Gösterici (pointer) değişkenlerin bildiriminde const anahtar sözcüğünün kullanımı

C dilinde **const** anahtar sözcüğünün bir pointer değişkenin tanımında kullanıldığı yere bağlı olarak verdiği anlam değişmektedir.

const anahtar sözcüğünün * (asterisk) atomundan sonra kullanılması

Aşağıdaki koda bakalım:

```
int x = 10;
int *const ptr = &x;
```

İngilizcede bu şekilde tanımlanmış gösterici değişkenlere **"const pointer"** denmektedir. C dilinde yaygın olarak kullanılmasa da C++'ta böyle pointer değişkenler için "*top level const*" terimi de kullanılmaktadır. Bu tür gösterici değişkenler için kurs boyunca "*const gösterici*" ya da bu durumu özellikle vurgulamak için "*kendisi const gösterici*" terimini kullanacağım.

Buradaki belirtilen *ptr* değişkeninin değerinin hayatı boyunca değişmeyecek olduğudur. Başka bir deyişle, *ptr* değişkeni hayatı boyunca *x* değişkenini gösterecektir. Bu durumda derleyici, *ptr* değişkeninin değerini değiştirmeye yönelik kodları geçersiz kabul etmekle yükümlüdür. Yani (yanlışlıkla) *ptr* değişkenine başka bir nesnenin adresini atarsak geçersiz kod (sentaks hatası) oluşur. Buradaki taahhüdümüz (sözümüz) *ptr* yoluyla erişilecek nesnenin, yani gösterilen nesnenin *(pointee)* yani

```
*ptr
```

ifadesine karşılık gelen nesnenin değerinin değerinin değiştirilmeyeceği değildir, *ptr*'nin kendi değerinin değiştirilemeyeceğidir. **ptr* nesnesine **(pointee)** yani *ptr*'nin gösterdiği nesneye atama yapılabilir. Bu konuda bir söz verilmemiştir. Aşağıdaki koda bakalım:

```
int main()
{
 int x = 10;
 int y = 20;

 int *const ptr = &x;  //ptr is a const pointer
 //ptr = &y;  //gecersiz
 *ptr = 90;   //geçerli
 //...
}
```

Peki, neden kendisi const pointer değişkenleri kullanırız?

- Kodun lojik yapısı gösterici değişkenin değerinin hiç değişmemesini gerektirmektedir. Gösterici değişkene başka bir değerin atanmasının (yani başka bir nesnenin adresinin atanmasının) mantıksal bir hata olduğunu düşünelim.

```
int g1 = 10;
int g2 = 20;

void func()
{
 int *ptr = &g1; //ptr'nin kendi hayatı boyunca g nesnesini göstermesi gerekiyor olsun.
  //...
 ptr = &g2; //Bu atama yanlışlıkla yapılmış olsa da geçerli
}
```

ptr'nin hayatı boyunca *g1* değişkenini göstermesi gereksin. Eğer gösterici değişkeni yukarıdaki gibi tanımlarsak ve bu gösterici değişkene başka bir nesnenin adresini atarsak, bu atama lojik bir hata olmasına karşın kod geçerlidir. Böyle durumlarda *kendisi const pointer değişkenin* kullanılması kodlama hatası yapma riskini düşürmektedir.

- Kodu okuyana *ptr*'nin değerinin değişmeyeceğini bildirmek birçok durumda kodun okunmasını kolaylaştırabilir.
- Bazı durumlarda derleyicinin *ptr* değişkeninin değerinin değişmeyeceğini bilemesi derleyiciye daha iyi bir optimizasyon *(eniyileme)* olanağı vermektedir.

const anahtar sözcüğünün bildirimde * (asterisk) atomundan önce kullanılması

Bu kez aşağıdaki koda bakalım:

```
int x = 10;
const int *ptr = &x;
```

ptr değişkeninin bildiriminde **const** anahtar sözcüğü ** (asterisk)* atomundan önce kullanılıyor. İngilizcede bu şekilde tanımlanmış gösterici değişkenlere **"pointer to const"** denmektedir. Örneğin yukarıdaki koddaki ptr için **"ptr is a pointer to const int"** diyebiliriz. C dilinde yaygın olarak kullanılmasa da C++ dilinde böyle gösterici değişkenler için **"low level const"** terimi de kullanılmaktadır. Bu tür gösterici değişkenler için kurs boyunca "const nesne göstericisi" ya da bu durumu özellikle vurgulamak için **"gösterdiği nesne const olan gösterici"** terimlerini kullanacağım.

Yukarıdaki tanımlamada tür belirten *int* anahtar sözcüğü ile **const** anahtar sözcüğünün bildirimde yer değiştirmesi bir anlam farklılığı oluşturmaz. Yani

```
const int *ptr = &x;
```

ile

```
int const *ptr = &x;
```

tanımlamaları tamamen aynı anlamdadır. Hangi biçimi tercih ettiğimiz kullandığımız kodlama konvensiyonları *(coding conventions)* ile ilgilidir. Burada belirtilen, *ptr* değişkeninin gösterdiği (ve ileride gösterebileceği) nesneleri salt okuma *(access)* amaçlı gösteriyor olmasıdır. Başka bir deyişle, **ptr* ifadesine karşılık gelen nesneyi, *ptr* yoluyla (aracılığı ile) değiştirmeme sözü vermiş oluyoruz. Bu durumda derleyici, **ptr* nesnesinin değerini değiştirmeye yönelik kodları geçersiz kabul etmekle yükümlüdür. Yani (yanlışlıkla) *ptr *yoluyla *ptr*'nin gösterdiği nesneye *(pointee)* bir atama yaparsak geçersiz kod (sentaks hatası) oluşur. Buradaki taahhüdümüz (sözümüz) *ptr*'nin değerini değiştirmemek değildir. *ptr*'nin değerini değiştirmemiz yani ona yeni bir değer atmamamız geçerlidir. Aşağıdaki koda bakalım:

```
int main()
{
 int x = 10;
 int y = 20;

 const int *ptr = &x; //ptr is a pointer to constr int
 //int * const ptr = &x;  //ptr is a pointer to constr int
 //*ptr = 90;   //geçersiz
 ptr = &y; //geçerli
 //...
}
```

Eğer yazdığınız kodda **const** anahtar sözcüğünü nereye yazacağınız konusunda tereddütünüz varsa (kuralları unutmuşsanız) her zaman şu cümleyi hatırlayın:

> const neden önce geliyorsa const olan odur

```
int * const p = &x;
```

const anahtar sözcüğü *p*'den önce geliyor. **const** olan *p*'nin kendisi. *p*'ye atama yaparsak sentaks hatası olacak:

```
const int *p = &x;
```

const anahtar sözcüğü **p*'den önce geliyor. const olan **p*, yani *p*'nin gösterdiği nesne. **p* ifadesine atama yaparsak sentaks hatası oluşacak.

```
int const *p = &x;
```

const anahtar sözcüğü **p*'den önce geliyor. **cons** olan **p*, yani *p*'nin gösterdiği nesne. **p* ifadesine atama yaparsak sentaks hatası oluşacak.

const anahtar sözcüğü iki konumda birden de kullanılabilir. Bu durumda her iki **const** anahtar sözcüğünün verdiği anlam da korunur:

```
int x = 10;
const int *const ptr = &x;
```

Yukarıdaki gibi tanımlanan bir gösterici değişkene İngilizcede **"const pointer to const int"** denmektedir. Bu

durumda verdiğimiz söz hem *ptr*'nin hem de **ptr*'nin değerini değiştirmemektir. Bir başka deyişle hem *ptr*'ye başka bir adres atamak hem de **ptr* ifadesine başka bir değer atamak mantıksal hata ise, gösterici değişkeni bu şekilde tanımlamalıyız. Aşağıdaki koda bakalım:

```
int main()
{
 int x = 10;
 int y = 20;

 const int *const ptr = &x; //ptr is a const pointer to const int
 //*ptr = 90;   //geçersiz
 //ptr = &y; //geçersiz
 //...

}
```

Gösterici değişkenlere ilişkin bu semantik yapı bizi daha çok fonksiyonların tanımlanması ve fonksiyonların çağırılması durumunda önem kazanıyor:

Parametre değişkeni gösterici olan fonksiyonlar

T bir tür olsun ve ismi func olan bir fonksiyon aşağıdaki şekilde bildirilmiş olsun:

```
void func(T *p);
```

Fonksiyonun başka parametre değişkenleri de olabilir. Diğer parametre değişkenlerini şimdilik görmezden geliyoruz. C'de böyle bir fonksiyon arayüzünün (interface) anlamı şudur: Fonksiyon, adresini istediği nesnenin değerini değiştirmek amacıyla (onu set etmek için) nesneye erişecek. Böyle bir fonksiyona yapılan çağrı ile fonksiyona bir nesnenin (ya da bir dizinin) adresini gönderdiğimizde nesnemiz değişecek, nesnemize yeni değer ya da değerler (dizi ise) yazılacak. Böyle fonksiyonlara İngilizce'de duruma göre şöyle isimler yakıştırılıyor: *set function*, *setter*, *mutator (değiştiren anlamında)*. Şimdi neden böyle fonksiyonların tanımlandığını konuşalım:

- Fonksiyon, bize hesapladığı bir değeri iletmek amacıyla tanımlanmış olabilir. Yani fonksiyon hesapladığı değeri, geri dönüş değeri *(return value)* ile bize iletmek yerine bu değeri bizden adresini istediği nesneye yazıyor olabilir. Bu noktada önemli olan sorulardan biri şu: Fonksiyon neden hesapladığı değeri geri dönüş değeri yöntemi ile değil de bu şekilde iletmeyi tercih etmiş olsun? Bu soruya daha sonra cevap vereceğim. Hadi böyle fonksiyonlara bir örnek verelim:

```
void get_random_date(struct Date *p)
```

Örneğimizde, *struct Date* isimli türden nesneler (türün etiket isminin de ima ettiği gibi) tarih verileri tutuyor olsun. Yukarıdaki işlevin amacı, müşteri koda rastgele oluşturulmuş bir tarihi iletmek. Fonksiyon bu değeri pek ala geri dönüş değeri ile de kendisini çağıran koda iletebilirdi. Böyle bir fonksiyonu oluşturduğumuz bir *struct Date* nesnesinin adresi ile çağırmamız gerekiyor. Böylece (adresini) gönderdiğimiz nesne fonksiyonun kodu çalıştıktan sonra rastgele bir tarih değeri tutuyor olacak.

```
{
 struct Date xdate;
 get_random_date(&xdate);
 //...
}
```

Böyle fonksiyonların bu biçimdeki parametre değişkenleri İngilizcede yaygın olarak "*out parameter*" olarak isimlendiriliyor *(output parameter anlamında)*. Çünkü buradaki amaç, fonksiyonun hesapladığı (ürettiği) değer ya da değerleri kendisini çağıran koda iletmesi.

- Fonksiyon yapmakla yükümlü iş gereği bizden adresini aldığı nesneyi değiştiriyor olabilir. Bir fonksiyonun bir başka yerel kapsamdaki *(local scope)* nesneyi değiştirebilmesinin başka bir yolu yok zaten. Örneğin *T* türünden iki nesneyi takas *(swap)* edecek bir fonksiyon şu parametrik yapıda olacaktır:

```
void swap(T *p1, T *p2);
```

Bu fonksiyona adreslerini gönderdiğimiz *T* türünden iki nesne takas edilecek. Diziler ya da yazılar üzerinde, dizilerin ya da yazıların öğelerini değiştirmeye yönelik işlemleri gerçekleştirecek fonksiyonlar böyledir değil mi? C'nin standart kütüphanesinden bir örnek verelim:

```
char *strcpy(char *pdest, const char *psource);
```

strcpy fonksiyonu *psource* adresindeki yazıyı *pdest* adresine kopyalar.

- Fonksiyon bizden bir yapı *(structure)* nesnesinin adresini istiyor bu adresteki nesnenin bazı öğelerinden *(member)* yapacağı işlemlerde kullanacağı değerleri *(input)* okuyor aynı zamanda yapı nesnesinin bazı öğelerine ise hesapladığı değerleri yazıyor olabilir. Böyle fonksiyonların gösterici parametre değişkenlerine İngilizcede yaygın olarak "*in-out parameter*" *(input - output anlamında) *deniyor. Standart kütüphanenin *time.h* başlık dosyasında bildirilen mktime isimli fonksiyon buna bir örnek olarak verilebilir:

```
time_t mktime (struct tm * timeptr);
```

Standart *mktime* fonksiyonu kendisini çağıracak koddan *struct tm* türünden bir nesnenin adresini istiyor. Adresini istediği nesnenin öğelerinden hesaplamada kullanacağı bazı değerleri alarak kullanacak, ancak duruma göre aynı nesnenin bazı öğelerine de hesapladığı değerleri yazacak.

Şimdi de diğer arayüzü inceleyelim. Yine *T* bir tür olsun ve ismi *func* olan bir fonksiyon aşağıdaki şekilde bildirilmiş olsun:

```
void func(const T *p);
```

Bu bildirim ile aşağıdaki bildirim arasında anlamsal bir fark olmadığını yine hatırlatayım:

```
void func(const T *ptr);
```

C'de böyle bir fonksiyon arayüzünün *(interface)* anlamı şudur: Fonksiyon, adresini istediği nesneye, onun değerini okumak amacıyla yani bu değeri yapacağı işlerde kullanmak için erişecek. Böyle bir fonksiyona çağrı yaptığımızda (adresini) gönderdiğimiz nesnenin değerinin fonksiyon tarafından değiştirilmeyeceğinden emin olabiliriz. Böyle fonksiyonlara İngilizce'de duruma göre şöyle isimler yakıştırılıyor: *get function, getter, accessor (erişen anlamında)*. Şimdi de neden böyle fonksiyonların tanımlandığını konuşalım...

Yukarıdaki gibi bir fonksiyon yerine doğrudan değerle çağrılan bir fonksiyon *(call by value)* oluşturabilirdik:

```
void func(T x);
```

Bu durumda bu fonksiyona yapılan her çağrı, argüman olarak gönderilen değerin fonksiyonun parametre değişkenine kopyalanması sonucunu doğuracaktı. Bu kopyalama maliyeti, *T* türünden bir nesnenin bellekteki yerinin *(storage)* büyüklüğü ile doğru orantılı olarak artacaktı. Örneğin sistemimizdeki kelime uzunluğu *4 byte*, *T* türü için de *sizeof* değeri *64* olsun. Bu durumda her fonksiyon çağrısı *64 byte*'lık bir bellek bloğunun kopyalanması sonucunu doğuracaktı.

gösterici dizileri ve const anahtar sözcüğü

Gösterici dizilerinin tanımlanmasında yine *const* anahtar sözcüğünün kullanıldığı yere bağlı olarak anlam değişir. Aşağıdaki koda bakalım:

```
int x = 10, y = 20, z = 30;

int *const pa1[] = { &x, &y, &z };
const int *pa2[] = { &x, &y, &z };
```

pa1 dizisinin tanımında const anahtar sözcüğü '* (asterisk) atomundan sonra ve dizinin isminden önce yazılmış. Bu durumda **const** olan dizinin kendisidir. Yani bu durumda dizinin öğelerini değiştirmeme sözü vermiş oluyoruz. Dizinin öğelerinin değiştirilmesi yani onlara atama yapılması derleyici tarafından sentaks hatası olarak işaretlenecek. Ancak bu durumda dizinin öğeleri olan göstericileri içerik *(dereferencing)* operatörünün operandı yaparak değiştirmemizde bir engel yok.

```
int main()
{
 int t = 0;
 pa1[0] = &t; //geçersiz
 *pa1[0] = 50; //geçerli
 //...
}
```

pa2 dizisinde ise durum farklı. Burada dizinin elemanları *const int** türünden. Dizinin kendisi **const** değil. Yani dizinin elemanlarının değerlerini değiştirebiliriz. Verdiğimiz söz dizinin öğeleri olan göstericiler ile eriştiğimiz nesneleri salt okuma amaçlı kullanmak. Aşağıdaki koda bakalım:

```
int main()
{
 int t = 0;
 pa2[0] = &t; //geçerli
 *pa2[0] = 50; //geçersiz
 //...
}
```

Burada da **const** anahtar sözcüğü iki yerde birden kullanılabilir:

```
int x = 10, y = 20, z = 30;

const int *const pa[] = { &x, &y, &z };

int main()
{
 int t = 0;
 pa[0] = &t; //gecersiz
 *pa[0] = 50; //gecersiz
 //...
}
```