strlen("Merhaba") );
        return 0;
}
```

* strcpy( ) ve strncpy( ) ile katar kopyalama

Bir katarı, bir başka katara kopyalamak için *strcpy( )* fonksiyonunu kullanırız. Katarlar aynı boyutta olmak zorunda değildir. Ancak kopya olacak katar, kendisine gelecek kelimeyi alacak boyuta sahip olmalıdır. Fonksiyon prototipi aşağıdaki gibidir, geriye pointer döner.

```
char *strcpy( char[ ], char[ ] );
```

strcpy( ) fonksiyonunu bir örnekle görelim:

```
#include<stdio.h>
#include<string.h>
int main( void )
{
        char kaynak[40]="Merhaba Dünya";
        char kopya[30] = "";
        strcpy( kopya, kaynak );
        printf( "%s\n", kopya );

        return 0;
}
```

strncpy( ) fonksiyonu, yine kopyalamak içindir. Fakat emsalinden farklı olarak, kaç karakterin kopyalanacağı belirtilir. Protopi aşağıda verilmiştir:

```
char *strncpy( char[ ], char[ ], int );
```

Yukardaki örneği strncpy( ) fonksiyonuyla tekrar edelim:

```
#include<stdio.h>
#include<string.h>
int main( void )
{
        char kaynak[40]="Merhaba Dünya";
        char kopya[30] = "";
        strncpy( kopya, kaynak, 9 );
        printf( "%s\n", kopya );

        return 0;
}
```

Yukardaki programı çalıştırırsanız, kopya isimli katara sadece 9 karakterin aktarıldığını ve ekrana yazdırılan yazının "*Merhaba D*" olduğunu görebilirsiniz.

* strcmp( ) ve strncmp( ) ile katar karşılaştırma

strcmp( ) fonksiyonu, kendisine verilen iki katarı birbiriyle karşılaştırır. Katarlar birbirine eşitse, geriye 0 döner. Eğer ilk katar alfabetik olarak ikinciden büyükse, geriye pozitif değer döndürür. Şayet alfabetik sırada ikinci katar birinciden büyükse, geriye negatif değer dönmektedir. Bu dediklerimizi, daha iyi anlaşılması için bir tabloya dönüştürelim:

Dönen Değer	Açıklama
< 0	Katar1, Katar2'den küçüktür.

0	Katar1 ve Katar2 birbirine eşittir.
> 0	Katar1, Katar2'den büyüktür.

strncmp( ) için de aynı kurallar geçerlidir. Tek fark, karşılatırılacak karakter sayısını girmemizdir. *strcmp( )* fonksiyonunda iki katar, *null* karakter işareti çıkana kadar karşılaştırılır. Fakat *strncmp( )* fonksiyonunda, başlangıçtan itibaren kaç karakterin karşılaştırılacağına siz karar verirsiniz.

Her iki fonksiyonu da kapsayan aşağıdaki örneği inceleyelim:

```
#include<stdio.h>
#include<string.h>
int main( void )
{
        int sonuc;
        char ilk_katar[40]="Maymun";
        char ikinci_katar[40]="Maytap";
        sonuc = strcmp( ilk_katar, ikinci_katar );
        printf( "%d\n", sonuc );
        sonuc = strncmp( ilk_katar, ikinci_katar, 3 );
        printf( "%d\n", sonuc );

        return 0;
}
```

İlk önce çağrılan *strcmp( )*, null karakterini görene kadar bütün karakterleri karşılaştıracak ve geriye negatif bir değer döndürecektir. Çünkü "*Maymum*" kelimesi alfabede "*Maytap*" kelimesinden önce gelir; dolayısıyla küçüktür. Fakat ikinci olarak çağırdığımız *strncmp( )* geriye 0 değeri verecektir. Her iki kataın ilk üç harfi aynıdır ve fonksiyonda sadece ilk üç harfin karşılaştırılmasını istediğimizi belirttik. Dolayısıyla karşılaştırmanın sonucunda 0 döndürülmesi normaldir.

* strcat( ) ve strncat( ) ile katar birleştirme

strcat( ) ve *strncat( )* fonksiyonları, bir katarı bir başka katarla birleştirmeye yarar. Fonksiyon adlarında bulunan cat, İngilizce bir kelime olan ve birleştirme anlamına gelen '*concatenate*'den gelmiştir. *strcat( )* kendisine verilen katarları tamamen birleştirirken, *strncat( )* belirli bir eleman sayısına kadar birleştirir. strcat ile ilgili basit bir örnek yapalım.

```
#include<stdio.h>
#include<string.h>
int main( void )
{
        char ad[30], soyad[20];
        char isim_soyad[50];
        printf( "Ad ve soyadınızı giriniz> " );
        scanf( "%s%s", ad, soyad );
        // isim_soyad <-- ad
        strcat( isim_soyad, ad );
        // isim_soyad <-- ad + " "
        strcat( isim_soyad, " " );
        // isim_soyad <-- ad + " " + soyad
        strcat( isim_soyad, soyad );
        printf( "Tam İsim: %s\n", isim_soyad );
        return 0;
}
```

Dilerseniz, *strncat( )* fonksiyonunu da siz deneyebilirsiniz.

* strstr( ) fonksiyonuyla katar içi arama yapma

Bir katar içinde, bir başka katarı aradığınız durumlarda, *strstr( )* fonksiyonu yardımınıza yetişir. *strstr( )* fonksiyonu, bir katar içinde aradığınız bir katarı bulduğu takdirde bunun bellekteki adresini geriye döndürür. Yani dönen değer çeşidi bir pointer'dır. Eğer herhangi bir eşleşme olmazsa geriye bir sonuç dönmez ve pointer *null* olarak kalır. Elbette insanlar için hafıza adreslerinin veya pointer değerlerinin pek bir anlamı olmuyor. Bir katar içinde arama yapıyorsanız, aradığınız yapının katarın neresinde olduğunu tespit etmek için aşağıdaki kodu kullanabilirsiniz:

```
/* strstr(  ) fonksiyon ornegi */
#include<stdio.h>
#include<string.h>
int main( void )
{
        char adres[] = "Esentepe Caddesi Mecidiyekoy Istanbul";
        char *ptr;
        // 'adres' katari icinde, 'koy' kelimesini
        // ariyoruz. Bu amacla strstr(  ) fonksiyonunu
        // kullaniyoruz. Fonksiyon buyuk-kucuk harf
        // duyarlidir. Eger birden fazla eslesme varsa,
        // ilk adres degeri doner. Hic eslesme olmazsa,
        // pointer degeri NULL olur.
        ptr = strstr( adres, "koy" );
        if( ptr != NULL )
                printf( "Başlangıç notkası: %d\n", ptr - adres );
        else
                printf( "Eşleşme bulunamadı.\n" );
        return 0;
}
```

* strchr( ) ve strrchr( ) fonksiyonları

strchr( ) ve *strrchr( )* fonksiyonları, tıpkı *strstr( )* gibi arama için kullanılır. Ancak *strstr( )* fonksiyonu katar içinde bir başka katarı arayabilirken, *strchr( )* ve *strrchr( )* fonksiyonları katar içinde tek bir karakter aramak için kullanılır. *strchr( )*, karakterin katar içindeki ilk konumunu gösterirken; *strrchr( )* fonksiyonu, ilgili karakterin son kez geçtiği adresi verir.

```
#include<stdio.h>
#include<string.h>
int main( void )
{
        char adres[] = "Esentepe Caddesi Mecidiyekoy Istanbul";
        char *ilk_nokta, *son_nokta;
        ilk_nokta = strchr( adres, 'e' );
        son_nokta = strrchr( adres, 'e' );
        if( ilk_nokta != NULL ) {
                printf( "Ilk gorundugu konum: %d\n", ilk_nokta - adres );
                printf( "Son gorundugu konum: %d\n", son_nokta - adres );
        }
        else
                printf( "Eşleşme bulunamadı.\n" );
        return 0;
}
```

* atoi( ) ve atof( ) ile katar dönüşümü

Verilen katarı, sayıya çevirmek gerekebilir. Eğer elinizdeki metni, bir tam sayıya (*int*) çevireceksiniz, *atoi( )* fonksiyonunu kullanmanız gerekir. Şayet dönüşüm sonunda elde etmek istediğiniz değişken tipi, virgüllü sayı ise (*float*), *atof( )* fonksiyonu kullanılır. Her iki fonksiyon *stdlib.h* kütüphanesi içindedir. Bu fonksiyonları kullanırken, *#include<stdlib.h>* komutunu program

başlangıcına yazmalısınız.

```
#include<stdio.h>
#include<stdlib.h>
int main( void )
{
        char kok_iki[] = "1.414213";
        char pi[] = "3.14";
        char tam_bir_sayi[] = "156";
        char hayatin_anlami[] = "42 is the answer";

        printf( "%d\n", atoi( tam_bir_sayi ) );
        printf( "%d\n", atoi( hayatin_anlami ) );
        printf( "%f\n", atof( kok_iki ) );
        printf( "%f\n", atof( pi ) );
        return 0;
}
```

Her iki fonksiyonda rakam harici bir şey görene kadar çalışır. Eğer nümerik ifadeler dışında bir karakter çıkarsa, fonksiyon o noktada çalışmayı keser.

main( ) Fonksiyonuna Argüman Aktarımı

İşlediğimiz bütün derslerde *main( )* fonksiyonu vardı. *main( )* fonksiyonuyla ilgili incelememizi de, fonksiyonlarla ilgili dokuzuncu dersimizde yapmıştık. Ancak main( ) fonksiyonuna hiçbir zaman parametre aktarmadık; aksine parametre almayacağını garantilemek için sürekli olarak *main(void)* şeklinde yazmıştık. Artık main( ) fonksiyonuna nasıl parametre verileceğini göreceğiz. Aşağıdaki kod, parametresi olan bir main( ) fonksiyonunu göstermektedir:

```
#include<stdio.h>
int main( int argc, int *arg[] )
{
        int i;
        for( i = 0; i < argc; i++ ) {
                printf( "%d. argüman: %s\n", i, arg[i] );
        }
        return 0;
}
```

Bu kodu yazıp, "*yeni_komut.c*" adıyla kaydedin. Ardından eğer Linux ve gcc kullanıyrsanız, aşağıdaki komutu kullanarak kodun derlemesini yapın.

```
$ gcc yeni_komut.c -o yeni_komut
```

Yukardaki komut, "*yeni_komut*" adında çalıştırılabilir bir program dosyası oluşturacak. Windows ve Dev-C++ kullanıyorsanız böyle bir komuta gerek yok. Kodu kaydedip, derlediğiniz zaman, çalışma klasörünüzde "*yeni_komut.exe*" adında bir dosya zaten oluşacaktır.

İkinci aşamada, programa parametre göndererek çalıştıracağız. Bunun için gerek Linux gerekse Windows kullanıcılarının yapacağı şey birbirine çok benziyor. Linux kullanıcıları aşağıdaki gibi bir komut girecekler:

```
$ ./yeni_komut Merhaba Dünya Hello World
```

Windows kullanıcılarınınsa, DOS komut istemini açıp, programın kayıtlı olduğu klasöre gelmeleri gerekiyor. Diyelim ki, "*yeni_komut.exe*" "*C:\Belgelerim*" içinde kayıtlı... O hâlde aşağıdaki komutu giriyoruz:

```
C:\Belgelerim> yeni_komut Merhaba Dünya Hello World
```

Her iki işletim sisteminde elde edeceğiniz sonuç aynı olacaktır:

```
0. argüman: ./yeni_komut
1. argüman: Merhaba
2. argüman: Dünya
3. argüman: Hello
4. argüman: World
```

Dışardan gelen argümanla çalışan bir başka main( ) fonksiyonu oluşturalım. Toplama ve çıkartma işlemini alacağı argümanlara göre yapan bir programı aşağıda bulabilirsiniz:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main( int argc, char *arg[] )
{
        // Eger eksik arguman soz konusuysa,
        // program calismamalidir.
        if( argc < 4 ) {
                printf( "Hata: Eksik argüman!\n");
                return;
        }

        float sayi_1, sayi_2;
        char islem_tipi[2];
        sayi_1 = atof( arg[1] );
        strcpy( islem_tipi, arg[2] );
        sayi_2 = atof( arg[3] );

        // Verilen sembolun neye esit oldugu asagidaki
        // if-else if merdiveniyle saptaniyor.
        if( !strcmp( islem_tipi, "+" ) )
                printf( "Toplam: %.2f\n", sayi_1 + sayi_2 );
        else if( !strcmp( islem_tipi, "-" ) )
                printf( "Fark: %.2f\n", sayi_1 - sayi_2 );
        else
                  printf( "Hatalı işlem!\n" );
        return 0;
}
```

Programı çalıştırmak için şu tarz bir komut verdiğimizi düşünelim:

```
$ ./hesapla 4 + 12
```

Programı bu şekilde çalıştırdığınız zaman argümanların, parametrelere atanması aşağıdaki gibi olur:

arg[0]	arg[1]	arg[2]	arg[3]
./hesapla	4	+	12

Bütün fonksiyonlara, program içersinden argüman aktarımı yaparken; main( ) fonksiyonuna program dışından değer gönderebiliyoruz. Unix komutlarının hemen hemen hepsi bu şekildedir. DOS komutlarının birçoğu da böyle yazılmıştır. main( ) fonksiyonun parametre alıp almaması gerektiğine, ihtiyacınıza göre sizin karar vermeniz gerekir.

Örnek Sorular

Soru 1: Kendisine verilen bir katarın boyutunu bulan fonksiyonu yazınız. (Çözüm için *strlen( )* fonksiyonunu kullanmayınız.)

Cevap 1:

```
#include<stdio.h>
#include<string.h>
int katar_boyutu_bul( char [] );
int main( void )
{
        char test_katari[50];
        strcpy( test_katari, "ABCDEF" );
        printf( "Katar boyutu: %d\n", katar_boyutu_bul( test_katari ) );
        return 0;
}
int katar_boyutu_bul( char katar[] )
{
        int i;
        for( i = 0; katar[ i ]!='\0'; i++ );

        return i;
}
```

Soru 2: Tersinden de aynı şekilde okunabilen kelime, cümle veya mısraya '*palindrome*' denmektedir. Adı palindrome( ) olan ve verilen katarın tersinin kendisine eşit olduğu durumda geriye 1; aksi hâlde 0 döndüren fonksiyonu yazınız.

Cevap 2:

```
#include<stdio.h>
#include<string.h>
int palindrome( char [] );
int main( void )
{
        char test_katari[50];
        strcpy( test_katari, "ABBA" );
        printf( "%d\n", palindrome( test_katari ) );
        return 0;
}
int palindrome( char katar[] )
{
        int boyut =0 , i;
        // Once katar boyutu bulunuyor
        for( boyut = 0; katar[ boyut ]!='\0'; boyut++ );


        for( i = 0; i < boyut/2; i++ ) {
                if( katar[i] != katar[ boyut - i - 1 ] )
                        return 0;
        }
        return 1;
}
```

Soru 3: Aşağıdaki gibi çalışıp, çıktı üretebilecek "*ters_cevir*" programını oluşturunuz.

```
$ ./ters_cevir Merhaba Dunya Nasilsin?
abahreM aynuD ?nislisaN
```

Cevap 3:

```
#include<stdio.h>
#include<string.h>
void ters_cevir( char [] );
int main( int argc, int arg[] )
{
        int i;
        for( i = 1; i < argc; i++ ) {
                ters_cevir( arg[i] );
        }
```

```
        printf("\n");
        return 0;
}
void ters_cevir( char katar[] )
{
        int i, boyut;
        for( boyut = 0; katar[ boyut ]!='\0'; boyut++ );

        for( i = 0; i < boyut; i++ )
                printf("%c", katar[ boyut - 1 - i ] );
        printf(" ");
}
```

Yeni Değişken Tipi Oluşturma

Kullandığımız birçok değişken tipi oldu. Tam sayıları, karakterleri, virgüllü sayıları, katarları gördük. Ancak kullanabileceğimiz değişken tipleri bunlarla sınırlı değildir. Kendi değişken tiplerimizi, yaratabiliriz. Örneğin *boolean* diye yeni bir tip yaratarak, bunun alabileceği değerleri *true* ve *false* olarak belirleyebiliriz; üçüncü bir ihtimal olmaz. Ya da mevsimler diye bir değişken tipi belirleyip, alabileceği değerleri aylar olarak kısıtlayabiliriz. İşte bu tarz işlemleri yapmak için **enum** kullanılır. *enum* kelimesi, **enumerator** yani '*sayıcı*', '*numaralandırmacı*'dan gelmektedir.

Hayat, rakamlarla ifade edilebilir. Bunun en bariz uygulamalarını programlama yaparken görürsünüz. Bir karakter olan A harfi, ASCII Tablo'da 65 sayısına denk düşer; büyük B harfiyse 66'dır ve bu böyle devam eder. Bilgisayarınız işaretlerden, sembollerden, karakterlerden anlamaz. Onun için tek gerçeklik sayılardır. İşte *enum* bu felsefeye hizmet ediyor. Örneğin doğruyu göstermek için 1, yanlış içinse 0'ı seçersek; yeni bir değişken tipi belirlemiş oluruz. Bilgisayar doğrunun ya da yanlışın ne olduğunu bilmez, onun için sadece 0 ve 1 vardır. Ancak insanların yararına, okunurluğu artan programlar ortaya çıkar. İsterseniz, *boolean* diye tabir ettiğimiz değişken tipini oluşturalım:

```
#include<stdio.h>
int main( void )
{
        // Degisken tipinin nasil olacagini tanimliyoruz
        enum boolean {
                false = 0,
                true = 1
        };
        // Simdi de 'dogru_mu' adinda bir degisken
        // tanimliyoruz
        enum boolean dogru_mu;
        // Tanimladigimiz 'dogru_mu' degiskenine
        // deger atayip, bir alt satirda da
        // kontrol yapiyoruz.
        dogru_mu = true;
        if( dogru_mu == true )
                printf( "Doğru\n" );
        return 0;
}
```

Daha önce *boolean* diye bir veri tipi bulunmuyordu. Şimdiyse, iki farklı değeri olabilen, doğru ve yanlışları göstermekte kullanabileceğimiz yeni bir değişken tipi oluşturduk. Yanlışı göstermek için 0; doğruyu ifade etmek içinse 1 rakamları kullandık. Yanlışın ve doğrunun karşılığını belirtmemiz gerekmiyordu; *boolean* veri tipini tanımlarken, 0 ve 1 yazmadan sadece *false* ya da *true* da yazabilirdik. Programınız derlenirken, karşılık girilmeyen değerlere sırayla değer atanmaktadır. İlla ki sizin bir eşitlik oluşturmanız gerekmez. Mesela üç ana rengi (Kırmızı, Sarı ve Mavi)alabilecek *ana_renkler* veri tipini oluşturalım:

```
#include<stdio.h>
int main( void )
{
        // Degisken tipinin nasil olacagini tanimliyoruz
        enum ana_renkler {
                Kirmizi,
                Mavi,
                Sari
        };

        // Degiskeni tanimliyoruz.
        enum ana_renkler piksel;

        // Degisken degerini Mavi olarak belirliyoruz.
        // Dilersek Sari ve Kirmizi da girebiliriz.
        piksel = Mavi;

        // Degisken degeri karsilastiriliyor.
        if( piksel == Kirmizi )
                printf( "Kırmızı piksel\n" );
        else if( piksel == Mavi )
                printf( "Mavi piksel\n" );
        else
                printf( "Sarı piksel\n" );
        return 0;
}
```

Kirmizi, Mavi ya da Sari'nin nümerik değerini bilmiyoruz; muhtemelen birden başlamış ve sırayla üçe kadar devam etmişlerdir. Değerlerin nümerik karşılığını bilmesek bile, bu onlarla işlem yapmamızı engellemiyor. Bir önceki örnekte olduğu gibi rahatça kullanabiliyoruz.

Oluşturduğumuz yeni veri tiplerinden, değişken tanımlarken her defasında *enum* koyduğumuzu görmüşsünüzdür. Bunu defalarca yazmak yerine iki alternatif biçim bulunuyor. Birincisi yeni veri tipini oluştururken, değişkeni tanımlamak şeklinde... boolean örneğimize geri dönüp, farklı şekilde nasıl tanımlama yapabileceğimizi görelim:

```
#include<stdio.h>
int main( void )
{
        // Yeni veri tipini olusturuyoruz
        // Ayrica yeni veri tipinden,
        // bir degisken tanimliyoruz.
        enum boolean {
                false = 0,
                true = 1
        } dogru_mu;

        dogru_mu = true;
        if( dogru_mu == true )
                printf( "Doğru\n" );
        return 0;
}
```

Yukarda gördüğünüz yöntem, yeni veri tipini oluşturduğunuz anda, bu veri tipinden bir değişken tanımlamanızı sağlar. Her seferinde *enum* yazmanızdan kurtaracak diğer yöntemse, **typedef** kullanmaktan geçer. *typedef* kullanımı şu şekildedir:

```
typedef veri_tipi_eski_adi veri_tipi_yeni_adi
```

Kullanacağınız *typedef* ile herhangi bir değişken tipini, bir başka isimle adlandırabilirsiniz. Örneğin yazacağınız "*typedef int tam_sayi;*" komutuyla, değişken tanımlarken *int* yerine *tam_sayi* da yazabilirsiniz. Bunun *enum* için uygulamasına bakalım:

```
#include<stdio.h>
int main( void )
{
        // Yeni veri tipini olusturuyoruz
        // Ayrica yeni veri tipinden,
        // bir degisken tanimliyoruz.
        enum boolean {
                false = 0,
                true = 1
        };
        // Alttaki komut sayesinde, boolean
        // veri tipini tek adimda yaratabiliyoruz.
        typedef enum boolean bool;

        bool dogru_mu;

        dogru_mu = true;
        if( dogru_mu == true )
                printf( "Doğru\n" );
        return 0;
}
```

Özel Değişken Tipleri ve Fonksiyonlar

enum konusu genişletilebilir. Örneğin *enum* tanımlamasını, global olarak yaparsanız, fonksiyon parametresi olarak kullanabilirsiniz. Çok basit bir fonksiyon oluşturalım. Alacağı değişken bilgisine göre, ekrana ona dair bilgi yazdırılsın:

```
#include<stdio.h>
// Ay listesini olusturuyoruz. Ocak
// ayi 1 olacak sekilde, aylar sirayla
// numerik degerler aliyor.
enum ay_listesi {
        ocak = 1, subat, mart, nisan,
        mayis, haziran, temmuz, agustos,
        eylul, ekim, kasim, aralik
};
// Degisken tanimlamasini kolaylastirmak
// icin typedef kullaniliyoruz. aylar diyerek
// tanimlama yapmak mumkun hale geliyor.
typedef enum ay_listesi aylar;

void ay_ismini_yazdir( aylar );
int main( void )
{
        // aylar tipinde bir degisken
        // yaratip, 'kasim' degerini atiyoruz.
        aylar bu_ay = kasim;
        // kasim, numerik olarak 11'i ifade edecektir.
        printf( "%d. ay: ", bu_ay );
        // fonksiyonumuzu cagiriyoruz.
        ay_ismini_yazdir( bu_ay );
        return 0;
}
// Kendisine verilen aylar tipindeki degiskene gore
// hangi ayin oldugunu ekrana yazmaktadir.
void ay_ismini_yazdir( aylar ay_adi )
{
        switch( ay_adi ) {
                case ocak: printf( "Ocak\n" );break;
                case subat: printf( "Şubat\n" );break;
                case mart: printf( "Mart\n" );break;
```

```
                case nisan: printf( "Nisan\n" );break;
                case mayis: printf( "Mayıs\n" );break;
                case haziran: printf( "Haziran\n" );break;
                case temmuz: printf( "Temmuz\n" );break;
                case agustos: printf( "Ağustos\n" );break;
                case eylul: printf( "Eylül\n" );break;
                case ekim: printf( "Ekim\n" );break;
                case kasim: printf( "Kasım\n" );break;
                case aralik: printf( "Aralık\n" );break;
        }
}
```

Gördüğünüz gibi *enum* ile oluşturacağınız özel veri tiplerini fonksiyonlara aktarmak mümkün. *enum* aracılığı ile yeni bir değişken tipi yaratmak, birçok konuda işinizi basit hâle getirir. Özellikle gruplandırılması/tasnif edilmesi gereken veriler varsa, *enum* kullanmak yararlıdır. Örnek olması açısından aşağıda *enum* ile tanımlanmış bazı veri tiplerini bulabilirsiniz:

```
enum medeni_durum { bekar, evli, dul };
enum medeni_durum ayse = bekar;

enum egitim_durumu { ilkokul, ortaokul, lise, universite, master };
enum egitim_durumu ogrenci;

enum cinsiyet { erkek, kadin };
enum cinsiyet kisi;
```

Yapılar (Structures)

Yapılar (structures); tam sayı, karakter vb. veri tiplerini gruplayıp, tek bir çatı altında toplar. Bu gruplandırma içinde aynı ya da farklı veri tipinden dilediğiniz sayıda eleman olabilir. Yapılar, nesne tabanlı programlama (Object Oriented Programming) dilleri için önemli bir konudur. Eğer Java, C# gibi modern dillerle çalışmayı düşünüyorsanız, bu konuya daha bir önem vermeniz gerekir.

Vakit kaybetmeden bir örnekle konumuza girelim. Doğum günü bilgisi isteyip, bunu ekrana yazdıran bir program oluşturalım:

```
#include<stdio.h>
int main( void )
{
        struct {
                int yil;
                int ay;
                int gun;
        } dogum_gunu;

        printf( "Doğum gününüzü " );
        printf( "GG-AA-YYYY olarak giriniz> ");
        scanf( "%d-%d-%d",      &dogum_gunu.gun,
                                &dogum_gunu.ay,
                                &dogum_gunu.yil );
        printf( "Doğum gününüz: " );
        printf( "%d/%d/%d\n",   dogum_gunu.gun,
                                dogum_gunu.ay,
                                dogum_gunu.yil );
        return 0;
}
```

Bir kullanıcının doğum gününü sorup, gün, ay ve yıl bilgilerini üç farklı *int* değişken içersinde tutabilirdik. Ancak gruplandırmak her zaman daha iyidir. Hem yaptığınız işlerin takibi kolaylaşır, hem de hata yapma riskinizi azaltır. Bunu daha iyi anlatmak için aynı anda sizin ve iki kardeşinizin

doğum günlerini soran bir program yazalım:

```
#include<stdio.h>
int main( void )
{
        struct {
                int yil;
                int ay;
                int gun;
        } siz, kiz_kardes, erkek_kardes;

        printf( "Doğum gününüzü giriniz> ");
        scanf( "%d-%d-%d",      &siz.gun,
                                &siz.ay,
                                &siz.yil );
        printf( "Kız kardeşiniz> " );
        scanf( "%d-%d-%d",      &kiz_kardes.gun,
                                &kiz_kardes.ay,
                                &kiz_kardes.yil );
        printf( "Erkek kardeşiniz> " );
        scanf( "%d-%d-%d",      &erkek_kardes.gun,
                                &erkek_kardes.ay,
                                &erkek_kardes.yil );
        return 0;
}
```

Eğer yapılardan (structures) yararlanmasaydık; üç kişinin doğum günü bilgilerini tutmak için toplamda 9 adet farklı değişken tanımlamak gerekecekti. Tanımlama zahmeti bir yana, değişkenlerin karıştırılma ihtimali de ayrı bir sıkıntı yaratacaktı. Sadece üç değişken olarak düşünmeyelim; nüfus cüzdanı bilgilerini soracağımız bir program, yirminin üzerinde değişkene ihtiyaç duyar. Bu kadar çok değişken barındırıp, yapıları kullanmadan hazırlanacak bir programı görmek bile istemezsiniz.

Yapıları kullanmanın bir diğer avantajı, kopyalama konusundadır. Örneğin, sizin bilgilerinizi, erkek kardeşinize kopyalamak için tek yapmanız gereken, "*erkek_kardes = siz*" yazmaktır. Bu basit işlem ilgili bütün değişkenlerin kopyalamasını yapar.

İç İçe Yapılar

Bir yapı içersine tıpkı bir değişken koyar gibi, bir başka yapı da koyulabilir. Örneğin kullanıcı bilgisi alan bir programda, isim, boy ve doğum tarihi bilgilerini aynı yapı altına toplayabilirsiniz. Ancak doğum tarihi bilgilerini daha alt bir yapı içersinde tutmak yararlı olabilir. Bunu koda dökersek şöyle olur:

```
#include<stdio.h>
int main( void )
{
        struct {
                char isim[40];
                int boy;
                struct {
                        int yil;
                        int ay;
                        int gun;
                } dogum_bilgileri;
        } kisi;

        printf( "Adınız: " );
        scanf( "%s", kisi.isim );
        printf( "Boyunuz: " );
        scanf( "%d", &kisi.boy );
```

```
        printf( "Doğum tarihi: ");
        scanf( "%d-%d-%d",      &kisi.dogum_bilgileri.gun,
                                &kisi.dogum_bilgileri.ay,
                                &kisi.dogum_bilgileri.yil );

        printf( "Girilen bilgiler:\n" );
        printf( "İsim: %s\n", kisi.isim );
        printf( "Boy: %d\n", kisi.boy );
        printf( "Doğum tarihi: %d/%d/%d\n",     kisi.dogum_bilgileri.gun,
                                                kisi.dogum_bilgileri.ay,
                                                kisi.dogum_bilgileri.yil );
        return 0;
}
```

Alt yapıya ulaşmak için nokta kullanıp, ardından yapının adını yazdığımızı görüyorsunuz. Yapıları kullanarak, bir arada durması yararlı olan değişkenleri gruplarız. İç içe yapıları kullanaraksa, bu gruplandırmayı daha da ufak boyutlara indirgemekteyiz. Kısacası her şey, daha derli toplu çalışma için yapılıyor. Yoksa programın temelinde değişen bir şey yok.

Yapı Etiketleri

Yapılara etiket koyabilir ve etiketleri kullanarak ilgili yapıyı temel alan değişkenler tanımlayabilirsiniz. Az evvel yaptığımıza benzer bir örnek yapalım:

```
#include<stdio.h>
#include<string.h>
int main( void )
{
        // sahis_bilgileri, yapimizin
        // etiketidir.
        struct sahis_bilgileri {
                char isim[40];
                int boy;
        };

        // Yapidan iki adet degisken
        // tanimliyoruz.
        struct sahis_bilgileri kisi_1;
        struct sahis_bilgileri kisi_2;

        // Birinci sahsin bilgilerini
        // kaydediyoruz.
        strcpy( kisi_1.isim, "AHMET" );
        kisi_1.boy = 170;

        // Ikinci sahsin bilgilerini
        // kaydediyoruz.
        strcpy( kisi_2.isim, "MEHMET" );
        kisi_2.boy = 176;

        return 0;
}
```

Yapıların etiket konarak tanımlanması, daha mantıklıdır. Aksi hâlde sadece yapıyı oluştururken tanımlama yaparsınız. Etiket koyduğunuz zamansa, programın herhangi bir yerinde istediğiniz kadar yapı değişkeni tanımlayabilirsiniz. Önemli bir noktayı belirtmek isterim: yapılarda etiket kullandığınız zaman elinizde sadece bir şablon vardır. O etiketi kullanarak yapıdan bir değişken yaratana kadar, üzerinde işlem yapabileceğiniz bir şey olmaz. Yapı (structure) bir kalıptır; bu kalıbın etiketini kullanarak değişken tanımlamanız gerekir.

Yapılarda İlk Değer Atama

Yapılarda da ilk değer ataması yapabilirsiniz. Aşağıdaki örnek etiket kullanmadan oluşturduğunuz yapılarda, ilk değer atamasının nasıl olduğunu göstermektedir. '*kisi*' isimli yapı içersinde bulunan *isim* ve *boy* değişkenlerine sırasıyla *Ali* ve *167* değerleri atanmaktadır.

```
#include<stdio.h>
int main( void )
{
        // kisi adinda bir yapi olusturulup
        // baslangic degerleri 'Ali' ve '167'
        // olacak sekilde atanir.
        struct {
                char isim[40];
                int boy;
        } kisi = { "Ali", 167 };

        return 0;
}
```

Etiket kullanarak oluşturduğunuz yapılarda, ilk değer ataması değişkenlerin tanımlanması aşamasında gerçekleşir. Önce yapıyı kurar ve ardından değişken tanımlarken, ilk değerleri atarsınız. Kullanımı aşağıdaki kod içersinde görülmektedir:

```
#include<stdio.h>
int main( void )
{
        // sahis_bilgileri adinda bir yapi
        // olusturuyoruz
        struct sahis_bilgileri {
                char isim[40];
                int boy;
        };

        // sahis_bilgileri yapisindan kisi adinda
        // bir degisken tanimliyoruz. Tanimlama
        // esnasinda atanacak ilk degerler belirleniyor.
        struct sahis_bilgileri kisi = { "Ali", 167 };

        return 0;
}
```

Bir yapı değişkenine, ilk değer ataması yapıyorsanız sıra önemlidir. Atayacağınız değerlerin sırası, ilgili değişkenlere göre olmalıdır. Yani ilk yazacağınız değer, ilk yapı içi değişkene; ikinci yazacağınız değer, ikinci yapı içi değişkene atanır. Sırayı şaşırmadığınız sürece bir problem yaşamazsınız. Aksi durumda, yanlış değer yanlış değişkene atanacaktır. Sırayı şaşırmak için, ekstra ayraç işaretleri kullanabilirsiniz. Örneğin { "Mehmet", 160, 23, 3, 1980 } yerine { "Mehmet", 160, {23, 3, 1980} } yazmakta mümkündür.

Yapı Dizileri

Veri tiplerine ait dizileri nasıl oluşturacağımızı biliyoruz. Bir tam sayı dizisi, bir karakter dizisi rahatlıkla oluşturabiliriz. Benzer şekilde yapı (structure) dizileri de tanımlanabilir. 3 kişilik bir personel listesi tutacağımızı düşünüp, ona göre bir program oluşturalım. Her eleman için ayrı ayrı değişken tanımlamaya gerek yoktur; yapılardan oluşan bir dizi yaratabiliriz.

```
#include<stdio.h>
int main( void )
{
        int i;
        // Dogum tarihi tutmak icin
```

```
        // 'dogum_tarihi' adinda bir yapi
        // olusturuyoruz
        struct dogum_tarihi {
                int gun;
                int ay;
                int yil;
        };

        // Kisiye ait bilgileri tutmak
        // icin 'sahis_bilgileri' adinda
        // bir yapi kuruluyor.
        struct sahis_bilgileri {
                char isim[40];
                int boy;
                // Yapi icinde bir baska yapiyi
                // kullanmak mumkundur. dogum_tarihi
                // yapisindan 'tarih' adinda bir
                // degisken tanimlaniyor.
                struct dogum_tarihi tarih;
        };

        // Dizi elemanlarina ilk deger ataması yapiyoruz. Dilerseniz
        // klavyeden deger girmeyi tercih edebilirsiniz.
        struct sahis_bilgileri kisi[3] = { "Ali", 170, { 17, 2, 1976 },
                                           "Veli", 178, { 14, 4, 1980 },
                                           "Cenk", 176, { 4, 11, 1983 } };

        // Yapi dizisi yazdiriliyor:
        for( i = 0; i < 3; i++ ) {
                printf( "Kayıt no.: %d\n", ( i + 1 ) );
                printf( "Ad: %s\n", kisi[i].isim );
                printf( "Boy: %d\n", kisi[i].boy );
                printf( "Doğum Tarihi: %d/%d/%d\n\n", kisi[i].tarih.gun,
                                                      kisi[i].tarih.ay,
                                                      kisi[i].tarih.yil );
        }

        return 0;
}
```

Tek bir yapı değişkeniyle, bir yapı dizisi arasında büyük fark bulunmuyor. Sadece köşeli parantezlerle eleman indisini belirtmek yetiyor. Yoksa, değer okuma, değer yazma... bunların hepsi tıpatıp aynı. Bu yüzden ayrıca detaya inmiyorum.

Yapı Dizilerine Pointer ile Erişim

Kambersiz düğün olmaz. Aynı şekilde, dizilerden bahsettiğimiz bir yerde pointer'lardan bahsetmemek mümkün değil. Bir yapı dizisinin başlangıç adresini pointer'a atadığınız takdirde, elemanlara bu işaretçi üzerinde de ulaşabilirsiniz. Bir üstteki örneğimizi pointer'larla kullanalım:

```
#include<stdio.h>
int main( void )
{
        int i;
        // Dogum tarihi tutmak icin
        // 'dogum_tarihi' adinda bir yapi
        // olusturuyoruz
        struct dogum_tarihi {
                int gun;
                int ay;
                int yil;
        };
```

```
        // Kisiye ait bilgileri tutmak
        // icin 'sahis_bilgileri' adinda
        // bir yapi kuruluyor.
        struct sahis_bilgileri {
                char isim[40];
                int boy;
                // Yapi icinde bir baska yapiyi
                // kullanmak mumkundur. dogum_tarihi
                // yapisindan 'tarih' adinda bir
                // degisken tanimlaniyor.
                struct dogum_tarihi tarih;
        };

        struct sahis_bilgileri *ptr;

        // Dizi elemanlarina ilk deger atamasi yapiyoruz. Dilerseniz
        // klavyeden deger girmeyi tercih edebilirsiniz.
        struct sahis_bilgileri kisi[3] = { "Ali", 170, { 17, 2, 1976 },
                                           "Veli", 178, { 14, 4, 1980 },
                                           "Cenk", 176, { 4, 11, 1983 } };

        // Yapi dizisi yazdiriliyor:
        for( i = 0, ptr = &kisi[0]; ptr <= &kisi[2]; ptr++, i++ ) {
                printf( "Kayıt no.: %d\n", ( i + 1 ) );
                printf( "Ad: %s\n", ptr->isim );
                printf( "Boy: %d\n", ptr->boy );
                printf( "Doğum Tarihi: %d/%d/%d\n\n", ptr->tarih.gun,
                                                      ptr->tarih.ay,
                                                      ptr->tarih.yil );
        }

        return 0;
}
```

Pointer'ın tanımlamasını yaparken, '*sahis_bilgileri*' şablonundan türetilen değişkenlerin işaret edileceğini bildirmemiz gerekiyor. Yazmış olduğumuz "*struct sahis_bilgileri *ptr;*" kodu bundan kaynaklanmaktadır. for döngüsüne gelirsek, *kisi* isimli yapı dizisinin ilk elemanının adresini, *ptr* işaretçisine atadığımızı görmüşsünüzdür. Her seferinde de, *ptr* değeri bir adres bloğu kadar artmaktadır. Döngünün devamı, adresin son dizi elemanından küçük olmasına bağlıdır. Kullandığımız **->** operatörüyse, pointer ile dizi elemanlarını göstermemizi sağlar. Bu cümleler size muhtemelen karışık gelecektir -ki bu kesinlike normal... İnanıyorum ki kodu incelerseniz, durumu daha basit kavrarsınız.

Yapılar ve Fonksiyonlar

enum ile yarattığımız değişken tiplerini, fonksiyonlarda kullanmak için global olarak tanımlıyorduk. Yapıları, fonksiyonlarda kullanılmak için izlenecek yöntem aynıdır; yine global tanımlanması gerekir. Çok basit bir örnekle yapıların fonksiyonlarla kullanımını görelim:

```
#include<stdio.h>
#include<string.h>
struct sahis_bilgileri {
        char isim[40];
        int boy;
};

struct sahis_bilgileri bilgileri_al( void );
void bilgileri_goster( struct sahis_bilgileri );

int main( void )
```

```
{
        struct sahis_bilgileri kisi;
        kisi = bilgileri_al( );
        bilgileri_goster( kisi );

        return 0;
}
struct sahis_bilgileri bilgileri_al( void )
{
        struct sahis_bilgileri sahis;
        printf( "İsim> " );
        gets( sahis.isim );
        printf( "Boy> " );
        scanf( "%d", &sahis.boy );
        return sahis;
}
void bilgileri_goster( struct sahis_bilgileri sahis )
{
        printf( "Ad: %s\n", sahis.isim );
        printf( "Boy: %d\n", sahis.boy );
}
```

Dinamik Yapılar

Dinamik bellek tahsis etmenin ne olduğunu, niçin bunu kullandığımızı açıklamaya gerek duymuyorum. Daha önceki derslerimizde bu konuya yer vermiştik. Çok basit bir örnekle dinamik yapıların kullanımı göstermek yeterli olacaktır:

```
struct sahis_bilgileri *ptr;
ptr = calloc( 1, sizeof( struct sahis_bilgileri ) );
free( ptr );
```

Üç adımda, yapıları dinamik kullanmayı görüyorsunuz. En başta ptr adında bir pointer tanımlıyoruz. İkinci aşamada, bellek ayrımı yapılıyor. Bu örnekte, sadece tek değişkenlik yer ayrılıyor. ptr ile işimiz bittikten sonra, *free( )* fonksiyonuyla, belleği boşaltıyoruz. Sadece üç temel adımla, yapılarda dinamik bellek kullanımını sağlayabilirsiniz. Yalnız *calloc( )* (ya da *malloc( )*) fonksiyonunun *stdlib.h* altında olduğunu unutmayın. (Bu yüzden kodun başına *#include<stdlib.h>* yazmak gerekmektedir.)

Yapılarda typedef Kullanımı

enum konusuna tekrar dönüyoruz. Hatırlayacağınız üzere, *typedef* orada da geçmişti. *typedef* kullanarak her seferinde fazladan *enum* yazma zahmetinden kurtuluyorduk. *typedef*, yapılar için de kullanılmaktadır. Her defasında tekrar tekrar *struct* yazmak yerine, bir kereye mahsus *typedef* kullanarak bu zahmetten kurtulabilirsiniz. Aşağıdaki gibi yazacağınız kodla, tekrar tekrar struct kelimesi kullanmanıza gerek kalmayacaktır.

```
typedef struct sahis_bilgileri kisi_bilgileri;
```

Noktalarken...

C programlama diline dair anlatımlarımız burada bitiyor. Bu demek değildir ki; C üzerine her şeyi anlattık; aksine daha birçok konu bulunuyor. (Örneğin dosya işlemleri, union kullanımı vb... konulara hiç değinilmedi.) Ancak şimdiye kadar öğrendikleriniz, bundan sonrasını öğrenebilmeniz için size temel teşkil edecektir. Programlama dili öğrenmek, yabancı dil öğrenmekle hemen hemen aynıdır. İngilizce üzerine dersler aldığınızda, kimse Shakespeare olacağınızı söyleyemez. Ama çok çalışıp kendinizi geliştirmek size bağlıdır. Programlama dilleri de aynen böyle... Burada ya da bir

başka kaynakta anlatılanlarla işin duayeni olamazsınız; fakat işi anlar duruma gelirsiniz. Bundan sonrası sizin elinizdedir... Lütfen bol bol pratik yapıp, olabildiğince çok algoritma kurun. Sizlere temel programlama gramerini vermeye çalıştım; umarım sonunda hepiniz birer "Macbeth" yazarsınız!