



# ITU ACM Student Chapter Course Program

## Introduction to C

### Week 6

#### Instructor

Gökalp Akartepe

#### Prepared by

Mehmet Yiğit Balık & Mihriban Nur Koçak & Emir Oğuz

## Static ve Dinamik Olarak Hafızada Yer Ayırma (Memory Allocation)

Kod içerisinde oluşturulan değişkenler işletim sisteminde belirli bir yer kaplar. Bu yerin boyutu kimi zaman belirli yani değiştirilemez iken kimi zaman ise kullanıcının program esnasında gireceği verilere göre değişebilecek durumdadır. Temel olarak bu farklılıktan dolayı verinin hafızada tutulduğu iki farklı bölge vardır. Verileri tutarken eğer program esnasında boyutları bildirilmiş değişmez bir yapı kullanılıyorsa **stack** bölgesinin, değiştirilebilir bir yapı kullanılıyorsa **heap** bölgesinin kullanılması gerekecektir. **Stack** ve **heap** kullanımları farklı ve dikkat edilmesi gereken bir konudur. **Stack** kullanılır, program sonlanana kadar boyutu değiştirilemez ve işi bittikten sonra kendini otomatik olarak bellekten yok eder. Fakat **heap**'te program sırasında boyut değiştirilebilir ve program bitmeden kullanıcı tarafından bellekten yok edilmelidir çünkü bu işi kendisi yapmaz.

### Stack:

- Oluşturulan değişkenler **stack** kapsamından çıkınca otomatik olarak yok edilir.
- Ulaşılması **heap**'e göre oldukça hızlıdır.
- **Stack** üzerinde kullanım fazla olduğunda alan yeterli olmayabilir. Örneğin 20 boyutlu bir diziye 21 eleman atamak gibi.
- Oluşturulan değişkenler **pointer** olmadan kullanılabilir.
- Derleme zamanında oluşturulur.
- Kullanılacak yerin boyutu tam olarak biliniyorsa **Stack** kullanılmalıdır.

### Heap:

- Bir blok içerisinde oluşturulan ve **heap**'de tutulan değişkenler, değişkenlerin bulunduğu kod bloğunun dışına çıktığında otomatik olarak yok edilemezler, bunun manuel olarak yapılması gerekir.
- **Stack** ile karşılaştırıldığında oldukça yavaştır.
- Doğru kullanılmaması durumunda bellek sorunları yaratır.
- Değişkenler **pointer** ile kullanılır.
- Çalışma zamanında oluşturulur.
- İhtiyaç duyulan boyut tam olarak bilinmiyorsa **Heap** kullanımı biçilmiş bir kaftandır.

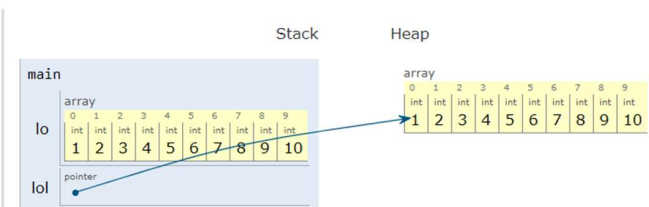
Verileri tutmak adına hafızada yer ayırmak için kullanılan iki farklı hafızada yer ayırma yöntemi vardır:

- **Statik** olarak hafızada yer ayırma (**Static Memory Allocation**)
  - Bu yöntemle değişkenlere kalıcı bir yer ayrılır.
  - Yer ayırma işlemi program başlamadan önce tamamlanmış olur.
  - Veriler **stack**'de depolanır.
  - Daha az verimli bir yöntemdir.
  - Bu yöntemde hafızanın yeniden kullanılabilirliği yoktur.
- **Dinamik** olarak hafızada yer ayırma (**Dynamic Memory Allocation**)
  - Bu yöntemle değişkenlere program aktif olduğunda yer ayırma işlemi gerçekleştirilir.
  - Yer ayırma işlemi program çalışırken başlar ve sona erer.
  - Veriler **heap**'te depolanır.
  - Daha verimli bir yöntemdir.
  - Bu yöntemle hafıza tekrar tekrar kullanılabilir ya da serbest bırakılabilir.

Şu ana kadar kullanılan diziler statik olarak tanımlandığı için sabit sayıda eleman içiyorlardı. Fakat diziler sabit sayıda değerlerden oluşabileceği gibi dinamik olarak tanımlanırlarsa bu sabitlik durumu ortadan kalkabilir. Dinamik olarak hafızada yer ayırmak için ise **malloc()** fonksiyonu kullanılır. Bu fonksiyon sayesinde bir pointer'ın tutabileceği veri boyutu dinamik olarak belirlenmiş olur. Bu boyut programın çalışması esnasında **realloc()** fonksiyonu ile değiştirilebilir. Dinamik olarak hafızada yer ayrılırken dikkat edilmesi gereken en önemli nokta ise program sonunda **free()** fonksiyonunu kullanarak hafızada ayrılmış yerin serbest bırakılması gerektiğidir. Bu serbest bırakma işlemi yapılmadığında hafızada **leak** adı verilen problemler oluşur ve bu problemler RAM'in verimsiz kullanılmasına neden olur.

```
C (gcc 4.8, C11)
(known limitations)

1 #include <stdio.h>
2
3 int main() {
4     int lol[10] = {1,2,3,4,5,6,7,8,9,10};
5     int * lol = (int *) malloc(10 * sizeof(int));
6     for(int i = 0; i <= 9; i++){
7         lol[i] = i + 1;
8     }
9
10    return 0;
11 }
```



**malloc() ,free() ve realloc() fonksiyonlarının kullanımı**

## malloc()

- Yazı düzeni (syntax):
  - `veri tipi* pointer = (veri tipi*) malloc(size);`
  - `int* ptr = (int*) malloc(100 * sizeof(int));`
- **sizeof()** fonksiyonu içine yazılan veri tipinin boyutunu döndürür.
  - Önceki derslerde sabit boyutları olduğu öğrenilmiş veri tiplerinin boyutları bu fonksiyon kullanılarak da rahatlıkla gözlemlenebilir.

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

int main()
{
    printf("size of int: %ld\n", sizeof(int));
    printf("size of float: %ld\n", sizeof(float));
    printf("size of double: %ld\n", sizeof(double));
    printf("size of char: %ld\n", sizeof(char));
    return EXIT_SUCCESS;
}
```

```
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07# gcc -std=c99 -Wall -Werror week07.c -o week07
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07# ./week07
size of int: 4
size of float: 4
size of double: 8
size of char: 1
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07#
```

## free()

- Yazı düzeni (syntax):

- `free(pointerın adı);`
- **malloc()** fonksiyonu ile tahsis edilen belleği yeniden kullanabilmek üzere serbest bırakır.
- Bu fonksiyon kullanıldıktan sonra, bellek alanı serbest bırakılan pointer'ın kesinlikle **NULL pointer**'a eşitlenmesi gerekir.
  - **NULL** değerine sahip olan bir pointer hiçbir adres ifade etmez

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

int main()
{
    int size;

    printf("Lutfen dizinin buyuklugunu giriniz: ");
    scanf("%d", &size);
    int *dizi = (int *)malloc(size * sizeof(int));
    int toplam = 0;
    for (int i = 0; i < size; i++)
    {
        printf("Lutfen %d. degeri giriniz: ", i + 1);
        scanf("%d", dizi + i);
        toplam += *(dizi + i);
    }
    printf("dizi'nin elemanlari toplami: %d\n", toplam);
    free(dizi);
    dizi = NULL;
    return EXIT_SUCCESS;
}
```

```
root@MSI: /mnt/c/Users/mybal/Desktop/C course/weeks/week07
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07# gcc -std=c99 -Wall -Werror week07.c -o week07
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07# ./week07
Lutfen dizinin buyuklugunu giriniz: 4
Lutfen 1. degeri giriniz: 1
Lutfen 2. degeri giriniz: 2
Lutfen 3. degeri giriniz: 3
Lutfen 4. degeri giriniz: 4
dizi'nin elemanlari toplami: 10
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07#
```

## realloc()

- Yazı düzeni (syntax):
  - `pointer = realloc(pointer, new size);`
- **malloc()** fonksiyonu ile tahsis edilen belleğin boyutunu değiştirir. Belleğin boyutu arttırılırsa, tahsis edilen önceki bellek içeriği değişmeden kalır ve eklenen bellek içeriğine herhangi bir değer atanmaz. Belleğin boyutu azaltılırsa veri kaybına yol açılabilir.

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

int main()
{
    int size;
    printf("Lutfen dizinin buyuklugunu giriniz: ");
    scanf("%d", &size);
    int *dizi = (int *)malloc(size * sizeof(int));
    printf("Dizinin tuttugu memory adresleri: \n");
    for (int i = 0; i < size; i++)
    {
        printf("%p\n", dizi + i);
    }
    printf("Lutfen realloc icin daha büyük bir buyukluk giriniz: ");
    scanf("%d", &size);
```

```

    printf("Dizinin tuttuğu memory adresleri: \n");
    for (int i = 0; i < size; i++)
    {
        printf("%p\n", dizi + i);
    }

    free(dizi);

    dizi = NULL;

    return EXIT_SUCCESS;
}

```

```

root@MSI: /mnt/c/Users/mybal/Desktop/C course/weeks/week07
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07# gcc -std=c99 -Wall -Werror week07.c -o week07
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07# ./week07
Lutfen dizinin buyuklugunu giriniz: 2
Dizinin tuttuğu memory adresleri:
0x7ffffdc4d26c0
0x7ffffdc4d26c4
Lutfen realloc icin daha buyuk bir buyukluk giriniz: 5
Dizinin tuttuğu memory adresleri:
0x7ffffdc4d26c0
0x7ffffdc4d26c4
0x7ffffdc4d26c8
0x7ffffdc4d26cc
0x7ffffdc4d26d0
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07#

```

**Not:** **malloc()** ile yer tahsis etme işlemi hangi kod bloğu içerisinde yapılıyorsa, tahsis edilen bu yeri **free()** ile serbest bırakma işlemi de aynı kod bloğu içerisinde yapılmalıdır.

Örneğin aşağıdaki örnekte **malloc()** ile yer tahsis etme işlemi main fonksiyonu içerisinde yapıldığı için **free()** ile serbest bırakma işlemi de main fonksiyonu içerisinde yapılmıştır.

```

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

void doldur(int *dizi, int size)
{

```

```

    for (int i = 0; i < size; i++)
    {
        printf("%d. elemani giriniz: ", i + 1);
        scanf("%d", dizi + i);
    }
}

void print(int *dizi, int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d. elemani: %d\n", i + 1, *(dizi + i));
    }
}

int main()
{
    int size = 10;
    int *dizi = (int *)malloc(size * sizeof(int));
    doldur(dizi, size);
    print(dizi, size);
    free(dizi);
    dizi = NULL;
    return EXIT_SUCCESS;
}

```



```
root@MSI: /mnt/c/Users/mybal/Desktop/C course/weeks/week07
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07# ./week07
1. elemani giriniz: 1
2. elemani giriniz: 2
3. elemani giriniz: 3
4. elemani giriniz: 4
5. elemani giriniz: 5
6. elemani giriniz: 6
7. elemani giriniz: 7
8. elemani giriniz: 8
9. elemani giriniz: 9
10. elemani giriniz: 10
1. elemani: 1
2. elemani: 2
3. elemani: 3
4. elemani: 4
5. elemani: 5
6. elemani: 6
7. elemani: 7
8. elemani: 8
9. elemani: 9
10. elemani: 10
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07#
```

## Struct Veri Yapısı

Şu ana kadar öğrenilen veri yapılarının hepsinde sadece tek bir veri tipinin yer alabildiği gözlemlendi. Örneğin diziler birden fazla **int**'in yer aldığı bir veri yapısıdır, **string**'ler ise birden fazla **char**'ın yer aldığı yapılardır. Bu durumda eğer bir **int** ve bir **char** aynı veri yapısında tutulmak isteniyorsa bu noktada devreye **struct** veri yapısı girer. **Struct** veri yapısı içerisinde birden çok veri tipini barındırabilir.

Struct yazım düzeni:

```
struct structureAdı
{
    veriTipi eleman1;
    veriTipi eleman2;
    ...
};
```

Struct global olarak tanımlanması gereken bir yapıdır. Bu sebeple kodun başlangıcında, tüm fonksiyonların üstünde bu veri yapısı şekillendirilir. İşlevine göre dizayn edilen bu yapı, tüm fonksiyonlar tarafından kendine özgü belirlenmiş ismine ait veri tipinde değişkenler tanımlanarak kullanılır. Örneğin:

```
struct Students
{
    int numara;
    char[20] isim;
};

int main
{
    struct Students student1;
};
```



Fonksiyonlar tarafından kullanılmak üzere tanımlanan **struct** veri yapısına sahip değişkenlerin içerdikleri elemanlara erişmek için değişkenin adı ve erişilmek istenen elemanın arasına “.” konulur. Örneğin:

```
int a = student1.id;
```

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

struct students
{
    char isim[20];
    char soyisim[20];
    int numara;
};

int main()
{
    struct students student1;
    printf("Lutfen isim, soyisim ve numara giriniz: ");
    scanf("%s %s", student1.isim, student1.soyisim);
    scanf("%d", &student1.numara);
    printf("Isim: %s\n", student1.isim);
    printf("Soyisim: %s\n", student1.soyisim);
    printf("Numara: %d\n", student1.numara);
    return EXIT_SUCCESS;
}
```

```
Select root@MSI: /mnt/c/Users/mybal/Desktop/C course/weeks/week07
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07# gcc -std=c99 -Wall -Werror week07.c -o week07
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07# ./week07
Lutfen isim, soyisim ve numara giriniz: Dennis Ritchie 1234
Isim: Dennis
Soyisim: Ritchie
Numara: 1234
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07#
```

## Keyword Typedef (Veri Tipi Tanımlama)

**typedef** ifadesi kullanılarak C dilindeki veri türlerini temsil eden kelimeler (**int**, **char**, **struct**, vs.) farklı isimlerle tanımlanabilir. Bu şekilde mevcut bir veri türü için yeni bir isim veya yeni bir veri türü oluşturulabilir. Genellikle struct veri yapısı için değişken tanımlarken kullanılan yazım düzenini basitleştirmek için kullanılır. Örneğin bir struct için **typedef** ifadesinin genel yapısı şu şekildedir:

```
typedef struct
{
    veriTipi eleman1;
    veriTipi eleman2;
    ...
}KEYWORD;

int main
{
    KEYWORD student1;
};
```

Eğer ki bir fonksiyon içerisinde tanımlanan **struct** veri yapısına sahip değişken bir pointer ise bu pointer'ın elemanlarına erişmek için pointer'ın adı ve erişilmek istenen elemanın arasına “->” konulur. Örneğin:

```
int main
{
    STUDENT *ptr;
    printf("%s",ptr->name);
};
```

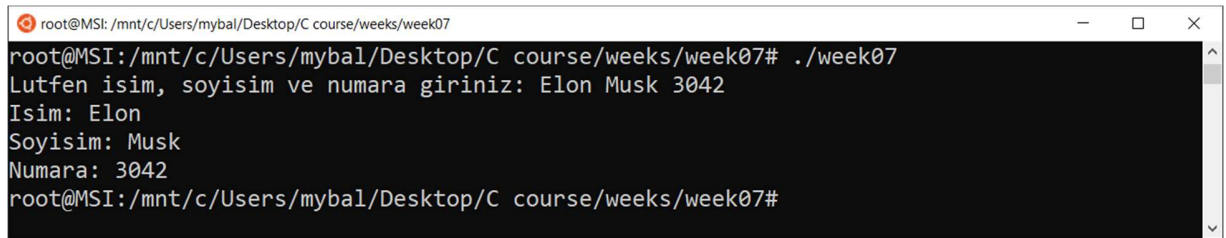
**Önemli Not:** *ptr->isim* ve *(\*ptr).isim* aynı şeylerdir.

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

typedef struct
{
    char isim[20];
    char soyisim[20];
    int numara;
} STUDENTS;

int main()
{
    STUDENTS *ptr;
    STUDENTS student1;
    ptr = &student1;
    printf("Lutfen isim, soyisim ve numara giriniz: ");
    scanf("%s %s", ptr->isim, ptr->soyisim);
    scanf("%d", &ptr->numara);
    printf("Isim: %s\n", student1.isim);
```

```
printf("Soyisim: %s\n", student1.soyisim);  
printf("Numara: %d\n", student1.numara);  
return EXIT_SUCCESS;  
}
```

A terminal window with a title bar showing the path /mnt/c/Users/mybal/Desktop/C course/weeks/week07. The terminal displays the execution of a C program. It prompts the user for name, surname, and ID number, and then prints the entered values.

```
root@MSI: /mnt/c/Users/mybal/Desktop/C course/weeks/week07  
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07# ./week07  
Lutfen isim, soyisim ve numara giriniz: Elon Musk 3042  
Isim: Elon  
Soyisim: Musk  
Numara: 3042  
root@MSI:/mnt/c/Users/mybal/Desktop/C course/weeks/week07#
```