C Hakkında Bilgiler

- Jian Huang, Dr. Jim Plank'ın referans notları
- CS360
- Programlar http://~huangj/cs360/notes/CStuff adresinde
- Web adresi: http://www.cs.utk.edu/~huangj/cs360/360/notes/CStuff/lecture.html

C Nedir?

Sistem programlama için yegâne dil C'dir. C bu şöhretini "programcıyı sisteme yaklaştıran" en iyi ve tek dil olmasına borçludur. C programlama dili hakkında bir kaç önemli temel bilgi var. Aşağıda bunlardan en önemli dört tanesini özetledim:

C dili işaretçiler, yaygın sayısal tipler ve topluluk veri tiplerini oluşturmak için yollar da dâhil olmak üzere çok sayıda gerçek veri tipi içerir.

- C dili işaretçi işlemleri için bir aritmetik tanımlar. Bu işlemler pek çok işlemcide yaygın olduğu şekilde gerçek indeks kayıtçıları temelli adreslemeyi doğal olarak eşler.

- C dili operatör yönünden oldukça zengindir. Aslında makina dili komut setinin hemen hemen tüm yeteneklerine doğrudan erişim sağlar.

- C dönüşümleri ve dolaylamaları etkileyecek sade bir sentaks sağlar.

Doğal ve kullanımı basit görünmesine rağmen bu işlevsellikler akıllıca tasarlanmış ve optimize edilmiştir. Buna rağmen, gerekli hususlar bilinmezse, kodlar hatalı olma eğilimindedir ve hatta bu sebeple orta ölçekli bir sistem çökebilir.

Detaylara girmeden önce, C hakkında çok "iyi" olmayan bazı gerçeklerde bakalım. Fark etmişsinizdir, C'nin mevcut şöhretine rağmen birçok bilimsel kullanıcı Fortran'da programlamaktadır.

Belli ki sebepleri var. Birincisi, C karmaşık sayılar, vektörler ve matrisler vb. gibi hazır yapılara sahip değildir. Gelişmiş matematik rutinleri C kullanarak programlamak kolay değildir. İkincisi, yüksek kalite bilimsel ve sayısal kütüphanelere sahip değildir. Aslında, yakın zamana kadar C'nin matematik kütüphaneleri sadece çift hassasiyetli idi. Sürekli en yüksek hassasiyeti kullanmanın yanlış bir tarafı olmadığını iddia edenler olabilirse de, hepimiz iyi biliyoruz ki minör rakamların sadece bir kaçını kaybetmeyi göze alabilirsek müthiş performans artışı ortaya çıkacaktır. Pratik olarak söylersek, %30'dan %50'ye kadar olan performans kazançları genellikle mümkündür. Günler sürebilecek benzetimler için bu fark büyük önem taşımaktadır.

C temelde bir sistem programlama dili olarak tasarlandı. C'nin diğer uygulamalara eş performans göstermesini beklemek adil ve gerçekçi olmayacaktır. Biz bilgisayar bilimcileri bu farkı anlamış olmalıyız.

C'de Tipler

C de değişkenlerin sahip olabileceği üç çeşit tip vardır. Sayısal tipler, topluluk tipleri ve işaretçiler. C'de bir şeyleri doğru yapma işinin yarısı tipleri karıştırmaktan kendinizi uzak tutmanızdır. Bu derste tipler konusunu biraz detaylandıracağız.

Sayısal Tipler

C'de 7 sayısal tip vardır:

- char: 1 byte

- short: 2 byte

- int: 4 byte
- long: 4 byte (bazı sistemlerde 8 byte)
- float: 4 byte
- double: 8 byte
- (işaretçi: 4 byte (bazı sistemlerde 8 byte))

Burada listelenenler yaygın bir sistemde ilgili tipe karşılık gelen her bir değişkenin boyutlarıdır. Bir tipin boyutunu öğrenmenin en uygun yolu sizeof(int) şeklinde sizeof makrosunu kullanmaktır.

Bütün bunlar size tanıdık gelebilir. Bir değişkeni global değişken, prosedür parametresi veya yerel değişken olarak üç şekilde tanımlayabilirsiniz. Mesela, aşağıdaki p1.c programına bakınız:

(Bu ve diğer bütün ders notlarındaki programları ve makefile dosyasını kendi dizininize kopyalayabilirsiniz ve sonra orada make kullanmak suretiyle derleyebilirsiniz. Mesela p1 programı için "make p1" demeniz gerekir.)

```
#include < stdio.h >

int i;

main(int argc, char **argv)
{
  int j;

  j = argc;
  i = j;
  printf("%d\n", i);
}
```

Burada üç tane sayısal değişken var: i, j ve argc. i global bir değişkendir, j yerel bir değişkendir, argc bir fonksiyon parametresidir. Sayısal tipler oldukça basittir. Bir şeylerin ters gidebileceğini düşünmeden onları fonksiyonlara parametre olarak geçebilirsiniz, fonksiyonlardan değer olarak döndürebilirsiniz.

Topluluk Tipleri

Diziler ve yapılar C'de topluluk tipleridir. Sayısal tiplerden daha karmaşıktırlar. Bir diziyi global ya da yerel değişken olarak tanımlayabilirsiniz. Mesela:

```
#include < stdio.h >

char s1[15];

main(int argc, char **argv)
{
  char s2[4];

}
```

s1 15 karakterlik global bir dizidir, s2 ise 4 karakterlik yerel bir dizidir.

Eğer bir dizi açıkça tanımlanmışsa artık onu başka bir diziye atayamazsınız. Mesela p2.c'ye bakın:

```
#include < stdio.h >

char s1[15];

main(int argc, char **argv)
{
  char s2[4];
```

```
  s2 = "Jim";
}
```

s0 = "Jim" ifadesi C'de illegaldir, çünkü s2 açık tanımlanmıştır. Bu programı derlemeyi denerseniz gcc aşağıdaki şekilde hata verecektir:

```
UNIX> gcc -o p2 p2.c
p2.c: In function `main':
p2.c:10: incompatible types in assignment
UNIX>
```

Hatırlatma: Eğer x açık tanımlanmış bir topluluk tipi (yapı ya da dizi) ise hiç bir zaman “x= something” diyemezsiniz. Daima hata verecektir.

Ancak, "something=x" diyebilirsiniz. Bunu aşağıda açıklayacağız.

İşaretçiler

C'de işaretçiler birçok insanın karıştırdığı bir konudur. Bir işaretçi basitçe bir bellek gözünü işaret eder. Bellek tahsisi iki türlü yapılır: Değişken tanımlayarak ya da malloc() çağrısı ile. Bellek gözü tahsis edildiği zaman işaretçi bu bellek gözünü gösterebilir.

Belleği devasa bir byte (char) dizisi olarak düşünebilirsiniz. Bu dizi 2147483648 (veya daha büyük bir sayı) kadar elemana sahiptir. Genellikle, bu dizinin indisi hexadecimal olur. Diğer bir deyişle dizi indisleri 0x0'dan 0x7fffffff'ye kadardır. Bu sayı niçin önemlidir?

İşaretçi basitçe bu dizinin bir indisidir. Ne zaman bellekten x byte kadar yer tahsisi yapsak, bu bellek dizisinden x tane bitişik yer rezerve ederiz. Eğer bir işaretçiyi bu byte'lere kurarsak, işaretçi ayrılan byte'lerin ilkinin indeksi olacaktır.

Mesela, aşağıdaki programı inceleyiniz (p3.c'de):

```
main()
{
  int i;
  char j[14];
  int *ip;
  char *jp;

  ip = &i;
  jp = j;

  printf("ip = 0x%x.  jp = 0x%x\n", ip, jp);
}
```

Bu program bir integer (i), 14 karakterlik bir dizi (j) ve iki işaretçi (ip ve jp) için bellek tahsisi yapar. Sonra bu işaretçileri i ve j için tahsis edilen bellek gözüne işaret etmesi için kurar. Son olarak bu işaretçilerin değerlerini - bunlar bellek dizisinin indisleridir- yazdırır. Çalıştırdığımızda şunu görürüz:

```
UNIX> p3
ip = 0xefffe924.  jp = 0xefffe910
UNIX>
```

Bu çıktının anlamı: Biz belleği bir dizi olarak görürsek, 0xefffe924, 0xefffe925, 0xefffe926 ve 0xefffe927 elemanları i yerel değişkeni için tahsis edilmiştir, 0xefffe910 - 0xefffe91d arası da j dizisi için tahsis edilmiştir.

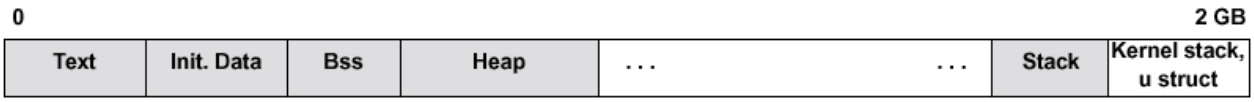
Dikkat ederseniz "jp=j" dedim, "jp= &j" demedim. Çünkü bir ifade olarak düşünüldüğünde, dizi bir işaretçiye eşdeğerdir. Tek farkı bir dizi değişkenine değer atanamamasıdır. Buna göre "jp=j" diyebilirsiniz ama "j=jp" diyemezsiniz. Ayrıca, bir dizi değişkeninin adresini alamazsınız. "&j" kullanımı illegaldir.

İşaretçiler bazı yönleriyle sayısal tipler gibidirler. İşaretçiler de global, yerel ya da fonksiyon parametresi

olarak tanımlanabilirler, onlara değer ataması yapılabilir, parametre olarak geçilebilirler, bir fonksiyondan dönebilirler. Bizim makinalarımızda bütün işaretçiler 4 byte'dir. Böylece p3.c'de main() fonksiyonunda i için 4, j için 14, ip için 4 ve jp için 4 olmak üzere 26 byte'lik bellek gözü yerel değişkenler için tahsis edilmiştir.

Adres Alanı

Aşağıda bir programın adres alanının BSD stili organizasyonu görülüyor.



BSD segmentleri (diğerleri gibi) sanal alanda bitişik bölgeler olarak tanımlar. Bir proses adres alanı beş temel segmentten meydana gelir: (i) text segmenti, icra edilebilir kodu tutar; (ii) ilklendirilmiş veri segmenti, proses başlangıcında belirli sıfır-olmayan değerlerle ilklendirilmiş veriyi barındırır; (iii) bss segmenti, proses başlangıcında sıfır olarak ilklendirilmiş veriyi barındırır; (iv) yığıt segmenti, ilklendirilmemiş veriyi ve proses'in yığıtını barındırır; (v) yığın.

Ayrıca yığın çekirdeğin yığınını (mesela ilgili prosesin adına sistem çağrıları icra edilirken) ve kullanıcı yapısını -büyük miktarda prosese özgü bilgi tutan çekirdek veri yapısı- tutan bir bölgedir.

Bu BSD'nin proses başına tahsis edilen sanal bellek alanının gösterimidir. Text, ilklendirilmiş veri, bss ve yığıt en küçük adresden başlayarak bitişik bellek bölgelerine koyulurlar. Yığın ise en yüksek adres alanından başlar, adres alanında aşağı doğru artar. Yığıtın tepe noktası ile yığının taban noktası arasındaki boş bölge'den gerektiğinde tahsisat yapılır.

Nereye ve nasıl bellek alanı tahsis edileceği bu bellek tahsisatının bulunacağı segmenti etkiler.

Tip Dönüşümü (Bazen Tip Zorlama da denir)

Bazen belirli bir tipin x byte kadarını alıp başka bir tipin y byte'sine atamak istersiniz. Buna tip dönüşümü denir. Basit bir örnek olarak p4.c'deki gibi char'ı int'e ya da int'i float'a dönüştürelim.

```
main()
{
  char c;
  int i;
  float f;

  c = 'a';
  i = c;
  f = i;
  printf("c = %d (%c).   i = %d (%c).  f = %f\n", c, c, i, i, f);
}
```

"i=c" ifadesi ve "f=i" ifadesi tip dönüşümüdür.

Bazı tip dönüşümleri, yukarıdaki gibi, çok doğaldır. C derleyicisi bunu şikâyet etmeden yapacaktır. Çoğunlukla diğer durumlarda ise ne yaptığınızı söylemezseniz (Bu, derleyiciye "Evet, Ben ne yaptığımı biliyorum" demenin bir yoludur) C derleyicisi bu durumdan şikâyet edecektir.

Mesela, malloc(n) fonksiyon çağrısını düşünün. Yer tahsisini yapar ve programcıya n byte kadar bellek döndürür. p5.c'ye bakın:

```
main()
{
  char *s;

  s = malloc(10);
  strcpy(s, "Jim");
  printf("s = %s\n", s);
}
```

p5.c'yi derlemeyi denerseniz C derleyicisinden aşağıdaki uyarıyı alırsınız.

```
UNIX> gcc -o p5 p5.c
p5.c: In function `main':
p5.c:5: warning: assignment makes pointer from integer without a cast
UNIX>
```

Neler oluyor? Aksi belirtilmediği müddetçe C'de bütün fonksiyonların integer döndüreceği varsayılır. "s= malloc(10)" ifadesi s isimli işaretçiyi malloc'un dönen değeri -integer olduğu varsayılıyor- olarak kurmaya çalışıyor. Derleyici aslında p5'i oluşturur, fakat garip bir şey yaptığınızı -bir işaretçiyi integer'a atamak- da bilmenize izin verir.

Burada yapılacak uygun iş nedir? malloc()'u char * döndürecek şekilde p6.c'deki gibi tanımlayabilirsiniz.

```
extern char *malloc();

main()
{
  char *s;

  s = malloc(10);
  strcpy(s, "Jim");
  printf("s = %s\n", s);
}
```

Bu, derleyiciye herhangi bir yerlerde tanımlanmış char * döndüren bir malloc() fonksiyonunu kullandığınızı söyler. Göreceğiniz gibi p6.c herhangi bir uyarı vermeden derlenecektir, fakat çalıştırdığınızda p5 ve p6 aynı işi yapacaktır.

```
UNIX> p5
s = Jim
UNIX> p6
s = Jim
UNIX>
```

Bir çok insan p6.c'deki gibi kod yazmaz. Yerine, p7.c'deki gibi yazarlar:

```
main()
{
  char *s;

  s = (char *) malloc(10);
  strcpy(s, "Jim");
  printf("s = %s\n", s);
}
```

Bu, derleyiciye "Evet, Ben biliyorum ki malloc() int döndürür, ama ben ona char * gibi muamele edeceğim". Fark edeceğiniz gibi p7.c derlenir ve p5 ve p6 gibi çalışır.

Diğer konulara geçmeden önce burayı iyice anlamalısınız, ama p7.c hala sizin yazmanızı tavsiye ettiğim kod değildir. Aşağıdaki kodlar daha uygun sayılır.

```
#include < stdlib.h >

main()
{
  char *s;

  s = malloc(10);
  strcpy(s, "Jim");
  printf("s = %s\n", s);
}
```

Bu kod parçasında, stdlib.h dahil edilince main fonksiyonundan önce "void * malloc (size_t size);" kod satırı efektif bir şekilde eklenmiş olur. Bu, derleyiciyi malloc'un dönen değerinin gerçekte integer olmadığına, aslında void * işaretçisi olduğuna ikna eder. Nasıl ki bir karakter tipi açık dönüşümle integer'e daima dönüşebiliyorsa, benzer şekilde void * işaretçisi de char * tipinde olmaya daşma zorlanabilir.

Taşınabilirlik Meselesi

Farketmiş olmalısınız ki bizim makinalarımızda işaretçiler ve tamsayıların her ikisi de 4 byte boyutundadır. Bu birçok insanın işaretçilerle tamsayıların birbirlerinin yerine geçebildiğini düşünmesine sebep oluyor. Mesela, p8.c'nin kodlarına bakalım:

```
main()
{
  char s[4];
  int i;
  char *s2;

  strcpy(s, "Jim");
  i = (int) s;
  printf("i = %ld (0x%lx)\n", i, i);
  printf("s = %ld (0x%lx)\n", s, s);

  i++;
  s2 = (char *) i;
  printf("s = 0x%lx.  s2 = 0x%lx, i = 0x%lx, s[0] = %c, s[1] = %c, *s2 = %c\n",
        s, s2, i, s[0], s[1], *s2);
}
```

Bu kötü bir varsayımdır, ancak, bazı makinalarda, DEC alpha gibi, tamsayılar 4 byte ve işaretçiler 8 byte'dir. Bu sebeple, p8.c'yi alpha üzerinde çalıştırdığınızda çalışan bir program yerine bir hata görürsünüz. Çünkü "i=(int) s" dediğimizde s işaretçisinin 4 byte'sini kaybederiz. Sonra "s2 = (char *) i" dediğimizde s2'nin 4 fazladan byte'si sıfır olur, *s2 ve s[1] bize farklı adresler verirler. Gerçekte, s2 segment hatasına sebebiyet veren illegal bir adrese dönüşür:

sparcs'da:

```
UNIX> p8
i = -268441312 (0xefffe920)
s = -268441312 (0xefffe920)
s = 0xefffe920.  s2 = 0xefffe921, i = 0xefffe921, s[0] = J, s[1] = i, *s2 = i
```

alpha'da:

```
UNIX> p8
i = 536864720 (0x1fffe7d0)
s = 4831832016 (0x11fffe7d0)
Segmentation fault (core dumped)
```

Eğer i'yi int değil de long olarak tanımlarsak, alpha'da her şey güzel çalışır, çünkü hem long hem de işaretçiler 8 byte'dir.

alpha'da:

```
UNIX> p9
i = 4831832016 (0x11fffe7d0)
s = 4831832016 (0x11fffe7d0)
s = 0x11fffe7d0.  s2 = 0x11fffe7d1, i = 0x11fffe7d1, s[0] = J, s[1] = i, *s2 = i
UNIX>
```

Bu sınıfta biz işaretçilerle tam sayıların daima 4 byte olduğu bir makinada çalıştığımızı varsayacağız. Ancak, siz daima kodunuzun işaretçilerin ve tam sayıların farklı olduğu makinalarda da çalışacağından emin olmalısınız.

İşaretçi Aritmetiği ile Son Derece Dikkatli Olunuz

Yüksek seviyeli bir programlama dilini diğerlerinden ayıran bir özelliği de tiplere ve tip denetimine sahip olmaktır. Şunu deneyin:

```
main()
{
  char * s;
  int  array[2] = { 10000, 1000000};
  s = array;
  s++;
  printf("my array has %d and %d\n",array[0], (int)(*s));
  printf("array = %lx and s=%lx\n",array,s);
}
```

Çıktı aşağıdaki şekilde olacaktır:

```
my array has 10000 and 0
in address space, array = ffbef848, s= ffbef849
```

Fonksiyon İşaretçileri

Herşey devasa byte dizileri olduğu için (icra edilebilir metinler bile), herhangi bir fonksiyonun başlangıç adresini alabiliriz. Bununla birlikte, fonksiyon işaretçilerine hafiften giriş yapalım. Bu işaretçiler parametre olarak diğer fonksiyonlara geçilebilirler, atanabilirler, dizide tutulabilirler, diğer değişkenler gibi fonksiyondan dönebilirler.

Dönüş tipini ve her bir parametresini belirterek fonksiyon işaretçileri tiplerini de aşağıdaki şekilde tanımlayabilirsiniz.

```
  int (*) (void *, void *)
```

Bu kod satırı iki tane void işaretçi alan ve geriye bir tane tam sayı döndüren bir fonksiyon işaretçisidir. Şunu not alın, yukarıda bir tip tanımlandı. Şimdi aşağıya bakalım:

```
#include < stdio.h >

int i = 0;
int main(void)
{
  int (*myfunc) (void);
  int (*foo) (void);

  myfunc = main;
  printf("main = %lx, myfunc = %lx, i = %d\n",main,myfunc,i);
  i ++;
  if (i == 5) return 0;

  foo = (i%2) ? myfunc : main;
  foo();
}
```

Çıktı şu şekilde olacaktır:

```
main = 106d8, myfunc = 106d8, i = 0
main = 106d8, myfunc = 106d8, i = 1
main = 106d8, myfunc = 106d8, i = 2
main = 106d8, myfunc = 106d8, i = 3
main = 106d8, myfunc = 106d8, i = 4
```

Segmentasyon İhlalleri ve Bus Hataları

Bellek devasa bir byte dizisi olarak görülebilir. Ancak, bu dizinin belirli bölgelerine erişim yasaktır. Mesela 0 -

0x1000 arası elemanlara erişim izni yoktur. Erişilmesi yasak bir elemana erişmeye çalışırsanız, bir segmentasyon ihlali oluşturursunuz ve programınız siz hata ayıklama yapabilesiniz diye ana bellek dökümünü (core dump) verir. “Eleman 0 erişilemez” hatası yaygın bir hatadır ve sebebi ilklendirilmemiş işaretçidir. Bu durum oluşursa işaretçinin değeri sıfırdır ve onu dereferans etmek isterseniz, hatayı bulmanıza yardım edecek bir segmentasyon ihlali oluşturursunuz.

pa.c programı NULL'a dereferans yapmaya çalışarak segmentasyon ihlali oluşturmaktadır.

```
#include < stdio.h >

main()
{
  char *s;

  s = NULL;

  printf("%d\n", s[0]);
}
```

Bir sayısal tip tanımladınız zaman, onun değeri bellekte hizalanmalıdır (align). Başka bir deyişle, eğer tip 4 byte ise, onun bellekteki yeri de 4'ün katı olan bir bellek indeksi ile başlamalıdır. Mesela eğer i bir (int *) ise, ama bu i 4'ün katı değilse, bu adrese dereferans etmek hataya neden olacaktır. Bu hata bir bus hatası ve ana bellek dökümü ile ortaya çıkar. Mesela pb.c programı 1 numaralı bellek gözüne tam sayı olarak dereferans etmeye çalıştığı için hata ortaya çıkıyor. Bu bir bus hatası ortaya çıkaracaktır (Bus hatalarının ana bellek dökümüne karşı önceliği vardır). Bu arada, aşağıdaki programda int'i char ile değiştirirseniz, hizalanmak (aligning) char tipi için sorun olmamasına rağmen segmentasyon hatası alırsınız. Eğer donanım adres kod çözücüsünün yapısını bilirseniz, olmayan bir adres segmentine erişerek bus hatası fırlatması için onu tetikleyebilirsiniz. Fakat bu sisteme bağımlıdır ve bunu basit bir programda göstermek zordur. Detaylı bilgi için temel bilgilerdeki ders notlarına bakınız.

```
main()
{
  int *i;

  i = (int *) 1;

  printf("%d\n", *i);
}
```

Not ediniz ki, malloc() daima 8'in katı olan işaretçiler döndürür. Bununla birlikte bir sayısal tip için ya da sayısal tip dizisi için bellek tahsisi amacıyla malloc() kullanırsanız düzgün hizalanmış işaretçiler oluşturur.

Bir de, derleyici yapıları (struct) alanları ayarlanacak şekilde düzenler. Aşağıdaki yapıyı inceleyin:

```
struct {
  char b;
  int i;
}
```

Burada bütün yapı 8 byte olacaktır. b için 1 byte, kullanılmayan 3 byte ve i için 4 byte. Hizalanabilmesi için boş bırakılan 3 byte gereklidir. Derleyici alanlar arasında belleğe rahat yerleşsinler diye yer değiştirme yapmaz. Mesela, şu şekilde bir yapı olsun:

```
struct {
  char b1;
  int i1;
  char b2;
  int i2;
}
```

Yapının tümü 16 byte olacaktır:

- 1 byte: b1
- 3 byte kullanılmadı

- 4 byte: i1
- 1 byte: b2
- 3 byte kullanımadı
- 4 byte: i2

Ancak, sıralarını farklı yaparsanız, bütün bu alanları 12 byte'ye yerleştirebilirsiniz:

```
struct {
  char b1;
  char b2;
  int i1;
  int i2;
}
```

Şimdi yapı şu şekilde olacaktır:
- 1 byte: b1
- 1 byte: b2
- 2 byte kullanılmadı
- 4 byte: i1
- 4 byte: i2

Üç Yaygın Tip Hatası

İlki biraz saçma görünüyor, fakat bütün tip hatalarının temelinde bu vardır. pc.c'ye bakınız:

```
main()
{
  char c;
  int i;
  int j;

  i = 10000;
  c = i;
  j = c;

  printf("I: %d,   J: %d,       C: %d\n", i, j, c);
  printf("I: 0x%04x,  J: 0x%04x,   C: 0x%04x\n", i, j, c);
}
```

c değişkeni char tipinde olduğu için 10000 değerini tutamaz. Onun yerine i'nin en düşük sıralı byte'sini tutar, o da 16'dır (0x10). Sonra j'yi c'ye (set) kurarsanız, göreceksiniz ki j 16 oluyor. Bu hatayı anladığınızdan emin olun.

İkinci hata math rutinleri ile uğraşırken karşılaşacağınız tipik bir hatadır. "man log10" derseniz göreceksiniz ki log10 double alır ve double döndürür:

```
    double log10(double x);
```

Öyleyse pd.c 100.0 sayısının logaritmasını alır, sonuç 2.00 olacaktır.

```
main()
{
  double x;

  x = log10(100);

  printf("%lf\n", x);
}
```

Derlediğiniz zaman, math kütüphanelerini eklemesi için bağlama satırına -lm ilave etmelisiniz. Bunu

yaptığınız zaman, tuhaf bir sonuç göreceksiniz:

```
UNIX> pd
-1035.000000
```

Niçin? Çünkü C programınıza math.h başlık dosyasını eklemediniz ve bu nedenle derleyici log10'a parametre olarak tamsayı geçtiğinizi düşünecektir ve tamsayı döndürecektir. Derleyici ayrıca int'i double'a dönüştürmeyi de düşünmeyecektir. Öyleyse hatayı buldunuz. C'nin bağlayıcısı (linker) parametre tipleri ile genelde ilgilenmez, aslında sadece fonksiyon isminin eşleşip eşleşmediğini denetler. C++'da olduğu gibi fonksiyon isimlerinin aşırı yüklenememesinin asıl sebebi budur. Şunu da not alın: Derleyici, fonksiyon tanımı verilirse, geçtiğiniz parametre sayısına bakar. Böylece biraz güvenlik eklenmiş oldu. Yukarıdaki hatayı pe.c programınıza math.h ekleyerek düzeltebilirsiniz, ayrıca bu şekilde c derleyicisi sizin için tip dönüşümü (burada integer'dan double'a) de yapacaktır. Diğer bazı diller tip denetimi için daha güçlü mekanizmalara sahiptirler.

```
#include < math.h >

main()
{
  double x;

  x = log10(100);

  printf("%lf\n", x);
}
```

```
UNIX> pd
2.00000
```

Aslında aşağıdaki program da derlenecektir:

```
#include < stdio.h >
main ()
{
  double x;
  x = log10();
  printf("x = %lf\n",x);
}
```

Burada log10 fonksiyonuna herhangi bir parametre geçilmiyor. Fakat derleme zamanında bağlayıcıya "-lm" bayrağını sağlarsanız, derlenecektir. Programın çıktısı ise aşağıdaki gibi olacaktır:

```
x = -1036.000000
```

Ama mat.h başlık dosyası eklendiğinde ise:

```
#include < stdio.h >
#include < math.h >
main ()
{
  double x;
  x = log10();
  printf("x = %lf\n",x);
}
```

Bu sefer derleyici hata verecektir:

```
prototype mismatch: 0 args passed, 1 expected
```

Özetle: Neler olup bittiğini anlamamışsanız, derleyici ve bağlayıcı garip şeyler yapacaklardır. Bazı tuhaf hataların ortaya çıkarılması çok zaman alacaktır. Aslında, burada konuştuğumuz üç yaygın hata da tip dönüşümü ile ilgilidir, daha açık belirtmek gerekirse açık tip dönüşümünün tetiklenip tetiklenmemesi ile ilgidir.

Çok dikkatli olunuz.

son olarak pf.c diğer bir yaygın hatayı gösteriyor:

```
main()
{
  double x;
  int y;
  int z;

  x = 4000.0;
  y = 20;
  z = -17;

  printf("%d %d %d\n", x, y, z);
  printf("%f %d %d\n", x, y, z);
  printf("%lf %d %d\n", x, y, z);
  printf("%lf %lf %lf\n", x, y, z);
}
```

```
UNIX> pf
1085227008 0 20
4000.000000 20 -17
4000.000000 20 -17
4000.000000 0.000000 0.000000
```

Birinci satırdaki tipik hatayı görebiliyorsunuz. Bir double'yi integer olarak yazdırmaya çalışıyorsunuz. Sadece değerini yanlış almakla kalmaz, x ve y'yi de yanlış alır. Sebebini daha sonra öğreneceksiniz. İkinci ve üçüncü satırlar iyi, fakat dördüncü satır yanlış, çünkü her üç miktarı da double olarak yazdırmaya çalışıyorsunuz. Yine, sebebini sonra göreceksiniz, fakat şimdilik bu tip hataların farkında olmalısınız, çünkü tekrar karşınıza çıkabilir.

Aşağıdaki pg.c programında daha ilginç bir durum ortaya çıkıyor.

```
#include < stdio.h >

main()
{
  double x;
  int y;
  char s[4] = "Jim";

  x = 4000.0;
  y = 20;

  printf("%lf %lf %s\n", x, y, s);
}
```

Bu kodu çalıştırırsanızi bazen aşağıdaki çıktıyı verecektir:

```
UNIX> pg
Bus error
```

Fakat, bazen de böyle bir şey görebilirsiniz:

```
4000.000000 0.00000
```

Ya da böyle:

```
4000.000000 0.00000 ?Š???‡%??Š???‡&H?Š?¡?‡'H?Š???‡(H?Š?□?‡)H?Š
```

Bu tip birçok hatayı yakalamanın bir yolu da derleyicinin uyarı seviyesini en yükseğe getirmektir ve her bir uyarıya UYARI gözüyle bakmaktır. Bu durumda her derleyici size yardımcı olacaktır.