Veri Yapıları
Lab Notları – 2

Bellek Yönetimi ve Göstericiler

Bellek Yönetimi

Bilgisayar üzerinde bir veri yapısını gerçekleştirebilmek için bilgisayar belleğinin kullanımına ihtiyaç vardır. Çalışan herhangi bir programdaki değişkenler, sınıflar, metotlar mutlaka bellekte bir yerde tutulurlar. Bellekte tutuldukları yerler bakımından 3 farklı bölge bulunmaktadır.

- Statik Bellek Bölgesi
- Çalışma Anı Yığını
- Heap Bellek Bölgesi

Statik Bellek Bölgesi

Bu bölgede yer alacak değişkenler hangileri olduğu, daha program başlamadan bellidir. Bu bölgede Global değişkenler, sabitler, static olarak tanımlanmış lokal değişkenler tutulurlar. Statik bellek bölgesinde tutulan değişkenler program çalıştığı sürece var olurlar, program sonlandığında bellekten silinirler.

Kod 1:

```
int kontrol;
const float pi=3.14;
int Arttir(){
   static int sayim = 0;
   sayim++;
   return sayim;
}
int main(){
   cout<<Arttir()<<endl;
   cout<<Arttir()<<endl;
   cout<<Arttir()<<endl;
}
```

Yukarıdaki kod örneğinde kontrol değişkeni global değişkendir. pi sayısı const ifadesi olduğu için sabittir ve Arttir metodunun içerisindeki sayim değişkeni de başında static olduğu için statik lokal değişkendir. Bu ismi anılan 3 değişkende statik bellek bölgesinde tutulur. Statik bellek bölgesinde tutulduğu için program sonlanıncaya kadar bellekte tutulurlar. Bundan dolayı Arttir metodu her çağrıldığında sayim değişkeni sıfırdan başlamak yerine kalmış olduğu değerden devam edecektir.

Global değişkenler bu bölgede tutuldukları için program sonlanıncaya kadar bellekte tutulacaktır ve programın herhangi bir satırından bu değişkenlere erişilebilecektir. İşte bu yüzden global değişkenlerin kullanımı (değişip değişmediklerinin kontrolü zor olduğu için) tavsiye edilmemektedir.

Kod 2:

```
bool dusman=false;
int main(){
   …
   if(dusman) FuzeGonder();
   …
}
```

Basitçe anlatmak için yukarıdaki kod bloğuna bakıldığında dusman değişkeni örneğin ilgili ülkenin düşman olup olmadığını tutuyor, global değişken olduğu için programın her satırından erişilebilir. Programın herhangi bir yerinde dusman değişkenine doğru değeri atanmış olup daha sonradan bu değiştirilmesi unutulmuş olabilir. Bu durumda if kontrolünün olduğu satır çalışacak ve ilgili yere füzeyi fırlatacaktır.

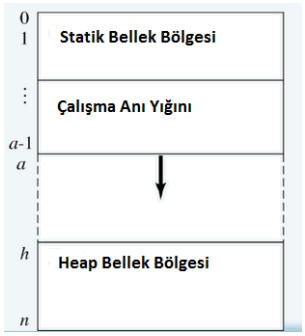
Çalışma Anı Yığını

En aktif bellek bölgesidir denilebilir. İsmini de oradan aldığı bu bellek bölgesi bir yığın (stack) şeklindedir ve bu yapıda çalışır. Bu yapıya ilk giren en son çıkar. Bir program çalıştığı sürece genişleyip daralan bitişik bir yapıya sahiptir. Bu bellek bölgesinde fonksiyon ve metot çağrımları ve bu fonksiyon ve metotların barındırdığı lokal değişkenler bulunur. Bir fonksiyon veya metot çağrıldığında bu fonksiyon veya metoda ait parametreler değişkenler ve dönüş değerleri bu bellek bölgesinde tutulur. Çalışma anı yığın bölgesi genişlemiş olur. Fonksiyon veya metot çağrılan yere döndüğünde bu fonksiyon veya metodun çalışma anı yığınında ayırmış olduğu yer geri döndürülür. Dolayısıyla geri döndürülen bu değişkenlere çağrım bittikten sonra erişim olamayacaktır.

Kod 3:

```
int DegerArttir(){
   static int sayac=0; // Statik bellek bölgesinde
   return ++sayac;
}
int topla(int a,int b){
    int sonuc = a+b;  // Çalışma anı yığınında
    return sonuc;
}
int main(){
    cout<<DegerArttir()<<endl;
    cout<<DegerArttir()<<endl;
    cout<<topla(21,10)<<endl;
    cout<<topla(5,7)<<endl;
    return 0;
}
```

Yukarıdaki kod örneğine bakıldığında, iki metot ve bir main metodu yer almaktadır. Main metodunda iki kere DegerArttir metodu çağrılmış ve daha sonra topla metodu çağrılmıştır. DegerArttir metodunun içerisindeki sayaç değişkeni statik lokal değişken olduğu için statik bellek bölgesinde diğer bütün değişkenler, çalışma anı yığınında oluşturulur. Topla metodunun çağrımı bittikten sonra, çalışma anı yığınında oluşturulmuş olan a, b ve sonuc değişkenleri bellekten yok edilirler.



Kod 4:

```
int sayi = 0;

int Arttir(int x){
        return ++x;
}
int main()
{
        cout<<Arttir(sayi)<<endl;
        cout<<sayi<<endl;
        return 0;
}
```

Yukarıdaki kod örneği incelendiğinde bir sayi adında global değişken tanımlanmıştır. Main metodu içerisinden Arttir metodu çağrılıyor ve daha sonra sayi ekrana yazdırılıyor. Arttir metodu parametre olarak aldığı değeri bir arttırıp geri döndüren bir metottur. Bu kodda sayi parametre olarak gönderiliyor, sayi global değişken olsa bile Arttir metodu içerisinde değeri değiştirilen lokal x değişkenidir dolayısıyla çağrım bittikten sonra bellekten silinecek fakat sayi değişkeni değerini yani sıfırı koruyacaktır.

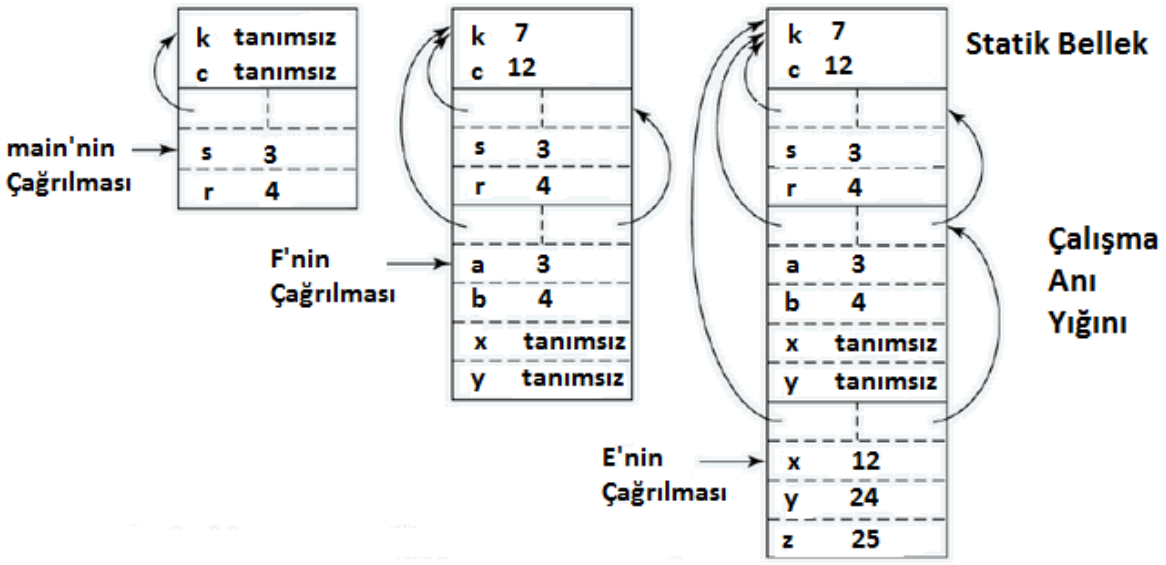
Kod 5:

```
int k,c;

void E(int x){
     int y = 2*x;
     int z = y + 1;
}
void F(int a,int b){
     int x,y;
     c=a*b;
     E(c);
}
int main(){
    int s=3,r=4;
    k=s+r;
    F(s,r);
}
```

Yukarıdaki kod bloğunda, k ve c değişkenleri global değişkenler, diğer bütün değişkenler lokal değişkenlerdir. Main metodundan çağrılma ve değer atanma sırasına bakıldığında bellekteki görüntü aşağıdaki gibi olacaktır. Şekilde gösterilen çalışma anı yığını bellek bölgesidir. Fakat bellek bölgesinin

üst tarafı statik bellek bölgesi olarak gösterilmiştir. Sol taraftan çıkan oklar statik çağrım. Sağ taraftan çıkan kollar ise dinamik çağrımdır. Örnekteki dinamik çağrımlar metot çağrımları. Statik çağrımlar ise global değişkenlerin çağrımını gösterir. Yığın aşağıya doğru genişliyor bu demek oluyor ki yığının baş tarafı aşağı taraf çünkü yığında ilk giren son çıkar prensibi olduğu için bu şekilde olmak durumunda.



Heap Bellek Bölgesi

Bu bellek bölgesi C ve C++ gibi programlama dillerinde dikkat edilmesi gereken çok önemli bir bölgedir. Çünkü C ve C++ gibi dillerde bu bölgenin kontrolü programcıya bırakılmıştır. Bu da demek oluyor ki eğer bu bölgenin kontrolü iyi sağlanmaz ise bellek taşması ya da yanlış değerlere erişim gibi problemler ile karşı karşıya kalınabilir. Bu bölgeye duyulan ihtiyacın nedeni, dinamik oluşturulan yapıların boyutları değişken olacak ve çalışma anında belirlenecektir. Dolayısıyla bu yapılar heap bellek bölgesinde tutulmalı, bu yapılara ve değişkenlere göstericiler yardımıyla erişilmelidir. Bu bellek bölgesinde tutulan bir değerin adı yoktur yani anonimdir ve ancak değerin bulunduğu adresi gösterecek bir gösterici yardımıyla erişilebilir.

Heap bellek bölgesinde C++ programlama dilinde bir yer ayırmak için new operatörü kullanılır. new operatörü kullanan göstericinin tutulduğu yer çalışma anı yığınıdır. Gösterdiği adres ise heap bellek bölgesinde bulunur. new operatörü kullanmadan tanımlanan göstericiler heap bellek bölgesinden yer ayıramazlar.

Kod 6:

```
// Adresi heap bellek bölgesinde
string *isimp = new string;

// Adresi yok.
string *adp;
```

Bu bölgede ayrılan yer işi bittiğinde belleğe geri verilmelidir. Bu bölgenin kontrolü programcıda olduğu için eğer geri döndürülmez ise çöp dediğimiz durum oluşur. Hatırlanırsa bu bölgedeki adreslere çalışma anı yığınındaki göstericiler yardımıyla erişiliyordu. Dolayısıyla çalışma anı yığınındaki

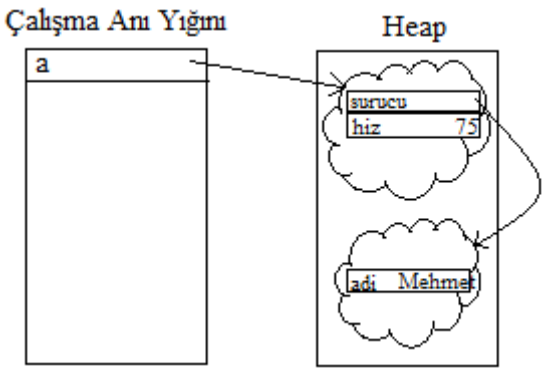
gösterici kaybedilmeden önce heap bellek bölgesinde ayrılan yer geri döndürülmelidir. Yoksa o bölge bilgisayar kapanıncaya kadar kullanılamaz duruma gelir.

Kod 7:

```
class Kisi{
        private:
                string adi;
        public:
                Kisi(string ad):adi(ad){ }
};
class Arac{
        private:
                Kisi *surucu;
                float hiz;
        public:
                Arac(string surucuAdi,float hz){    // Yapıcı fonksiyon
                                surucu = new Kisi(surucuAdi);
                                hiz=hz;
                        }
                ~Arac()   // Yıkıcı fonksiyon.
                {
                        delete surucu;
                }
};

int main()
{
        Arac *a = new Arac("Mehmet",75);
        delete a;
        return 0;
}
```

Yukarıdaki kod örneğinde araç sınıfından heap bellek bölgesinde bir nesne oluşturuluyor. Fakat dikkat edilirse Arac sınıfının yapıcı metodunda bir new operatörü daha var. Burada aracın sürücüsü olan Kişi nesnesi oluşturuluyor. Bu durumda Arac geri döndürülmeden içinde oluşturulmuş olan ve heap bellek bölgesinde bulunan sürücünün gösterdiği kişi nesnesi geri döndürülmesi gerekiyor. İşte bundan dolayı Arac sınıfının yıkıcı metodunda o alan geri döndürülüyor.



Göstericiler (Pointers)

C++ programlama dilinde göstericiler, herhangi bir bellek bölgesindeki bir değerin adresini gösterebilirler. Gösterdikleri yer, statik bellek bölgesi, çalışma anı yığını ya da heap bellek bölgesi olabilir.

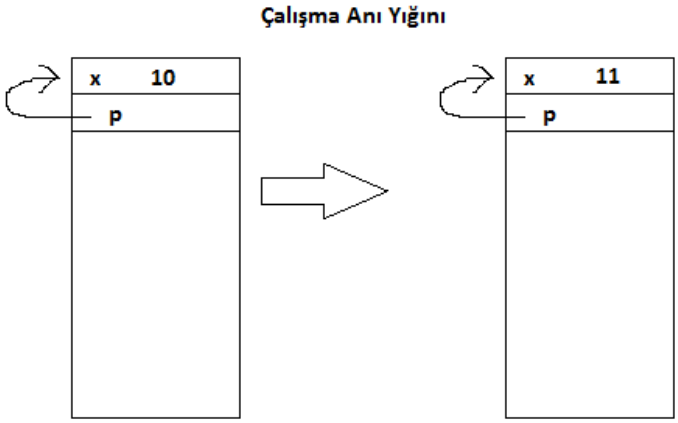
Kod 8:

```
int x=10;
int *p;
p=&x;  // x değişkenin adresini tut
x++;
cout<<*p;
```

Yukarıdaki kod bloğunda bütün değişkenler çalışma anı yığınında tutulurlar. X bir değer barındıran bir değişken, p ise bir adres barındıran bir göstericidir. 3 satırda x'in adresi p göstericisine atanıyor. Daha sonra x değişkenin değeri bir arttırıldığı ve p göstericisi x'i gösterdiği için p ekrana yazdırıldığında x'in en son değeri yani 11 ekrana yazacaktır.

Göstericilerde iki önemli operatör bulunmaktadır. * ve & operatörü

& operatörü adres operatörüdür. Başına gelen değişken ya da göstericinin adresini getirir. * operatörü ise tutulan adresteki değeri getirir.



Kod 9:

```
int *p;
int a=12;
p=&a;

cout<<*p;       // Gösterdiği yerdeki değeri yaz
cout<<p;        // Gösterdiği yerin bellek adresi
cout<<&p;      // Göstericinin kendi bulunduğu adres
cout<<&a;      // a değişkenin bulunduğu adres.
```

Göstericilerde atama işlemi sadece adres ataması olduğu için atanan gösterici de aynı adresi yani aynı değeri gösterecektir.

Kod 10:

```
int x=10;
int *p,*r;
p=&x;
r=p;
cout<<*r; // Ekrana 10 yazar.
```

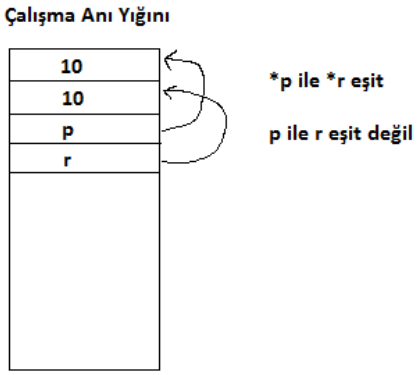
Yukarıdaki kod bloğuna bakıldığında p x değişkenini gösteren bir gösterici ve atama operatörü kullanılarak p'deki adresi r'ye atanıyor, böylelikle r'de x değişkenini göstermeye başlıyor. Bu durumda önemli bir kavram ortaya çıkmaktadır. Gösterici karşılaştırma.

Gösteri karşılaştırma aslında karşılaştırmadan ne kastedildiğine göre değişir. Eğer aynı adresleri gösterdikleri kontrol ediliyorsa

if(p == r) // Aynı adres olup olmadığı kontrol ediliyor.

Eğer aynı değeri tutup tutmadıkları kontrol ediliyorsa bu durumda aşağıdaki gibi bir karşılaştırma yapılması gerekmektedir.

if(*p == *r) // Aynı değerin saklanıp saklanmadığı kontrol ediliyor.



Aşağıdaki örneğe bakıldığında x ve y farklı adreslerde tutulan iki değişken xp x'in adresini gösteren gösterici ve yp y'nin adresini gösteren gösterici. x=y dendiği zaman y'nin değeri x'in değerine atandığı için iki farklı adreste de 4 değeri vardır. Göstericileri ekrana yazdırdığınızda her ikisinde de 4 yazacaktır. Göstericilerin gösterdiği adreslerde bir değişiklik olmamıştır.

Kod 11:

```
int x=3,y=4;
int *xp,*yp;
xp=&x;
yp=&y;
x=y;
cout<<*xp<<endl;
cout<<*yp<<endl;
```

Yukarıdaki kod aşağıdaki gibi değiştirildiğinde, xp'nin gösterdiği adresi değiştirmektir. xp artık x'in adresini değil yp'nin gösterdiği yani y'nin adresini göstermektedir. Ekrana her ikisi de 4 yazacaktır.

Kod 12:

```
int x=3,y=4;
int *xp,*yp;
xp=&x;
yp=&y;
x=y;
cout<<*xp<<endl;
cout<<*yp<<endl;
```

Yine aynı şekilde aynı kod aşağıdaki gibi değiştirilse, yp'nin gösterdiği adresteki değeri yani y'nin değerini xp'nin gösterdiği adrese ata demektir. Yani x'in değerini de 4 yap demektir. Dikkat edilmesi gereken konu xp'nin ve yp'nin gösterdikleri adresler değişmemiş ve farklı adresleri göstermektedirler.

Diziler

Homojen verilerin bir araya gelerek oluşturdukları yapıdır. Bir dizi içerisinde aynı tür veri bulunur. Dizi indeksi sıfırdan başladığı için son indeks elemansayısı – 1 olarak ifade edilir. Diziler bellekte ilk adresleri tutularak erişilirler. Bundan dolayı C++'ta bir dizi bir işaretçiye (pointer) atanır ve aşağıdaki gibi işaretçi ekrana yazılırsa dizinin ilk elemanı ekrana yazılacaktır.

```
int main(){
   int x[5];
   x[0]=100;
   int *p = x;
   cout<<*p;
   return 0;
}
```

İki boyutlu dizilerde tanımlama yukarıdakine benzer koşullarda aynıdır. Burada dikkat edilmesi gereken dizilerin arka planda aslında bir gösterici şeklinde tutuldukları için aşağıdaki iki boyutlu dizi tanımlamasında ekrana adres yazacaktır.

```
int x[3][3];
x[0][0]=100;
cout<<x[0];
```

Dizilerin bellekte tutulma şekilleri bir göstericinin bellekte tutulma şekli ile aynıdır. Yapılan iş sadece ilk elemanın adresini tutmaktır. Derleyici dizinin ilk elemanının adresini tutmakla yetineceği için diziler tanımlandıkları yerde boyutları belirtilmelidir ki derleyici adresi nereye kadar arttırabileceğini bilsin. Siz sayilar[3]'teki elemanı getir dediğinizde derleyici arka tarafta aslında *(sayilar+3) adresindeki değeri getir demektedir.

Programda sayıların tutulduğu adresleri ekrana yazmaya kalkarsanız. Adreslerin ardışık olduğunu göreceksiniz.

0x28ff00
0x28ff04
0x28ff08
0x28ff0c

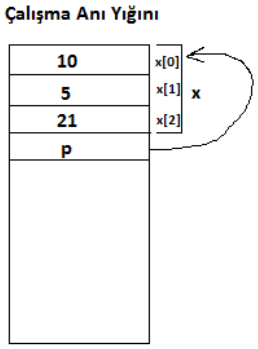
Adreslere bakıldığında 4 farkla ilerlediği görülür. 00+04=08 hexadecimal olarak. Bunun nedeni int bellekte 4 byte olarak tutulduğundan kaynaklanmaktadır (C++).

Dizi Göstericileri

C++ programlama dilinde diziler bellekte ardışık bir şekilde tutulurlar. Bir dizi tanımlamak aslında arka planda bir gösterici tanımlamakla çok farklı değildir. Bunun da en büyük göstergesi aşağıdaki örnektir. Aşağıdaki örneğe bakıldığında bir tamsayılar dizisi tanımlanmış ve bir göstericiye atanmıştır. Ama dikkat edilirse x'in başında & operatörü yoktur. Bunun nedeni deminde bahsettiğimiz x'inde arka planda aslında bir gösterici olduğudur. Burada p göstericisine dizi göstericisi denir. Bu dizi göstericileri dizilerin ilk adreslerini tutarlar.

Kod 13:

```
int x[]={10,5,21};
int *p;
p=x;
```



Eğer sistem tamsayılar için bellekte 4 byte yer ayırıyorsa p göstericisinin değerine 1 eklemek yani 4 byte eklemek olacaktır, dizideki bir sonraki elemana işaret edecektir. X dizisinin elemanlarının adresleri yazdırılırsa görülecektir ki adres değerleri hex decimal olarak 4 byte, artarak ilerlemektedir.

Gösterici Göstericileri

Bu tip göstericilerin diğer göstericilerden farkı, gösterdikleri yerde yine bir gösterici bulunmaktadır. Yani değere iki adımda erişilebilmektedir.

int *p; // Bir tamsayı değerinin adresini gösteren gösterici

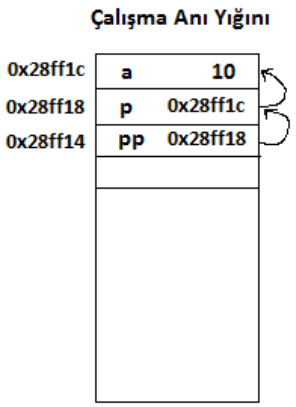
int **pp; // Bir tamsayı göstericisini gösteren gösterici.

Aşağıdaki örneğe bakıldığında a bir tamsayı değişkenidir, p ise a'yı gösteren bir gösterici, pp'de p'yi gösteren bir gösterici yani gösterici göstericisidir (pointer to pointer).

Kod 14:

```
int a=10;
int *p = &a;
int **pp = &p;
cout<<*p;  // 10 yazar
cout<<**pp; // 10 yazar
```

```
cout<<*pp; // a'nın adresi
cout<<pp;  // p'nin adresi
cout<<&pp; // pp'nin adresi
```

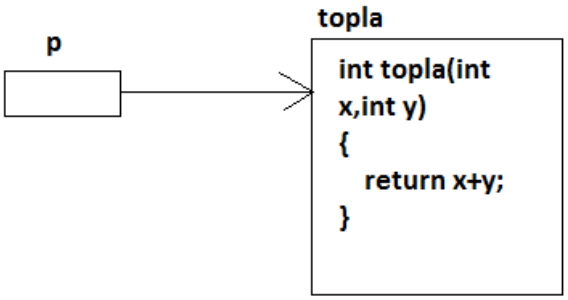


Fonksiyon Göstericileri

Göstericiler, ilkel türleri ve tanımlamış olduğunuz nesneleri gösterebilecekleri gibi, bir fonksiyona da işaret edebilirler. Bu tip göstericilerde en önemli husus, gösterici ile gösterdiği fonksiyonun prototipi uymalıdır. Örneğin **int (*p)(int,int);** şeklinde tanımlanmış bir fonksiyon göstericisi, işaret edebileceği fonksiyon iki parametre alan ve tamsayı döndüren bir fonksiyon olabilir. Göstericideki parantez, derleyici tarafından normal göstericiymiş gibi ya da gösterici döndüren bir fonksiyonmuş gibi algılanmasın diye kullanılmıştır. Aşağıdaki örneğe bakıldığında p fonksiyon göstericisi tanımlanıyor. p fonksiyon göstericisi topla fonksiyonunu göstermesi sağlanıyor daha sonra p göstericisi çağrılıyor.

Kod 15:

```
int topla(int x,int y)
{
   return x+y;
}
int main()
{
  int (*p)(int,int);
  p=topla;
  cout<<(*p)(25,30);
}
```



Yine aynı yöntemle bir sınıfın alt metodunu gösteren bir gösterici de tanımlanabilir. Aşağıdaki örneğe bakıldığında Sayi sınıfının Deger metodunu gösteren bir gösterici tanımlanıyor.

int (Sayi::*pdeger)(); // Sayi::*pdeger denmesinin sebebi bu göstericinin bir sınıfın elemanı olan metodu göstereceği içindir. Burada dikkat edilmesi gereken durum, sınıfın dışından private alandaki metotları gösteren gösterici tanımlanamaz.

Kod 16:

```
class Sayi{
      private:
          int deger;
      public:
          Sayi(int dgr=0):deger(dgr) {  }
          int Deger() { return deger; }
};
int main()
{
  Sayi *s = new Sayi(25);
  int (Sayi::*pdeger)();
  pdeger = &Sayi::Deger;
  cout<<(s->*pdeger)();
}
```

Void Göstericisi

Türü olmayan bir göstericidir. Dolayısıyla yeri geldiğinde bir tamsayıyı gösterebileceği gibi yeri geldiğinde bir ondalık sayıyı da gösterebilir. Sadece göstericinin gösterdiği yer kullanılacağı zaman, derleyicinin o anki hangi tür olduğu bilmesi açısından dönüştürme işlemi uygulanmalıdır. Bunun örneği aşağıdaki kod parçasında görülebilir.

Kod 17:

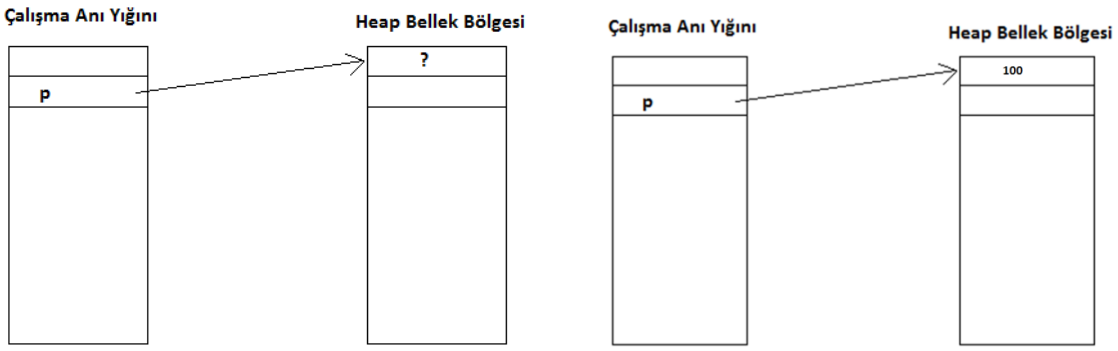
```
int x=100;
float a=12.5;
void* obj;
obj=&x;
cout<<*static_cast<int *>(obj)<<endl;
obj=&a;
cout<<*static_cast<float *>(obj)<<endl;
```

new Operatörü

Programcının kontrolüne bırakılmış olan heap bellek bölgesinde, bellekten yer ayırmak için new operatörü kullanılır. Programın herhangi bir yerinde new operatörü görülmüş ise mutlaka orada heap bellek bölgesinden bir yer ayırma vardır. Bu ayrılan alan gösterici kaybedilmeden geri döndürülmelidir. Yoksa ayrılmış olan o alan bilgisayar kapanan kadar kullanılamaz bir alan olacak yani çöp olayı dediğimiz durum gerçekleşecektir.

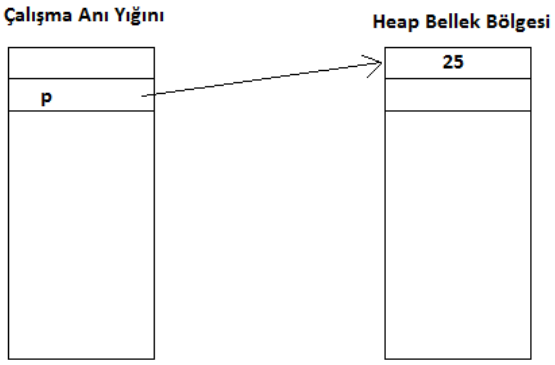
int *p = new int; (a)

int *p = new int(100);



(a) (b)

*p=25; // p nin gösterdiği adresteki yere 25 değerini yaz.

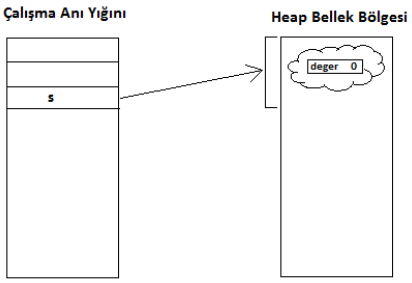


Heap bellek bölgesinde herhangi bir tür veri için yer ayrılabilir. Örneğin programcı tarafından yazılmış olan sayi turu heap bellek bölgesinde yere ayırmak için aşağıdaki kod bloğuna bakılabilir.

Kod 18:

```
class Sayi{
      private:
          int deger;
      public:
          Sayi():deger(0) {  }
};

Sayi *s = new Sayi();
```



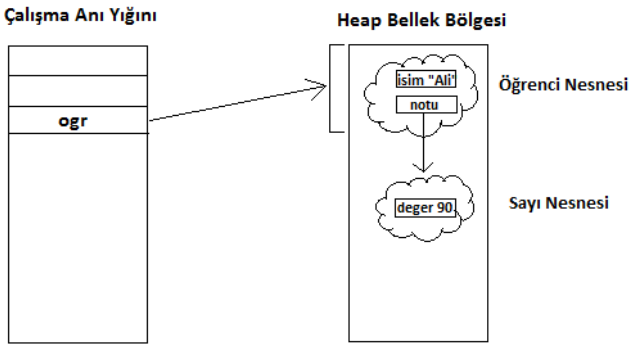
Yukarıdaki kod bloğunda Sayi nesnesini heap bölgesinde oluşturan kişi onu kullandıktan sonra belleğe geri döndürecektir. Fakat aşağıdaki örnekte de verilen durumda oluşturulan bir nesne içerisinde bir veya daha fazla nesne heap bellek bölgesinde nesnenin kendisinin oluşması ile birlikte oluşabilir. Bu durumda sınıfı tasarlayanların görevi olarak nesnenin yıkılma anında yani yıkıcı metot içerisinde bu heap'te oluşan nesneleri geri döndürmelidirler.

Kod 19:

```
class Ogrenci{
     private:
          string isim;
          Sayi *notu;
     public:
```

```
        Ogrenci(string ad,int nt){
            isim=ad;
            notu=new Sayi(nt);
        }
        ~Ogrenci(){ // Yıkıcı metot
            delete notu;
        }
};

int main(){
  Ogrenci *ogr = new Ogrenci("Ali",90);
  delete ogr;
  return 0;
}
```



Heap bellek bölgesinde oluşturulmuş bir alanı geri döndürmek için delete komutu kullanılır. delete sadece göstericilerde kullanılabilecek bir komuttur.

```
Sayi s1;
Sayi *sp = new Sayi();
delete s1; // Hatalı kullanım
delete sp; // Doğru kullanım
```

Çöp olayını daha iyi anlayabilmek açısından aşağıdaki örnekte Heap bellek bölgesinde oluşturduğumuz Sayi nesnesini for döngüsünün sonunda geri döndürmediğimiz için *s göstericisi de for döngüsü içinde tanımlandığı için bir döngü bittiğinde gösterici yok edilecektir ama gösterdiği bölge programcıya ait olduğu için ve göstericisini kaybettiği için orası çöp bölgeye dönmüştür. Ve for döngüsü ile adresler yazdırıldığında görülecektir ki sürekli farklı adresten yer ayrılmaktadır.

Kod 20:

```
for(int i=0;i<10;i++){
  Sayi *s = new Sayi(125);
  cout<<s<<endl;
}
```

```
0x7b1838
0x7b2458
0x7b2468
0x7b2478
0x7b2488
0x7b2498
0x7b24a8
0x7b24b8
0x7b24c8
0x7b24d8
```

Doğru olan yapıldığında yani her döngünün sonunda göstericiyi kaybetmeden gösterdiği yeri belleğe geri iade ettiğimizde ekrana adresleri yazdırırsak sürekli aynı adresin bize verildiğini görürüz. Bunun nedeni o alanın geri döndürüldüğü için tekrar kullanılabiliyor olmasıdır.

Kod 21:

```
for(int i=0;i<10;i++){
  Sayi *s = new Sayi(125);
  cout<<s<<endl;
  delete s;
}
```

```
0x4d1838
0x4d1838
0x4d1838
0x4d1838
0x4d1838
0x4d1838
0x4d1838
0x4d1838
0x4d1838
0x4d1838
```

Sallanan Göstericiler (Dangling Pointers)

Eğer bir gösterici daha önce belleğe geri döndürülmüş bir adresi tutuyorsa bu göstericiye sallanan gösterici denir. Sallanan göstericiler tehlikeli sonuçlara sebep olabilirler. Örneğin göstermiş olduğu adres farklı tipte bir alan için ayrılmışsa üzerinde işlem yapmak hataya sebep olacaktır.

```
int main()
{
    Sayi *s1,*s2;
    s1 = new Sayi(50);
    s2=s1;
    delete s1;
    Sayi *s3 = new Sayi(220);
    cout<<"s1'in tuttugu adres:"<<s1<<endl;
    cout<<"s2'nin tuttugu adres"<<s2<<endl;
    cout<<"s3'nin tuttugu adres"<<s3<<endl;

    cout<<"s2'nin tuttugu değer:"<<s2->Deger()<<endl;



    return 0;
}
```

Hazırlayan
Arş. Gör. Dr. M. Fatih ADAK