Sistem Programlama

DR. ÖĞR. ÜYESİ ABDULLAH SEVİN

İçerik

a) .h dosyaları

b) extern ve static anahtar kelimesinin anlamı

c) derleme vs bağlama

d) bir prosesin hafızada nasıl organize edildiği (yığın, yığıt, vb.)

e) makefiles

- http://web.eecs.utk.edu/~huangj/cs302/notes/Some%20fundamentals.htm
- http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/CStuff-1/lecture.html

Program nedir?

- Elbette işletim sistemlerinde bir program çalışırken hepimiz buna <proses> dendiğini biliyoruz.
- Programın kendisi birçok biçimde olabilir, yani makine kodu, assembly kodu, C kodu, C++ veya Fortran, Java, …
- İşlemciler YALNIZCA **makine kodunu** yürütür. Makine kodu, işlem kodları (Opcode) ve işlenenlerden (Operands) oluşan ikili makine sözcükleriyle yazılır.
- **İşlem kodları**, örneğin bir aritmetik toplama gibi farklı talimatları temsil eder. **İşlenenler** komut tarafından çalıştırılır. Bir makine kodu şöyle görünebilir:

1000100011010100101010110101011

- Çoğu aklı başında insan bu formatta kod yazmak istemez. Bu yüzden assembly dili gelişti;

```
ADD AX, BX
```

Assembly

- Assembly dilindeki anımsatıcılar, **işlemci bağımlı** olan makine kodlarıyla birebir ilişkilidir. Fakat hala dönüşüm için «**assembler»** gerekli!
- Intel x86'dan Motorola 68k'ye geçerseniz, yeni bir program yazılmalıdır.
- İşte Motorola 68k üzerinde çalışan gerçek bir assembly programı:

```
ORG     $1000
N       EQU     5
CNT1    DC.B    0
CNT2    DC.B    0
ARRAY   DC.B    2,7,1,6,3

        ORG     $1500
MAIN    LEA     ARRAY,A2
        MOVE.B  #N,D1
        CLR     D6
        CLR     D7
        JSR     SORT
        STOP    #$2700
```

- Bu arada, C programınızın Assembly takmaya nasıl benzediğini görmek istiyorsanız, “cc –S xxx.c” veya “gcc –S xxx.c” kullanmayı deneyin. Bu komutlar, C kodunuzun derleme sürümünü içeren .s dosyaları üretir.

Assembly

- Yukarıdaki programda, **adresler,** bellek bankalarınızdaki ayrı baytlara karşılık gelen fiziksel adreslerdir.
- Bu adresleri sabit kodladıysanız, işlemcinizde her seferinde yalnızca bir program çalıştırsanız iyi olur!
- Assembly dilinde programlama yapmak sıkıcı olabilir! Sistem çağrıları yapmıyorsanız, konsolunuza bir karakter koymak için 30 satır kod gerekebilir.
- Kendi kod parçanızı her yeniden kullanmak istediğinizde uğraşmanız gerekli. Bütün yapıları siz kuruyorsunuz, …… **Başkalarının kodunu** kullandığınızı hayal edin.
- Bu gibi problemleri aşmak için C/C++ gibi diller geliştirildi.

C dili

- Kaynak kodunuz bir **derleyici** ile çalıştırılır ve C programınızı ikili makine koduna çevirir ve genellikle 'nesne-object' olarak adlandırılan bir modül oluşturur.
- Derleyicinin ortaya çıkardığı her object dosyasının içinde, temelde platforma bağlı bir biçimde bir makine kodu modülü elde edersiniz.
- Ancak, önemli nokta; orada **kodlanmış fiziksel adres yok**. Bu işlem, modülü ana bellekte **yeniden yerleştirilebilir hale** getirir.
- Bu durumda xxx.o adlı object dosyası diske yazılır.

C dili

- daha sonra, dış referanslar (external ref.), fonksiyon çağrıları veya değişkenler için nesne dosyanıza bakmak üzere ayrı bir **bağlayıcı-linker** program çağrılır.
- Örneğin, “printf(“Merhaba Dünya\n”);” fonk. çağırdınız. Derleyiciniz, sistem kitaplığında sağlandığı için bu çağrının nasıl çalıştığını bilmiyordur.
- Sistem kitaplığından bir printf makine kodu modülünü bulup xxx.o ile bir araya getirmek **bağlayıcınızın işidir.**
- Bağlama oturumunuzun sonucu, diskte yürütülebilir-executable bir dosyadır.

C dili

- komut satırına a.out yazdıktan sonra, işletim sistemindeki **yükleyici-loader** a.out'u diskten okur ve ana belleğe yazar.
- Bu sırada a.out'taki her satırın fiziksel adresi belirlenir.
- Ardından, işletim sistemi programınızın giriş noktasını (xxx.c'deki main fonksiyon) bulur ve oradan çalıştırmaya başlar.

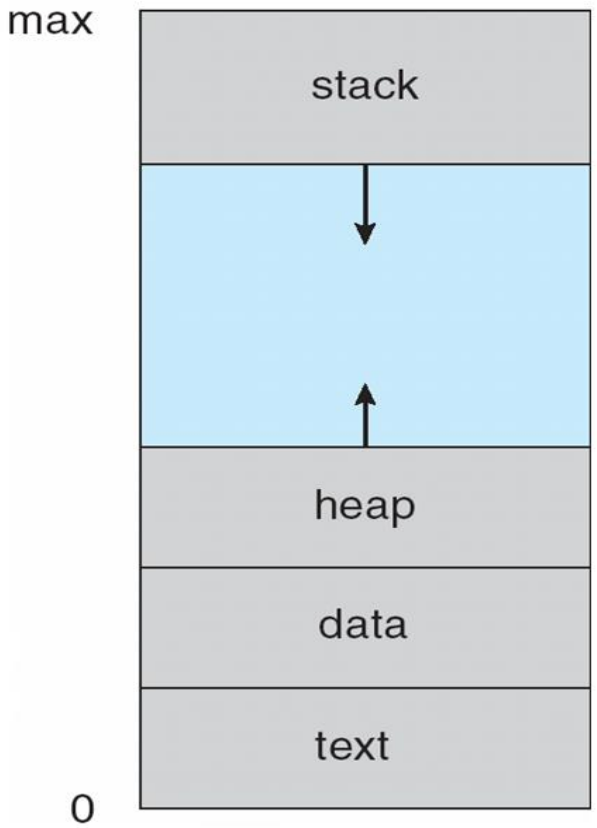
Neden .h dosyasına ihtiyacımız var?

- İşletim sistemindeki **kütüphaneleri-library** ve bazen de başkaları tarafından sağlanan kütüphaneleri kullanarak ciddi bir uygulama geliştirmek için birçok kişi gerekir.
- Modüler geliştirme bir normdur.
- Derleyicilerin tip kontrolleri yapması ve kütüphaneye ve diğer kişilerin fonksiyonlarına yaptığınız çağrıların doğru olduğundan emin olması gerekir.
- Çağırdığınız fonksiyon gövdesinin C kaynağına erişiminiz olsun ya da olmasın, **kodunuza bir fonksiyon gövdesi yapıştırmak kötü bir fikirdir** (neden?).
- Genellikle **veri türlerini, fonksiyon bildirimlerini ve makroları** içeren .h dosyalarının kullanılması bu sorunu çözer.

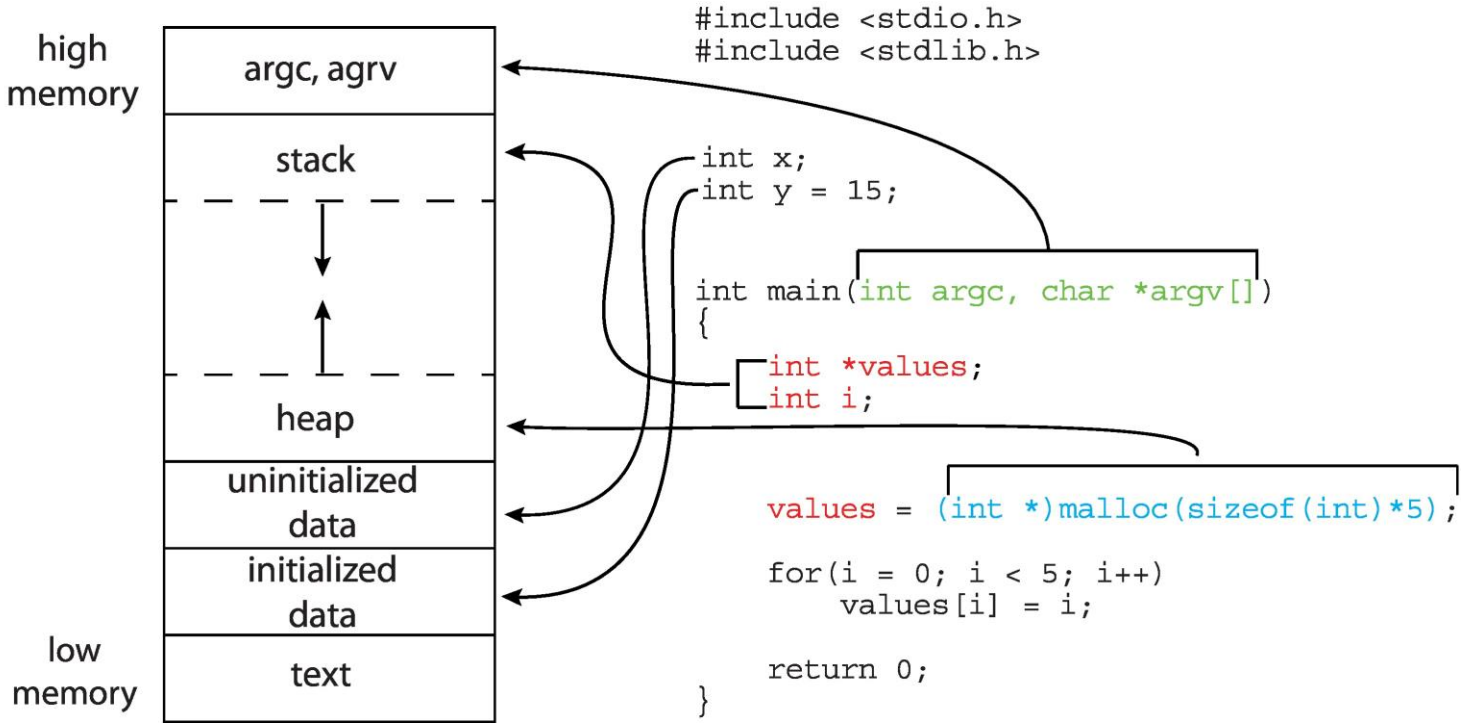
Prosesin hafızadaki yerleşimi

Bölümleri;

- **program kodu** (text section),
- **Mevcut aktiviteler**, **program sayacı** ve diğer kaydedicileri içeren mevcut faaliyetler.
- **Yığın** (stack) geçici veriyi içerir.
  - Fonksiyon parametreleri, geri dönüş değerleri, yerel değişkenler
- **Veri bölümü** (data section) global değişkenleri içerir.
- **Bellek kümesi-Yığıt** (heap) çalışma zamanında dinamik olarak tahsis edilmiş belleği içerir.



Memory Layout of a C Program



Fonksiyonlar ve Yığın-Stack bölümü

1. OS, ana programdaki geçerli adresi yığına gönderir. **(Geri dönülecek adres)**

2. CPU kayıtları da dahil olmak üzere ana programı çalıştırırken kurtarmak için gerekli tüm verileri yığına gönderir.

3. bu fonksiyon çağrısındaki tüm bağımsız değişkenleri birer birer yığına gönderir.

4. o fonksiyonun text (kod) bölümünün baş adresini bulur ve yürütmeye başlar

5. fonksiyon içinde, yerel değişkenler için yığında daha fazla bellek alanı ayrılır

6. malloc() çağrıları varsa, yığın alanından bellek ayrılır

Fonksiyonlar ve Yığın-Stack bölümü

7. fonksiyonda hesaplamayı bitirin
8. dinamik bellek ayırmaları varsa, belki de yer ayırmanız gerekir?
9. tüm yerel değişkenleri yığından çıkarın
10. tüm fonksiyon bağımsız değişkenlerini yığından çıkarın
11. orijinal çalışma durumunu yığından çekin
12. yığından yeni alınan adresten tekrar yürütmeye başlayın

Kapsam ve Görünürlük

- Dosya kapsamı-File scope;
- Dosya kapsamına sahip tanımlayıcı adları genellikle "**genel-global**" veya "**harici-external**" olarak adlandırılır.
- Herhangi bir blok veya parametre listesinin dışında görünür ve bildiriminden sonra herhangi bir yerden erişilebilir.

- Fonksiyon Kapsamı-Function scope;
- Bir fonksiyon içerisinde tanımlıdır ve Etiket adları, bir fonksiyon içinde benzersiz olmalıdır.

Kapsam ve Görünürlük

- Blok kapsamı-Block scope;
- Yalnızca bildirimi veya tanımı noktasından bildirimi veya tanımı içeren bloğun sonuna kadar görünür.
- Kapsamı, o blokla ve o blokta iç içe geçmiş herhangi bir blokla sınırlıdır ve ilişkili bloğu kapatan kaşlı ayraçta biter. Bu tür tanımlayıcılara bazen "**yerel değişkenler**" denir.
- Fonksiyon Prototipi-Function-prototype scope;
- Bir fonksiyon prototipindeki parametre bildirimleri listesinde görünür.

Extern ve static

- **static** ile bildirilen bir değişken. Statik değişkeni açıkça sabit bir ifadeyle başlatabilirsiniz. Başlatıcıyı atlarsanız, değişken varsayılan olarak 0 olarak başlatılır.

```
static int k = 16;
static int k;
```

Extern ve static

- **static** belirleyicisini lokal değişkenlerle kullanabiliriz.
- Normal olarak, içinde lokal değişken tanımlanan bir fonksiyonu her çağırdığımızda, lokal değişken değeri yenilenir.
- Ancak, bu lokal değişkeni static olarak tanımlarsak, fonksiyonu her çağırmamızda, lokal değişken bir önceki fonksiyon çağrısındaki en son değerini korur.
- Sonuç olarak, static lokal bir değişkene sadece fonksiyonun ilk çağrılışında bir defaya mahsus olmak üzere değer verebiliriz.

Extern ve static

```c
#include <stdio.h>

void fonk(void);
void fonk_sta(void);

int main(void)
{
fonk();
fonk_sta();
printf("\n");
fonk();
fonk_sta();
 return 0;
}
```

```c
void fonk(void) {
int id = 1;
printf("fonk() id değişken değeri: %d\n", id);
id = id + 21;
printf("fonk() id değişken değeri: %d\n", id);
}
void fonk_sta(void) {
static int id = 1;
// Sadece fonksiyonun ilk çağrısında çalışır.
printf("fonk_sta() id değişken değeri: %d\n", id);
id = id + 21;
printf("fonk_sta() id değişken değeri: %d\n", id);
}
```

Extern ve static

```
fonk() id değişken değeri: 1
fonk() id değişken değeri: 22
fonk_sta() id değişken değeri: 1
fonk_sta() id değişken değeri: 22
fonk() id değişken değeri: 1
fonk() id değişken değeri: 22
 fonk_sta() id değişken değeri: 22
fonk_sta() id değişken değeri: 43
```

Extern ve static

- **static** belirleyicisini global değişkenlerle de kullanabiliriz.
- **static** global bir değişken tanımladığımızda, bu değişkeni sadece içinde tanımlandığı dosyada bulunan fonksiyonlar kullanabilir.
- Bunun yanında, aynı programa ait farklı dosyalarda bulunan ve aynı isme sahip biri normal diğeri de static olan iki global değişken tanımlayabiliriz.

Extern ve static

```c
// deneme1.c
#include <stdio.h>
void fonk1(void);
void fonk2(void);
static int gid = 21; // Static global int değişken bildirimi
int main(void) {
fonk1();
fonk2();
return 0; }
void fonk1(void)
{ printf("deneme1.c gid değişken değeri: %d\n", gid); }
```

```c
// deneme2.cpp
#include <iostream>
using namespace std;
int gid = 35; // global int değişken bildirimi
void fonk2(void)
{
printf("deneme2.c gid değişken değeri: %d", gid);
}
```

```
deneme1.c gid değişken değeri: 21
deneme2.c gid değişken değeri: 35
```

Extern

Bir değişkeninin başında **extern** ifadesini kullanmak, o değişkenin projede yer alan kod dosyalarının birinde tanımlandığını ve o değişkene extern ifadesini kullandığımız dosya içinden erişim sağlamak istediğimizi gösterir.

```c
// deneme1.c
#include <stdio.h>
void fonk(void);
int gid = 287; // global int değişken tanımlaması
int main(void) {
printf("deneme1.c gid değişken değeri: %d\n", gid);
fonk(); // deneme2.c dosyasındaki fonk() fonksiyonuna çağrı
return 0;
}
// deneme2.c
extern int gid; // global int değişken bildirimi
void fonk(void)
{ printf("deneme2.c gid değişken değeri: %d", gid); }
```

```
deneme1.c gid değişken değeri: 287
deneme2.c gid değişken değeri: 287
```

Lifetime

- “Yaşam süresi”, bir değişkenin veya fonksiyonun var olduğu bir programın yürütülmesi sırasındaki periyottur.
- Tanımlayıcının depolanma süresi, statik süre (global yaşam süresi) veya otomatik süre (yerel yaşam süresi) olarak yaşam süresini belirler.
- Statik depolama sınıfı belirticisi olmadan bildirilen bir tanımlayıcı, bir fonksiyon veya blok içinde bildirilmişse, otomatik depolama süresine sahiptir.
- **Global yaşam süresi**: Tüm fonksiyonların global ömrü vardır. Bu nedenle, program yürütme sırasında her zaman var olurlar.
- **Yerel yaşam süresi**: Yerel bir değişkenin bir başlatıcısı varsa, değişken her oluşturulduğunda başlatılır (statik olarak bildirilmedikçe).

Make

Header files

- C++'da olduğu gibi, standart başlık dosyalarını #include ile ekleriz.
- Küçüktür/büyüktür işaretlerine dosya adını ve .h uzantısını ekleriz.

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

- C' de ise;

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

C’de veri tipleri

- C'de değişkenlerin sahip olabileceği üç tür tür vardır –
- skalerler, toplamlar/eklemeli ve işaretçiler (scalars, aggregates, and pointers)
- Skaler tipler;
  - char -- 1 byte
  - short -- 2 bytes
  - int -- 4 bytes
  - long -- 4 or 8 bytes, depending on the system and compiler
  - float -- 4 bytes
  - double -- 8 bytes
  - (pointer -- 4 or 8 bytes, depending on the system and compiler)

C’de veri tipleri

- C'de bir türün boyutunu doğrulamak veya kullanmak istiyorsanız, sizeof() makrosunu kullanırsınız.
- Örneğin, sizeof(long), sisteminizde bir long'un ne kadar büyük olduğuna bağlı olarak 4 veya 8 değerini döndürür.

```
#include <stdio.h>
 #include <stdlib.h>
 int i;
 int main(int argc, char **argv) {
int j; /* Copy argc to j to i and print i */
j = argc;
 i = j;
printf("Argc: %d\n", i); /* Print the size of a long. */
 j = sizeof(long);
printf("Sizeof(long): %d\n", j); /* Print the size of a pointer. */
 j = sizeof(int *);
 printf("Sizeof(int *): %d\n", j); return 0; }
```

```
UNIX> bin/p1
Argc: 1
 Sizeof(long): 8
Sizeof(int *): 8
UNIX> bin/p1 using many arguments
Argc: 4
Sizeof(long): 8
Sizeof(int *): 8
UNIX>
```

C’de veri tipleri

- Bazı makineler, işaretçileri ve uzunlukları dört bayt olmaya zorlayan 32 bit modunda derlemenize izin verir;

```
UNIX> gcc -m32 -o bin/p1-32 src/p1.c
UNIX> bin/p1-32
Argc: 1
Sizeof(long): 4
Sizeof(int *): 4
UNIX>
```

C’de veri tipleri

- Aggregate (Küme-Eklemeli tipler);
- **Diziler ve struct’lar,** C'deki küme türleridir.
- Skaler’lerden daha karmaşıktırlar. Bir diziyi statik olarak global veya yerel bir değişken olarak bildirebilirsiniz.

```
#include <stdio.h>
#include <stdlib.h>
char s1[15];
int main(int argc, char **argv) {
char s2[4];
 ...
```

C’de veri tipleri

- Bir dizi statik olarak tanımlanmışsa, onu başka bir diziye atayamazsınız.
- ``s2 = "Jim"'' ifadesi C'de geçersiz çünkü s2 statik olarak bildirildi. Bu programı derlemeye çalışırsanız, gcc size bir hata verecektir:

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

char s1[15];

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

  s2 = "Jim";         // This line will not compile.
  return 0;
}
```

```
UNIX> gcc -o bin/p2 src/p2.c
src/p2.c:16:6: error: array type 'char [4]' is not assignable
s2 = "Jim"; // This line will not compile.
~~ ^
1 error generated.
UNIX>
```

Pointer

- İşaretçi-Pointer aslında bir sayıdır, bellekteki bir adresi temsil eden bir tam sayıdır.
- Bir yandan, birçok programlama dilinde işaretçi kavramı yoktur.
- Ancak C'nin en büyük avantajlarından biri, yani esneklik, işaretçi kullanımından gelir.
- Öte yandan dizi, çoğu programlama dilinin sahip olduğu bir kavramdır.
- C'de de aynı şeye sahibiz.

```
int myarray [30];
```

- **myarray** değişkeni gerçekten (int *) türünde bir işaretçidir. C'de dizi öğelerine işaretçiler kullanılarak erişilir:

`myarray[5]` aslında `*(myarray + 5)` olarak çevrilebilir

Pointer

- İşaretçiyi de işaret edebiliriz;

```
int myarray[30];
main()
{
     int * myptr;
     int ** mydblptr;
     myptr = myarray;
     mydblptr = &myarray;
}
```

- Bss segmentinde de tahsis edilmiş 30 tamsayı vardır. Bss'deki bu 120 bayt, 30 tamsayınızı gerçekten koyduğunuz yerdir ve myarray, bu 120 baytın baş adresine eşittir.

Pointer

- İşaretçiyi de işaret edebiliriz;

```
int myarray[30];
main()
{
     int * myptr;
     int ** mydblptr;
     myptr = myarray;
     mydblptr = &myarray;
}
```

- Bss segmentinde de tahsis edilmiş 30 tamsayı vardır. Bss'deki bu 120 bayt, 30 tamsayınızı gerçekten koyduğunuz yerdir ve myarray, bu 120 baytın baş adresine eşittir.

Örnek

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

/* This sets all lower-case letters in a to upper case. */

void change_case(char a[20])
{
  int i;

  for (i = 0; a[i] != '\0'; i++) {
    if (a[i] >= 'a' && a[i] <= 'z') a[i] += ('A' - 'a');
  }
}

/* This initializes a 19-character string of lower-case letters, and then calls change_case(). */

int main()
{
  int i;
  char s[20];

  /* Set s to "abcdefghijklmnopqrs". */

  for (i = 0; i < 19; i++) s[i] = 'a' + i;
  s[19] = '\0';

  /* Print, call change_case() and print again. */

  printf("First, S is %s.\n", s);
  change_case(s);
  printf("Now, S is   %s.\n", s);

  return 0;
}
```

- **<u>Dizileri parametre olarak ilettiğinizde, diziler değil işaretçiler iletilir.</u>**
- A dizisi 20 karakterlik bir dizi olarak bildirseniz de, yordama iletilen işaretçinin yalnızca işaretçi olduğunu fark edeceksiniz.
- Bu nedenle change_case() bir kopya üzerinde değil dizi üzerinde çalışır:

```
UNIX> bin/p2a First, S is abcdefghijklmnopqrs.
                    Now, S is ABCDEFGHIJKLMNOPQRS.
UNIX>
```