GÖSTERİCİLER VE REFERANSLAR

Gösterici nedir? Niçin İhtiyaç Duyarız?

Şu ana kadar gördüğümüz `char`, `int`, `long`, `float`, `double` tipli değişkenler ve bunlara ait diziler, değer tipler (value type) olarak adlandırılır. Çünkü kimliklendirilen değişken tutulacak değeri içerir.

Değer tiplerin yanı sıra değişkenlerin adreslerini tutan değişkenlere de ihtiyaç duyarız. İşte adres tutan değişkenlere gösterici tipler (pointer type) adını veririz. Gösterici tipler, gösterdiği yerdeki verileri değiştirmek için de kullanılır. Bu nedenle gösterici tipler tanımlanırken gösterdiği yerdeki verinin tipine göre tanımlanırlar. Bütün gösterici tipler bellekte aynı miktarda yer kaplar. Çünkü hepsi adres tutar. Ancak işaret ettikleri yerdeki veriler, verinin tipine bağlı olarak bellekte farklı miktarda yer kaplar. Bütün bu tanımlamaların ışığında göstericileri, vatandaşların ikametlerini gösteren adres numaraları olarak düşünebiliriz.

Verilerin tipine göre gösterici tanımlandığından göstericiler türetilmiş tiplerdir (derived type). Gösterici tiplere ihtiyaç duyulmasının sebebi hız ve esnekliktir. Genel olarak birkaç madde ile sıralayacak olursak;

- Belleğin istenilen bölgesine erişim sağlamak için göstericiler kullanılır. Günümüz modern dillerinde referans tipler kullanılır, ancak referans tiplerde gösterilen adreste bir nesnenin bulunduğu garanti edilir göstericiler gibi her bellek bölgesine erişilemez.
- Bazen çalıştırma anında yeni bellek gölgelerine ihtiyaç duyarız bu durumda göstericiler gereklidir. Dinamik veri yapıları (dynamic data structure) göstericiler yardımıyla oluşturulup yönetilir.
- Fonksiyonlarda yerel değişkenler ve argümanlar yığın belleğe itilir ve fonksiyondan geri dönüldüğünde bu değişkenler eski haline yığın bellekten geri çekilerek çevrilir. Fonksiyonun yerel değişken ya da argümanlardan birini değiştirmesi istendiğinde fonksiyona argüman olarak değişkenin adresi verilir. Böylece fonksiyon içinde gösterici olarak işlem gören argümanlar fonksiyondan geri döndüğünde değerleri değişmiş olur.

Göstericilere olan ihtiyacı bir başka örnekle de açıklayabiliriz; Bir ormandaki ağaçları boylarına göre sıralamaya kalkarsak her birini yer değiştirme yöntemiyle sıralamak oldukça külfetli ve zaman alana bir işlemdir. Halbuki ağaçlara numara vererek ve bu numaraların karşısına uzunluklarını yazacağımız bir liste oluşturulduğunda sadece numaraları sıralamak oldukça kolay ve zahmetsiz bir işlemdir. İşte buradaki numaraları tutan değişkenler göstericilerdir.

Gösterici Tanımlama

Göstericiler, gösterdiği yerdeki verinin tipine göre tanımlanırlar.

Gösterici tanımlanırken veri tipi izleyen başvuru kaldırma işleci (dereference operator) olan yıldız (`*`) kullanılır. Başvuru kaldırma terimi, gösterici tarafından tutulan bellek adresinde saklanan değere erişmeyi de ifade eder.

```
veritipi* gostericikimligi;
```

Göstericilere değer atatama aşağıdaki şekilde yapılır;

```
gostericikimligi = &degiskenkimliği;
```

Bir değişkenin adresinin referans işleci (`&`) ile elde edildiğini öğrenmiştik. Aşağıda göstericilere ilişkin kod örnekleri verilmiştir;

```
char* ptrToChar;  /* tuttuğu adreste char tipinde veri olan
                     ptrToChar kimlikli göstericisi tanımlandı. */
int i=10;
int* ptrToInt=&i; /* tuttuğu adreste int tipinde veri olan
                     ptrToInt kimlikli göstericisi tanımlandı ve
                     ilk değer olarak i değişkeninin adresi atanıyor */
```

```
*ptrToInt =20; /* ptrToInt göstericisinin gösterdiği adresteki tamsayı
                  değeri 20 yaptık. ptrToInt göstericisi, i değişkeninin
                  adresini tuttuğundan i değişkeninin değeri de 20 olmuştur */
```

Bazı durumlarda göstericilerin veya gösterdiği değerlerin sabit olması gerekebilir. Üç durum ortaya çıkabilir;

- Gösterilen değerin sabit olması: Bu durumda gösterici tanımlaması aşağıdaki gibi yapılır;

```
int main() {
   int i=10;
   const int* ptrToi=&i; // değerin sabit olması durumunda gösterici tanımı
   *ptrToi=20; /* HATA! pi göstericisinin tuttuğu adresteki tamsayı
                  değeri 20 yapılamaz. */
}
```

- Göstericinin işaret ettiği adresin sabit olması: Kısaca göstericinin sabit olması durumunda tanımlama aşağıdaki gibi yapılır;

```
int main() {
   int i=10;
   int* const ptrToi=&i; /* göstericinin sabit olması durumunda gösterici tanımı.
                            Bu tür tanımlamada adres ilk değer (initial value)
                            olarak mutlaka verilmelidir. */
   int j=30;
   *ptrToi=20; // pi göstericisinin tuttuğu adresteki tamsayı değeri 20 olur.
   ptrToi =&j; //HATA! : pi göstericisinin tuttuğu adres değiştirilemez!
}
```

- Hem göstericinin hem de değerin sabit olması durumu:

```
int main() {
   int j=30;
   const int* const ptr=&j; /* Hem gösterici, hem de gösterilen veri sabit
                               Bu tür tanımlamada da adres ilk değer olarak
                               mutlaka verilmelidir. */
}
```

Atanmış kesin bir adresiniz yoksa, bir gösterici değişkenine `NULL` değeri atamak her zaman iyi bir uygulamadır. İlk değeri olmayan gösterici tanımlamak yerine, ilk değeri sıfır olan göstericiler tanımlanmak doru bir yöntemdir. Bu durumda ilk değer `nullptr` olarak atanır ve bu göstericiye, NULL gösterici (NULL pointer) adı verilir. Aşağıdaki gibi kullanılır;

```
veritipi* gostericikimligi = nullptr;
gostericikimligi = nullptr;
```

Örnek kod aşağıda verilebilir;

```
#include <iostream>
using namespace std;
int main(){
 int* ptr = nullptr;
 cout <<"ptr göstericisinin tuttuğu adres:" << ptr << endl;
 int agirlik=80;
 ptr=&agirlik;
 cout << "ptr göstericisinin tuttuğu adres:" << ptr << " ve değeri:" << *ptr << endl;
}
/* Program Çıktısı:
ptr göstericisinin tuttuğu adres: (nil)
ptr göstericisinin tuttuğu adres: 0x7ffdedc33dec ve değeri: 80

...Program finished with exit code 0
*/
```

Bir gösterici bir başka göstericinin adresini tutabilir. Buna çifte gösterici (double pointer) adı verilir. Aşağıda buna ilişkin bir örnek bulunmaktadır;

```
#include <iostream>
using namespace std;
int main(){
   int var = 10;
   int* intptr = &var;
   int** ptrptr = &intptr;  //doble pointer

   cout << "var:" << var << " &var:" << &var << endl;
   cout << "inttptr:" <<  intptr  << " *inttptr:" << *intptr << endl;

   cout << "var:" << var << " *intptr:" << *intptr << endl;
   cout << "ptrptr:" << ptrptr << " &ptrtptr:" << &ptrptr << endl;
   cout << "&intptr:" << &intptr << " *ptrptr :" << *ptrptr << endl;
   cout << "var:"<< var<< " *intptr:" << *intptr<<" **ptrptr:" << **ptrptr;
}
/* Program Çıktısı:
var:10 &var:0x7ffe0e839f74
inttptr:0x7ffe0e839f74 *inttptr:10
var:10 *intptr:10
ptrptr:0x7ffe0e839f78 &ptrtptr:0x7ffe0e839f80
&intptr:0x7ffe0e839f78 *ptrptr :0x7ffe0e839f74
var:10 *intptr:10 **ptrptr:10

...Program finished with exit code 0
*/
```

Gösterici Aritmetiği

Bilgisayarların tasarımından ötürü işlemciye bağlı belleklerde her bir ayrı adreslenir. Birkaç bayt(`char`) veri içeren bir bellek bölgesini işaret eden bir gösterici tanımlandığında göstericinin işaret ettiği adres, bir sonraki baytı göstermesi istendiğinde gösterici adresi 1 artar.

```
char dizi1[5]={'I','L','H','A','N'}; /* Bellekte Art arda 5 bayt yer ayrılmış dizi. */
char* pc=&dizi1[0]; /* pc göstericisi dizinin ilk elemanını gösteren adres (65FDE0)
varsayıldı. */
*pc='X';         //dizi {'X','L','H','A','N'} oldu.
pc++;            /* pc göstericisi bir sonraki baytı gösterir: (65FDE1) adres 1 artırıldı */
*pc='Y';         // dizi {'X','Y','H','A','N'} oldu.
pc+=2;           /* pc göstericisi 2 sonraki baytı gösterir: (65FDE2) adres 2 artırıldı. */
*pc='Z';         // dizi {'X','Y','H','A','Z'} oldu.
```

Benzer şekilde birkaç tamsayı (`int`) veri içeren bir bellek bölgesini işaret eden bir gösterici tanımlandığında göstericinin işaret ettiği adres, bir sonraki tamsayıyı göstermesi istendiğinde gösterici adresi 4 artar. Çünkü tamsayılar bellekte 4 bayt yer kaplar.

```
int dizi2[5]={2,0,3,10,2}; /* Bellekte art arda 5 tamsayı (5x4bayt) yer ayrılmış dizi. */
int* pi=&dizi2[0];  /*pi göstericisi dizinin ilk elemanını gösteren  adres (65FDE0)
varsayıldı. */
*pi=-1;          //dizi {-1,0,3,10,2} şekline döndü.
pi++;            /* pi göstericisi ikinci elemanını gösterir: (65FDE4) adres 4 artırıldı. */
*pi=-2;          //dizi {-1,-2,3,10,2} şekline döndü.
pi+=2;           /* pi göstericisi iki sonraki elemanını gösterir: (65FDEC) adres 8 artırıldı.
*/
*pi=-3;          //dizi {-1,-2,3,10,-3} şekline döndü.
```

Görüldüğü üzere göstericilerin üzerinde işleçler (operator) işlem yaparken göstericinin işaret ettiği verinin bellekte kapladığı yer dikkate alırlar. Göstericiler ile sadece aşağıdaki işleçler çalışır, diğer işleçlerde işlenen (operand) olarak kullanılmaz.

– Aritmetik ve atama işleçlerinden `++`, `--`, `+`, `-`, `+=` ve `-=`
– Karşılaştırma işleçlerinden `<`, `>` ve `==`

Gösterici Dizileri

Göstericiler de dizi olarak tanımlanabilir. Aşağıda buna ilişkin bir örnek verilmiştir;

```
#include <iostream>
using namespace std;
int main() {
  int i = 10, j=20;
  float f=1.0;
  int k=30, l=40;

  int* ptr[4]; /* int gösteren 4 gösteri içeren bir dizi tanımlandı */

  /* Göstericilere ilk değer veriliyor*/
  ptr[0] = &i; // Birinci eleman i değişkeninin adresini
  ptr[1] = &j; // İkini eleman j değişkeninin adresini
  ptr[2] = &l; // Üçüncü eleman l değişkeninin adresini
  ptr[3] = &k; // Dördüncü eleman k değişkeninin adresini

  // Değerlere Ulaşma
  int indis;
  for (indis = 0; indis < 4; indis++)
     cout << "ptr[" << indis <<"] ile gösterilen değer:" << *ptr[indis] << endl;
}
/* Program çalıştırıldığında;
ptr[0] ile gösterilen değer:10
ptr[1] ile gösterilen değer:20
ptr[2] ile gösterilen değer:40
ptr[3] ile gösterilen değer:30

...Program finished with exit code 0
*/
```

Parametre Olarak Göstericiler

Fonksiyonlar çağrılırken argüman olarak giren değerlerin değişmeyeceği *Fonksiyon Çağırma Süreci* başlığında anlatılmıştı. Fonksiyona parametre olarak giren değerler, fonksiyon bloğu içinde değişse bile fonksiyondan geri dönülürken aynı çağrı ortamını sağlamak için kaybolurlar.

```
#include <iostream>
using namespace std;
void degistir(int,int);
int main(){
   int a = 10, b = 20;
   degistir(a, b);
   cout << "Değiştir Sonrası a:" << a<< ", b:" << b << endl;
   //Çıktı: Değiştir Sonrası a:10, b:20
}
void degistir(int pX, int pY) {
   int z = pX;
   pX=pY;
   pY=z;
}
```

Fonksiyona giren argümanları gösterici olarak tanımlarsak çağrı ortamına geri dönüldüğünde yerel değişkenler de değişmiş olur. Çünkü fonksiyona değişken adresleri değer olarak girmiştir. Buna ilişkin örnek aşağıda verilmiştir;

```
#include <iostream>
```

```
using namespace std;
void degistir(int,int);
void degistir2(int*, int*);
int main(){
   int a = 10, b = 20;
   degistir(a, b);
   cout << "Değiştir Sonrası a:" << a << ", b:" << b << endl;
   //Çıktı: Değiştir Sonrası a:10, b:20
   degistir2(&a, &b);
   cout << "Değiştir2 Sonrası a:" << a << ", b:" << b << endl;
   //Çıktı: Değiştir2 Sonrası a:20, b:10
}
void degistir(int pX, int pY) {
   int z = pX;
   pX=pY;
   pY=z;
}
void degistir2(int* pX, int* pY){
   int z = *pX;
   *pX=*pY;
   *pY=z;
}
```

Göstericilerin Dizileri İşaret Etmesi

Çoğu durumda, bir dizi yardımıyla gerçekleştirdiğimiz işleri gösterici ile de gerçekleştirilebiliriz. Genellikle dizilerin kendisi yerine ilk elemanlarını işaret eden göstericiler kullanırız.

```
#include <iostream>
using namespace std;
int main () {
   int dizi[5] = {10, 20, 30, 40, 50};
   int* ptr = dizi;  // int* ptr=& dizi[0]; olarak da kodlanabilir.
   cout << "Dizi elemanlarına gösterici ile erişim:" << endl;
   for(int indis = 0; indis < 5; indis++)
      cout << "dizi[" << indis << "]:" << *(ptr+indis) << endl;
}
```

Fonksiyonlara dizileri gösteren göstericileri parametre olarak verebiliriz;

- Diziler parametre olarak kullanılırken adres operatörü kullanılmaz. Diziler, gösterici gibi davranırlar. Yani bir dizinin adı, dizinin ilk öğesinin adresi gibi davranır.

```
float notlar[10];
float notlarOrtalamasi(float*); //prototype
//...
float ort=notlarOrtalamasi(notlar); // yada notlarOrtalamasi(&notlar[0]);
//...
float matris[4][3];
float defetminant(float**,int,int); //prototype
//...
float det=defetminant(matris,4,3);
//...
```

- Gösterici gibi davrandıklarından dizi elemanları fonksiyon içinde yine dizi gibi kullanılabilir.
- Gösterici gibi davrandıklarından dizi elemanları fonksiyon içinde değiştirilebilir. Fonksiyondan dönüldüğünde elemanlar değişiklik yapılmış şekliyle işlemler devam eder.

Aşağıda öğrencilerin notlarına ilişkin işlemler yapılan bir program verilmiştir.

```
#include <iostream>
using namespace std;
#define OGRENCISAYISI 10
```

```
float notlarOrtalamasi(float*);
void notlariArtir(float*);
int main(){
   float notlar[OGRENCISAYISI]= { 55.0,60.0,70.0,35.0,30.0,50.0,65.0,90.0,95.0,100.0 };
   float toplam=notlarOrtalamasi(notlar);
   cout << "Notlar Ortalaması:" << toplam << endl;
   notlariArtir(notlar);  // Bu fonksiyonla dizi elemanları değiştiriliyor
   int indis;
   for (indis=0; indis<OGRENCISAYISI; indis++)
      cout << "Not[" << indis << "]=" << notlar[indis] << endl;
   return 0;
}
float notlarOrtalamasi(float* pNotlar){
   int indis;
   float top=0.0;
   for (indis=0; indis<OGRENCISAYISI; indis++)
      top+=pNotlar[indis]; // Gösterici dizi gibi kullanılıyor
   return top/OGRENCISAYISI;
}
void notlariArtir(float* pNotlar) {
   int indis;
   for (indis=0; indis<OGRENCISAYISI; indis++)
      if (pNotlar[indis]<=90) //Göstericinin gösterdiği değerler değiştiriliyor
          pNotlar[indis]+=10.0;
};
/* Program Çıktısı:
Notlar Ortalaması: 65.0
Not[0]=65.0
Not[1]=70.0
Not[2]=80.0
Not[3]=45.0
Not[4]=40.0
Not[5]=60.0
Not[6]=75.0
Not[7]=100.0
Not[8]=95.0
Not[9]=100.0

...Program finished with exit code 0
*/
```

Fonksiyonlardan Bir Diziyi Geri Döndürme

C++ dilinde bir fonksiyonun geri dönüş değeri olarak tüm bir dizi verilemez! Ancak, bir dizinin göstericisini fonksiyondan geri dönüş değeri olarak döndürebilirsiniz. Burada dikkat edilmesi gereken fonksiyondan çıkılınca dizinin bellekten kaldırılmamasıdır. Aşağıda bir tamsayı dizi göstericisini döndüren bir fonksiyon örnekleri verilmiştir;

```
int* tekBoyutluDiziDondur() {
    /*... */
}
int** ikiBoyutluDiziDondur() {
    /*... */
}
```

Yerel değişkenlerin yığın bellekte (stack segment) tutulduğu ve fonksiyondan geri dönünce fonksiyon yerelindeki değişkenlerin kaybolduğu *Çağrı Kuralı*

Çağrı kuralı (calling convention), bir fonksiyon çağrısıyla karşılaşıldığında argümanların fonksiyona hangi sıraya göre iletileceğini belirtir. İki olasılık vardır; Birincisi argümanlar soldan başlanarak sağa

doğru fonksiyona geçirilir. İkincisi ise C dilinde kullanılır ve argümanlar sağdan başlanarak sola doğru fonksiyona geçirilir.

```
#include <iostream>
using namespace std;
int topla(int, int, int);
int main () {
    int toplam;
    toplam=topla(10,20,30);
    cout << "toplam:" << toplam << endl;
}
int topla(int x, int y, int z) {
    cout << "x:"<< x << " y:" << y  << " z:" << z<< endl;
    return x+y+z;
}
/* Program çalıştırıldığında:
x:10 y:20 z:30
toplam:60
*/
```

Yukarıdaki örnek incelendiğinde argümanların hangi sırada fonksiyona geçirildiğinin bir önemi yoktur. Ancak aşağıdaki verilen örnekteki durumu inceleyelim;

```
#include <iostream>
using namespace std;
int topla(int, int, int);
int main () {
    int a = 1,toplam;
    toplam = topla(a, ++a, a++);
    cout << "toplam:" << toplam << endl;
}
int topla(int x, int y, int z) {
    cout << "x:"<< x << " y:" << y  << " z:" << z<< endl;
    return x+y+z;
}
/* Program çalıştırıldığında:
x:3 y:3 z:1
toplam:7
*/
```

Bu kodun çıktısı `1 2 3` olarak beklenir ancak çağrı kuralından ötürü çıktı `3 3 1` olur. Bunun nedeni C++ dilinin çağrı kuralının sağdan sola olmasıdır. Bunu şöyle açıklayabiliriz;

6. Fonksiyona sağdan ilk argüman olarak a değişkeni değeri 1 geçirilir.
7. Ardından a++ ifadesi ile a değişkeni 2'ye yükseltilir.
8. Sonra ++a ifadesi ile a değişkeni 3'e yükseltilir.
9. Fonksiyona ikinci argüman olarak 3 geçirilir.
10. Fonksiyona son olarak a'nın son değeri olan 3 geçirilir.

Depolama Sınıfları başlığında anlatılmıştı. Bu nedenle bir fonksiyon içinde tanımlanan bir dizinin göstericisi geri döndürülecek ise `static` olarak tanımlanır.

```
#include <iostream>
#include <iomanip> //setw()
#include <ctime>
using namespace std;
#define BOYUT 10
int* rastgeleOgrenciNotlari( ) {
    static int  notlar[BOYUT];
    for (int i = 0; i < BOYUT; ++i)
        notlar[i] = rand()%101;
    return notlar;
}
```

```
void notlariYaz(int* pNotlar,int pUzunluk) {
    cout << "Öğrenci Notları:" << endl;
    for (int i = 0; i < pUzunluk; ++i)
        cout << setw(4) << pNotlar[i];
    return;
}
int main () {
   int *notlar;
   srand(time(nullptr));
   notlar = rastgeleOgrenciNotlari();
   notlariYaz(notlar,BOYUT);
}
/* Program Çıktısı:
Öğrenci Notları:
  30  31  13  40  41  37  34  83  93  76

...Program finished with exit code 0
*/
```

Dinamik Hafıza

C++ dilinde bir fonksiyon bloğu içerisinde tanımlanan tüm değişkenler yığın bellek (stack segment) üzerinde kaplayacaktır. Program çalıştığında bir değişkene belleği dinamik olarak tahsis etmek için öbek bellek (heap segment) kullanılır. Dinamik bellek tahsisi için new işlecini (new operator) kullanırız. Bu işleç, dinamik dizi için öbek bellekte yer tahsis edilip tahsis edilen bellek bölgesinin ilk baytının adresini göstericiye atar. Tahsis edilen bellek bölgesini serbest bırakmak için ise `delete` işleci (delete operator) kullanılır.

Aşağıda bir `float` değişkene dinamik olarak yer tahsisi örneği verilmiştir;

```
#include <iostream>
using namespace std;
int main(){
    float* ptrFloat=nullptr;
    ptrFloat=new float;
    if (ptrFloat!=nullptr) {
        *ptrFloat=4.5;
        cout << *ptrFloat << endl;
    } else
        cout << "Bellek tahsisi yapılamadı!" << endl;
    delete ptrFloat;
}
```

C++ dilinde diziler de dinamik olarak göstericiler yardımıyla oluşturulur. Dinamik bellek tahsisi, çalışma zamanı sırasında bellek tahsis etmemize olanak tanır. Bir dizideki dinamik tahsis, özellikle bir dizinin boyutu derleme zamanında bilinmediğinde ve çalışma zamanı sırasında belirtilmesi gerektiğinde faydalıdır. Bir diziyi dinamik olarak tahsis etmek için;

- Tahsis edilen dizinin ilk elemanının adresini depolayacak bir gösterici tanımlanır.
- Sonra, belirli bir veri tipindeki bir diziyi barındıracak dizi için `new` işleci ile bellek alanını tahsis edilir.
- Bu tahsisi yaparken, kaç öğe içerebileceğini belirten dizinin boyutunu belirtilir. Bu belirtilen boyut, tahsis edilecek bellek miktarını belirler.

Dizi uzunluğu tamsayı olacak şekilde dinamik dizi aşağıdaki gibi tanımlanır;

```
veritipi* diziGöztericiKimliği= new veritipi[diziUzunluğu];
```

Aşağıda çalıştırma anında dinamik olarak oluşturtulan bir dizi örneği verilmiştir;

```
#include <iostream>
using namespace std;
int main() {
```

```
    // Çalıştırma Anında Oluştutulacak Dizi Boyutu Soruluyor
    cout << "Dizinin Boyutunu Girin: ";
    int size;
    cin >> size;

    int* arr = new int[size]; // Diziye dinamik bellek (heap) tahsis ediliyor.

    for (int i = 0; i < size; i++) // Bir döngü ile dizi elemanlarına değer atanıyor
        arr[i] = i * 2;

    cout << "Dinamik Dizi Elemanları: " << endl;
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;

    delete[] arr; // dizi bellekten kaldırılıyor. Tahsis edilen bellek serbest bırakılıyor.
}
/*Program çalıştırıldığında:
Dizinin Boyutunu Girin: 12
Dinamik Dizi Elemanları:
0 2 4 6 8 10 12 14 16 18 20 22

...Program finished with exit code 0
*/
```

Çok boyutlu dizilere de dinamik olarak bellek tahsisi yapılabilir;

```
double** pvalue  = nullptr;   // Göstericiye ilk değer olarak nullptr veriliyor
pvalue  = new double [3][4];  // 3x4 buyutlu martis için yer tahsisi yapılıyor.
//...
delete[] pvalue; // tahsis edilen bellek serbest bırakılıyor
```

Çöp Toplama

Çöp toplama (garbage collection), dinamik bellek yönetimi söz konusu olduğunda önemli bir kavramdır. C++ dilinde geliştiriciler belleği manuel olarak yönetmelidir, bu hem güçlendirici hem de göz korkutucu olabilir. Otomatik çöp toplama özelliğine sahip dillerin aksine, C++ programcıların `new` ile belleği tahsis etmesini ve `delete` kullanarak ve tahsis edilmiş belleği serbest bırakmasını gerektirir.

C++'da bellek yönetimi yığın bellek (stack memory) ve öbek belleği (heap memory) belleği kavramları etrafında döner. Yığın belleği, yerel değişkenlerin depolandığı yerdir ve boyutu sınırlıdır. Bu belleğin boyutu derleyici tarafından belirlenir ve otomatik olarak yönetilir. Ancak öbek (heap) belleği, dinamik bellek tahsisinin gerçekleştiği yerdir. Burada, bellek çalışma zamanında tahsis edilir ve bu da daha fazla esneklik sağlar ancak aynı zamanda dikkatli bir yönetim gerektirir.

`new` İşleci kullandığınızda, öbek bellek üzerinde bir miktar bellek tahsis edilir. Bu bellek, `delete` işleci kullanarak açıkça serbest bırakana kadar tahsis edilmiş olarak kalır. Tahsis edilen belleği serbest bırakmamak, artık ihtiyaç duyulmayan belleğin tahsis edilmiş olarak kalmasına ve kaynakların verimsiz kullanımına yol açan bellek sızıntılarına (memory leak) yol açabilir. Buna karşılık, hala kullanımda olan belleği silerseniz, tanımlanmamış davranışa ve program çökmelerine yol açabilir.

C++ dilinde etkili çöp toplama için yığın (stack) ve öbek (heap) belleği arasındaki dengeyi anlamak esastır. Belleği düzgün bir şekilde yönetmek uygulama performansını artırabilir, bellek sızıntılarını önleyebilir ve genel kararlılığı iyileştirebilir.

C++ dilinde bellek yönetimi çoğunlukla manuel olarak programcı tarafından yapılır. Bu, programcıların belleği ayırma ve ayırmayı kaldırma sorumluluğunu üstlenmesi gerektiği anlamına gelir. `new` ve `delete` işleçlerini doğru kullanmak, uygulamanızın sorunsuz çalışmasını sağlamak için hayati önem taşır.

```
#include <iostream>
int main() {
```

```
    int* ptr = new int;
    *ptr = 61;

    std::cout << "Value: " << *ptr << std::endl;
    delete ptr;
}
```

Yukarıdaki örnekte `ptr` göstericisine öbek bellekte `new` ile yer ayrılmış, bellekteki değer `61` olarak değiştirilmiş ve `delete` ile bu bellek bölgesi serbest bırakılmıştır. Burada `delete` unutulmuş olsaydı, bellek ayrılmış olarak kalır ve bu da bellek sızıntısına yol açardı. Bu manuel yaklaşım, programcının dikkat ve disiplinini gerektirir çünkü bu ayrıntıların gözden kaçırılması önemli sonuçlara yol açabilir.

Manuel bellek yönetimi esneklik sağlarken, aynı zamanda birkaç tuzağı da beraberinde getirir. Yaygın bir sorun, tahsis edilen belleğin `delete` kullanılmasının unutulması ve dolayısıyla asla serbest bırakılmaması durumunda oluşan bellek sızıntılarıdır. Bir diğer tuzak da sarkan göstericilerdir (dangling pointers). Bu göstericiler, tahsisi kaldırılmış belleğe hala başvurduğunda meydana gelir. Bu tür belleğe erişim, tanımlanmamış davranışa, çökmelere veya veri bozulmasına yol açabilir;

```
#include <iostream>
int main() {
    int* ptr = new int(61);
    delete ptr;
    std::cout << *ptr << std::endl;
    return 0;
}
/* Program Çıktısı:
Segmentation fault
*/
```

Bu örnekte, `delete` işleci ile bellek serbest bırakıldıktan sonra, `ptr` sarkan gösterici haline gelir. Serbest bırakılmış belleğe erişmeye çalışmak bir segmentasyon hatasıyla (segmentation fault) sonuçlanır. Bu tür sorunları önlemek için, silme işleminden sonra işaretçileri geçersiz kılmak çok önemlidir.

C++'da manuel bellek yönetimiyle ilişkili yaygın tuzaklardan kaçınmak için izleyebileceğiniz birkaç iyi uygulama vardır:

- her zaman `new` ve `delete` işleçlerini eşleştirin: Belleği `new` kullanarak ayırdığınızda, `delete` ile onu serbest bıraktığınızdan emin olun. Bu, bellek sızıntılarını önlemeye yardımcı olur.
- Akıllı göstericiler kullanın: C++ 11 ile hayatımıza belleği otomatik olarak yöneten `std::unique_ptr` ve `std::shared_ptr` gibi akıllı göstericiler girdi. Bu göstericiler ihtiyaç duyulmadığında belleği otomatik olarak silerek bellek sızıntılarını ve sarkan göstericileri (dangling pointer) işaretçileri önlemeye yardımcı olurlar. *Akıllı Göstericiler* başlığını inceleyiniz.
- Göstericilere her zaman `nullptr` ile ilk değer ver: Bu uygulama sarkan göstericileri önlemeye yardımcı olur.

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<int> ptr = std::make_unique<int>(61);
    std::cout << "Value: " << *ptr << std::endl;
}
```

Referanslar

Bir değişken referans olarak tanımlandığında, var olan bir değişken için alternatif bir isim haline gelir. Bir değişken, bildirime '&' eklenerek referans olarak bildirilebilir. Yani, bir referans değişkenini başka bir değişkene referans görevi görebilen bir değişken türü olarak tanımlayabiliriz.

```
veri-tipi &referans-kimliği = varolan-bir-değişken;
```

Buradaki referans işleci bir değişkenin veya herhangi bir belleğin adresini belirtmek için kullanılır. Aşağıda buna ilişkin örnek bir program verilmiştir;

```
#include <iostream>
using namespace std;
int main(){
    int x = 10;
    int& ref = x; // ref ile var olan x değişkenine referans tanımlanmıştır.
    ref = 20; // x değişkeninin yeni değeri 20 olmuştur.
    cout << "x = " << x << endl; // x = 20
    x = 30; // x değişkeninin yeni değeri 30 olmuştur
    cout << "ref = " << ref << endl; // ref = 30
}
```

Fonksiyonlara parametre geçirme konusunda yaygın olarak kullanılan yöntemler değerle geçme (pass by value) ve referansla geçmedir (pass by reference).

- Değerle geçme, bir fonksiyonun parametrelerinin değerlerinin bir kopyası oluşturularak fonksiyon icra edilmesidir. Bu durumda fonksiyondan geri dönüldüğünde girdi olarak kullanılan değişkenler değişmez.
- Referansla geçme, değişkenlerin kendisi değil de referanslarının parametre olarak fonksiyona geçirilmesidir. Böylece parametre olarak kullanılan girdi değişkenlerinin fonksiyon içerisinde değiştirilmesine izin verilir. Çünkü değişkenin değeri değil adresi parametre olarak fonksiyona geçirilmiştir.

Göstericilerle bunu yapabiliriz ancak bu durumda kod okunabilirliğini azalır ve programcının hata yapma olasılığı artar.

```
#include <iostream>
using namespace std;
void degistir(int,int);
void degistir2(int&, int&);
int main(){
   int a = 10, b = 20;
   degistir(a, b);
   cout << "Değiştir Sonrası a:" << a << ", b:" << b << endl;
   //Çıktı: Değiştir Sonrası a:10, b:20
   degistir2(a, b);
   cout << "Değiştir2 Sonrası a:" << a << ", b:" << b << endl;
   //Çıktı: Değiştir2 Sonrası a:20, b:10
}
void degistir(int pX, int pY) {
   int z = pX;
   pX=pY;
   pY=z;
}
void degistir2(int& pX, int& pY){
   int z = pX;
   pX=pY;
   pY=z;
}
```

Referanslar ile göstericileri karşılaştırırsak;

- Göstericiler herhangi bir zamanda NULL gösterici (NULL pointer) olabildiği gibi başka bir nesneyi işaret edilebilir. Bir referans tanımlandığı anda bir değişken ile ilk değer verilmelidir. Göstericiler tanımlandıktan sonra herhangi bir zamanda ilk değer verilebilir ya da gösterdiği değişken değiştirilebilir.
- Hiçbir değişkeni işaret etmeyen NULL referanslarınız olamaz! Her zaman bir referansın meşru bir değişkene bağlı olduğunu varsayabilmelisiniz.

– Bir referans imal edilen bir nesneyi işaret ediyorsa, başka bir nesneyi göstermek üzere referansı değiştirilemez!

Göstericiler yerine referanslar kullanmak kodumuzun okunmasını ve bakımını daha kolay hale getirilebilir. Bir C++ fonksiyonu, bir gösterici döndürdüğü gibi bir referansı da döndürebilir.

Bir fonksiyon bir referans döndürdüğünde, dönüş değerine örtük bir gösterici döndürür. Böylece atama ifadesinin sol tarafında bir fonksiyon kullanılabilir hale gelir.

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

float dizi[5] = {10.0, 20.0, 30.0, 40.0, 50.0};
float& indistekiDeger( int indis ) {
   return dizi[indis];
}
int main () {

   cout << "Değiştirmeden Önce Dizi" << endl;
   for ( int i = 0; i < 5; i++ ) {
      cout << "dizi[" << i << "] = ";
      cout << dizi[i] << endl;
   }
   indistekiDeger(1) = 100.0;
   indistekiDeger(3) = 200.0;
   cout << "Değiştirmeden Sonra Dizi" << endl;
   for ( int i = 0; i < 5; i++ ) {
      cout << "dizi[" << i << "] = ";
      cout << dizi[i] << endl;
   }
}
/* Program Çıktısı:
Değiştirmeden Önce Dizi
dizi[0] = 10
dizi[1] = 20
dizi[2] = 30
dizi[3] = 40
dizi[4] = 50
Değiştirmeden Sonra Dizi
dizi[0] = 10
dizi[1] = 100
dizi[2] = 30
dizi[3] = 200
dizi[4] = 50

...Program finished with exit code 0
*/
```

Bir referans döndürürken, referans alınan nesnenin kapsam (scope) dışına çıkmamasına dikkat edilmesi gerekir;

```
int& fonksiyon() {
   int q;
   // return q; /* Derleme hatası olur.
                   Çünkü fonksiyon dışında bu değişken bellekte yoktur.*/
   static int x;
   return x; /* x değişkeni tüm program süresince hayatta olduğundan
                bu kod güvenlidir. */
}
```