Temel Veri Yapıları

(Veri Yapıları 1. Bölüm)

Kaan Aslan
10 Haziran 2013

1. Giriş

Programlarımızda tanımladığımız nesneler ya tek parçadan ya da birden fazla parçadan oluşurlar. Tek parçadan oluşan nesnelerin türlerine tekil türler, birden fazla parçadan oluşan nesnelerin türlerine ise bileşik türler denilmektedir. Örneğin, *int* türü tekil bir türdür. Çünkü *int* türden bir nesne tek parçadan oluşmaktadır. Halbuki diziler bileşik türlerdir. Bir dizi türünden nesne tanımladığımızda o nesne kendi içerisinde birden fazla parçadan oluşur. Örneğin:

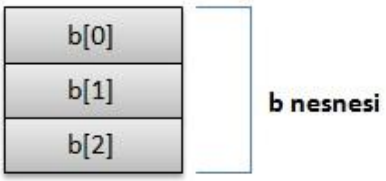
```
int a;          /* a, int türden */
```

Burada *a int* türden tekil bir nesnedir:

Fakat örneğin:

```
int b[3];       /* b, int[3] türünden */
```

Burada *b* kendi içerisinde 3 tane nesneden oluşan bileşik bir nesnedir:

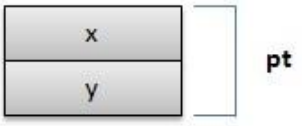


Benzer biçimde:

```
struct POINT {
    int x, y;
};

struct POINT pt;
```

Burada ekrandaki bir noktayı temsil eden *POINT* yapısı iki parçadan oluşmaktadır: *x* ve *y* parçaları:



Bileşik türler için *İngilizce* genellikle *aggregate types* terimi kullanılmaktadır. Diziler, yapılar, birlikler ve sınıflar bileşik türlerdir.

Belli bir amaca yönelik olarak organize edilmiş bir grup nesnenin oluşturduğu topluluğa veri yapısı (*data structure*) denilmektedir. Veri yapısı denildiğinde aklımıza bir grup nesne gelir. Bazı veri yapıları çok gereksinim duyulan bazı olguların temsil edilmesinde kullanılır. Bunlara *temel veri yapıları* diyeceğiz. Temel veri yapıları genellikle programlama dillerinde, dile entegre edilmiş bir biçimde bulundurulmakta ve dilin sentaksıyla desteklenmektedir. Örneğin diziler (*arrays*), yapılar (*structures*), birlikler (*unions*), sınıflar (*classes*) temel veri yapılarıdır ve pek çok programlama dili bunları dilin bir parçası olarak bünyesinde bulundurmaktadır. Diğer veri yapıları ise dil tarafından doğrudan desteklenmeseler de temel veri yapıları kullanılarak prosedürel teknikte fonksiyonlarla, nesne yönelimli teknikte sınıflarla gerçekleştirilebilirler. Örneğin bağlı listeler (*linked lists*) pek çok dilde programlama dili tarafından doğrudan desteklenmemektedir. Fakat yapılar kullanılarak birtakım fonksiyonlarla bağlı listeler oluşturulabilir.

Pek çok programlama dili bir sentaks ve semantik sunmanın yanı sıra standart bir kütüphaneye de sahiptir. Örneğin *C* Programlama Dilinin fonksiyonlardan ve makrolardan oluşan standart bir kütüphanesi vardır. *C++* hem *C*'nin standart kütüphanesini destekler hem de *template* tabanlı ayrı ve geniş bir sınıf kütüphanesine de sahiptir. Benzer biçimde *C#* ve *Java* ile çalışırken bu dillerin kullanıldığı ortamların sunmuş olduğu geniş sınıf kütüphanelerinden faydalanırız. İşte dil tarafından doğrudan desteklenmeyen pek çok veri yapısı sözünü ettiğimiz bu kütüphanelerde hazır olarak bulundurulmaktadır. Biz bu ortamlarda çalışırken çoğu kez gereksinim duyduğumuz veri yapısını sıfırdan oluşturmak yerine bize sunulan hazır fonksiyonları ya da sınıfları kullanırız. Örneğin dinamik büyütülen diziler *C++*'ın standart kütüphanesinde *vector* isimli sınıfla, *.NET*'in ve *Java* platformlarının kütüphanelerinde de *ArrayList* isimli sınıfla temsil edilmektedir. Bu ortamlarda çalışan programcılar dinamik büyütülen dizilere gereksinim duyduklarında doğrudan bu sınıfları kullanabilirler.

Veri yapıları konusunda çok karşılaşılan diğer bir kavram da *Soyut Veri Türleri*'dir *(Abstract Data Types)*. Bu kavram veri yapısını oluşturan veriler ile onlar üzerinde işlem yapan fonksiyonlardan oluşan organizasyonu belirtir. Buradaki soyutluk "ayrıntıların göz ardı edilmesi ve dikkatin veri yapısı üzerinde işlem yapan fonksiyonlara çevrilmesi" sürecini anlatmaktadır. Böylelikle soyut veri türlerini kullanan programcılar onların nasıl gerçekleştirildiği hakkında ayrıntılı bir bilgiye sahip olmak zorunda kalmazlar. Soyut veri türleri nesne yönelimli dillerde tipik olarak sınıflarla gerçekleştirilirler.

2. Temel Veri Yapıları

Bu bölümde pek çok programlama dilinde sentaks tarafından doğrudan desteklenen dizi, yapı, birlik ve sınıflar üzerinde durulacaktır.

2.1. Diziler (Arrays)

Elemanları aynı türden olan ve bellekte ardışıl bir biçimde bulunan veri yapılarına dizi denir. Tanımda da belirtildiği gibi, diziyi dizi yapan iki önemli özellik vardır:

1) Dizi tipik olarak birden fazla elemandan oluşur ve bu elemanlar aynı türdendir.

2) Dizi elemanları bellekte ardışıl bir biçimde tutulurlar.

Dizi elemanlarına *O(1)* karmaşıklıkta yani çok hızlı bir biçimde erişilir. Erişim dizinin tüm elemanlarına aynı sürede yapılmaktadır. (Bu tür erişim sistemlerine genellikle rastgele erişimli (*random access*) sistemler denilmektedir.) Şüphesiz dizi elemanlarına çok hızlı erişilmesi onların belleğe ardışıl yerleştirilmelerinden kaynaklanmaktadır.

Programlama dillerinde diziler tür ve uzunluk belirtilerek belli bir sentaksla oluşturulurlar. Örneğin *C* ve *C++*'ta dizilerin tanımlanması aşağıdaki gibi bir sentaksla yapılmaktadır:

```
int a[10];
```

C ve *C++*'ta dizilerin uzunluları sabit ifadesi (*constant expression*) biçiminde verilmek zorundadır. Çünkü derleyici dizinin uzunluğunu derleme zamanı sırasında belirlemek ister [1]. Bu dillerde eğer dizilerin gerçek uzunluğu programın çalışma zamanı sırasında belirleniyorsa bu durumda dizinin de programın çalışma zamanı sırasında dinamik olarak yaratılması gerekir. Örneğin *C*'de *malloc* ve *calloc* gibi dinamik bellek fonksiyonları *heap* denilen alanda ardışıl *byte* tahsisatı yaparlar. Elde edilen alanın adresi bir göstericiye atanırsa o bölge bir dizi gibi kullanılabilir. Örneğin:

```
int *pi;
...
pi = (int *) malloc(sizeof(int) * 10);
```

C# ve *Java*'da diziler de sınıfsal bir biçimde temsil edilmişlerdir. Bu dillerde diziler *new* operatörü ile dinamik olarak yaratılmak zorundadır. Örneğin:

```
int[] a;
a = new int[10];
```

Pekiyi dizilere neden gereksinim duyulmaktadır? Dizilerin en önemli faydası bir döngü içerisinde bir indis yardımıyla elemanları gözden geçirebilmemizi sağlamasıdır. Örneğin diziler olmasaydı *int* türden 100 tane nesneyi nasıl tanımlayabileceğinizi ve bunların hepsine nasıl değer atayacağınızı bir düşünün. Oysa diziler sayesinde tek bir döngü ile tüm elemanları gözden geçirebiliyoruz. Örneğin, bir dizinin elemanlarının toplamını bulan kod şöyle olabilir:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int i;
int total;

total = 0;
for (i = 0; i < 10; ++i)
    total += a[i];
```

Dizi elemanlarının bellekte ardışıl bir biçimde bulunması onları yalnızca başlangıç adresleri yoluyla fonksiyonlara geçirebilmemizi mümkün hale getirir. Örneğin:

```
int GetTotal(const int *pi, int size)
{
    int k;
    int total;

    total = 0;
    for (k = 0; k < size; ++k)
        total += pi[k];

    return total;

}

/*...*/

int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int result;
result = GetTotal(a, 10);
```

C ve *C++*'ta dizi isimleri o dizilerin bellekteki başlangıcına ilişkin adres bilgisi belirtirler. Başka bir deyişle dizi isimleri (yani dizi nesneleri) işleme sokulduğunda derleyici tarafından otomatik olarak adres bilgisine dönüştürülmektedir. Yani biz programımızda bir dizi ismini kullandığımızda aslında o dizinin ilk elemanının

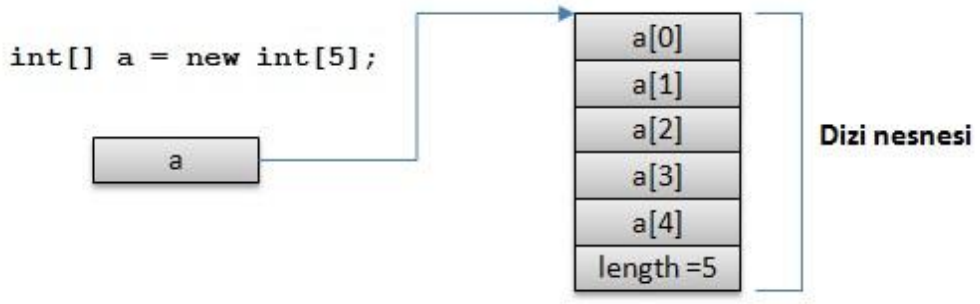
adresini kullanıyor gibi oluruz. Yukarıdaki örneğimizde *Foo* fonksiyonuna biraz daha genellik katmak için uzunluk parametresi de ekledik. *Java* ve *C#*'ta diziler zaten sınıfsal bir biçimde temsil edildiklerinden dolayı onların uzunluğu bu sınıfsal temsilin içerisinde zaten belirtilmektedir. Dolayısıyla bu dillerde dizileri parametre olarak alan metotlara ayrıca uzunluk parametresi eklemeye gerek yoktur. *C#*'ta *Length property*'si yoluyla, *Java*'da da *length* elemanı yoluyla dizi uzunlukları elde edilebilir. Örneğin:

```
public static int GetTotal(int[] a)
{
    int total;

    total = 0;
    for (int i = 0; i < a.Length; ++i)   // C#'ta Length, Java'da length
        total += a[i];

    return total;
}
```

C# ve *Java*'da dizi isimleri dizi nesnelerinin adreslerine ilişkin referans belirtmektedir. Bu dillerde bir dizi referansını şöyle düşünebiliriz:

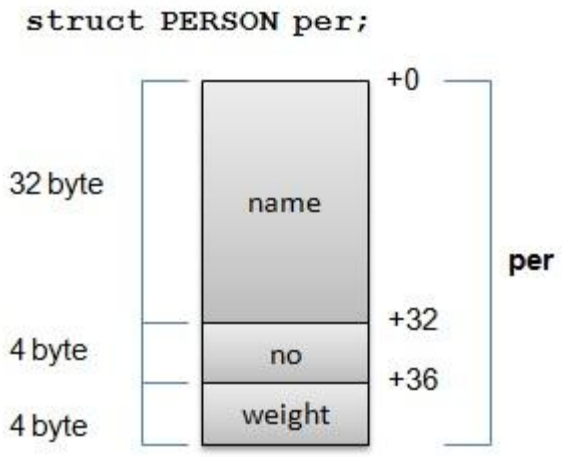


2.2. Yapılar (Structures)

Elemanları farklı türlerden olabilen ve bellekte ardışıl bir biçimde bulunan veri yapılarına yapı (*structure*) denilmektedir. Yine buradaki ardışıllık yapının başlangıç adresinin bilinmesi durumunda onun tüm elemanlarına sabit zamanlı, yani *O(1)* karmaşıklıkta erişilmesini sağlar. Öyle ki, eğer biz bir yapı nesnesinin başlangıç adresini biliyorsak onun herhangi bir elemanına tek bir makine komutuyla erişebiliriz. Yapı elemanları farklı türlerden olabildiğine göre hangi elemanın hangi türden olduğunun da bir biçimde derleyiciye açıklanması gerekmektedir. Yapı elemanlarının türlerinin ve isimlerinin derleyiciye tanıtılması işlemine yapı bildirimi denilmektedir. Örneğin *C/C++*'ta bir yapı aşağıdaki gibi bildirilir:

```
struct PERSON  {
    char name[32];
    int no;
    float weight;
};
```

Bu bildirimle derleyici bileşik yapı nesnesinin hangi parçalarının hangi türden olduğunu ve *offset*'ten başladığını anlar.



Örneğin *per.no* ya da *per.weight* gibi ifadelerle yapının elemanlarına erişilmesi tek bir makina komutuyla yapılabilmektedir. Benzer biçimde *p* bir yapı nesnesinin adresini belirtiyor olsun. *p->no* ve *p->weight* gibi erişimler de yapı elemanlarının ardışıl olmasından dolayı birkaç makina komutuyla gerçekleştirilebilmektedir. Pekiyi yapıların bize sağladığı yararlar nelerdir?

1) Yapılar gerçek bir nesnenin ya da kavramın birbirleriyle ilişkili birtakım özelliklerini temsil etmek için mantıksal bir küme oluştururlar. Örneğin bu sayede biz bir tarih bilgisini *DATE* isimli bir yapıyla, bir noktayı *POINT* isimli bir yapıyla temsil edebiliriz. Bu tür temsiller soyutlamayı artırarak algılamayı kolaylaştırmaktadır.

2) Yapılar farklı türlerden çok sayıda bilgiyi fonksiyonlara tek bir parametre olarak geçirmemize olanak sağlarlar. Örneğin eğer yapı kavramı olmasaydı yukarıdaki *PERSON* yapısının elemanlarını fonksiyonlara tek tek geçirmek zorunda kalırdık:

```
void Foo(char *name, int no, float weight);
...
Foo("Ahmet Kaya Aslan", 123, 78.3);
```

Halbuki yapılar sayesinde tüm bu bilgileri tek bir parametreyle fonksiyona aktarabiliriz:

```
void Foo(const struct PERSON *per);
...
struct PERSON per = {"Ahmet Kaya Aslan", 123, 78.3};
Foo(&per);
```

Aktarım yapı nesnesinin başlangıç adresiyle yapıldığı için, yapı ne kadar büyük olursa olsun gerçekte fonksiyona aktarılan yalnızca bir adres bilgisidir (32 bit sistemlerde 4 byte, 64 bit sistemlerde 8 byte).

3) Fonksiyonların tek bir geri dönüş değeri vardır. Eğer biz bir fonksiyondan birden fazla değer elde etmek istiyorsak ona bir yapı nesnesinin adresini göndererek onun o yapı nesnesinin içini doldurmasını sağlayabiliriz. Örneğin:

```
void Foo(const struct PERSON *per);
...
struct PERSON per;
Foo(&per);
```

Bu noktada yapılarla ilgili önemli bir anımsatma yapmak istiyoruz: *C/C++*'ta yapı elemanları derleyici tarafından aralarında boşluklar olacak biçimde belleğe yerleştirilebilmektedir. Yapı elemanlarının ve genel olarak nesnelerin belirli sayının katlarında olacak biçimde belleğe yerleştirilmelerine hizalama (*alignment*)

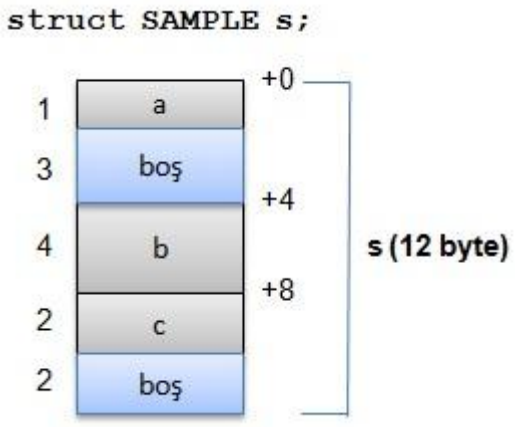
deniyor. Pek çok modern işlemci, eğer nesneler bellekte belli sayının katlarında bulunuyorsa (örneğin 32 bit işlemcilerde 4'ün katlarında) onlara daha hızlı erişebilmektedir. Örneğin, 32 bit Intel işlemcilerinde:

```
MOV EAX, [XXXXXXXX]
```

gibi bir makina komutu eğer XXXXXXXX adresi 4'ün katlarındaysa daha hızlı çalışmaktadır. Varsayılan hizalama pek çok derleyicide *DWORD* yani 4'ün katları biçimindedir. Şimdi hizalamanın etkisini aşağıdaki gibi bir yapıyla gösterelim:

```
struct SAMPLE {
    char a;
    int b;
    short c;
};
```

4 *byte*'ın (*DWORD)* katlarına göre yapılan hizalamada bir *SAMPLE* nesnesi belleğe şöyle yerleştirilecektir:



Görüldüğü gibi derleyici yapı elemanlarını 4'ün katlarına yerleştirebilmek için elemanlar arasına da boşluklar bırakabilmektedir.

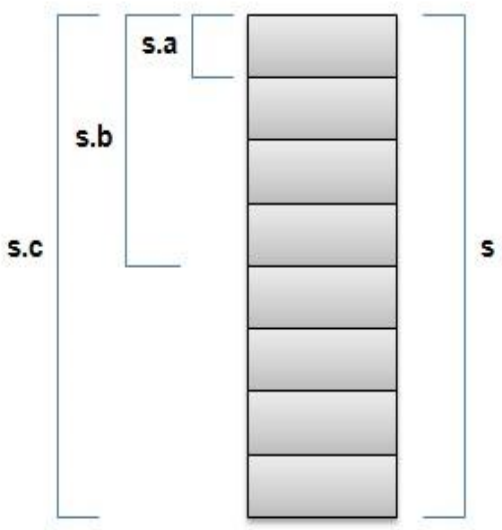
Java'da *C/C++*'taki gibi bir yapı kavramı yoktur fakat *C#*'ta vardır. *C#*'ta yapılar kategori olarak değer türlerine (*value types*) ilişkindir ve genel kullanımı *C/C++*'takine oldukça benzemektedir.

2.3. Birlikler (Unions)

Elemanların çakışık olarak yerleştirildiği veri yapılarına birlik (*union*) denilmektedir. Birlikler *C/C++*'ta yapılara benzer bir sentaksla bildirilirler. Örneğin:

```
union SAMPLE {
    char a;
    int a;
    double c;
};
```

```
union SAMPLE s;
```

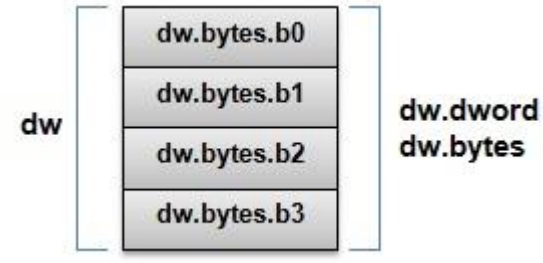


Bir birlik için birliğin en büyük elemanı kadar yer ayrılır. Yukarıdaki örnekte birliğin en büyük elemanı *double* olduğundan *s* isimli birlik nesnesi için toplam 8 byte yer ayrılmıştır. Birlik elemanlarının çakışık yerleştirildiğine dikkat ediniz. Elemanların bu biçimde çakışık yerleştirilmesi onların birinde değişiklik yapılması durumunda hepsinin bundan etkileneceği anlamına gelir. Birlikler iki amaçla kullanılmaktadır:

1) Farklı bilgilerden herhangi birinin ekonomik bir biçimde tutulması gerektiği durumlarda. Örneğin bir kişinin *T.C.* numarası ya da adresi yeterli ise bu bilginin çakışık olması yer kazancı sağlar. Tabi böylesi durumlarda hangi bilginin dolu olduğunun ayrıca bir bayrakla tutulması gerekecektir.

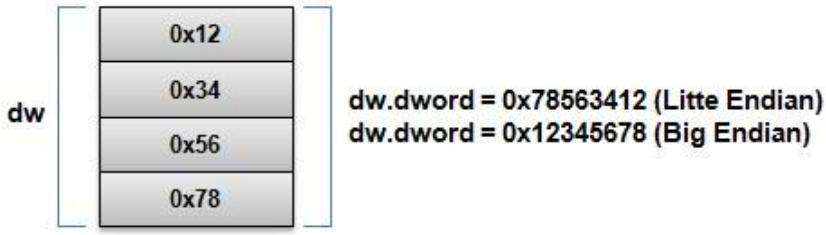
2) Bütünü parçalara ayırmak veya parçalardan bütünü oluşturmak için. Örneğin 4 *byte* işaretsiz bir tamsayı türünün *byte*'larını ayrı ayrı oluşturup onu bir bütün olarak almak ya da bütünü oluşturup herhangi bir *byte*'ını elde etmek için şöyle bir birlik oluşturulabilir:

```
union DWORD {
    unsigned int dword;
    struct {
        unsigned char b0, b1, b2, b3;
    } bytes;
};
```




```
dw.bytes.b0 = 0x12;
dw.bytes.b1 = 0x34;
dw.bytes.b2 = 0x56;
dw.bytes.b3 = 0x78;

printf("%x\n", dw.dword);
/* Litte Endian: 0x78563412, Big Endian:   0x12345678 */
```



Yukarıdaki örnekte 4 *byte*'lık bir nesnenin *byte*'larına sırasıyla bazı değerler yerleştirilmiş ve değerin bütünü elde edilmiştir. Bir *byte*'tan uzun olan nesnelerin içsel *byte* yerleşimlerinin işlemcinin *Endian*'lık durumuna göre değişebileceğini anımsayınız. *Intel* gibi *Little Endian* işlemciler sayıları düşük anlamlı *byte* değerleri düşük adreste olacak biçimde yerleştirirler.

Java'da *C/C++*'taki gibi birlik veri yapısı yoktur. Fakat yukarıdaki birinci maddede belirttiğimiz "farklı bilgilerden herhangi birinin ekonomik bir biçimde tutulması" amacını gerçekleştirmek için yapay bir birlik temsili oluşturulabilir. Bunun için bir taban sınıf alınır. Bu sınıftan iki ayrı sınıf türetilir. Bu iki ayrı sınıf iki ayrı bilgiyi temsil etmektedir. Taban sınıfın sanal metotlarıyla türemiş sınıfın veri elemanlarına erişim yapılabilir:

```
class U<A, B>
{
    public A getA() { return null; }
    public B getB() { return null; }
}

class UA<A, B> extends U<A, B>
{
    private A m_a;

    public UA(A a)
    {
        m_a = a;
    }

    public A getA()
    {
        return m_a;
    }
    //...
}

class UB<A, B> extends U<A, B>
{
    private B m_b;

    public UB(B b)
    {
        m_b = b;
    }

    public B getB()
    {
        return m_b;
    }
    //...
}
```

C#'ta birlik etkisi yaratmak için sınıf ya da yapı *StructLayoutAttribute* sınıfı ile, veri elemanları da *FieldOffsetAttribute* sınıfı ile özniteliklendirilir. *FieldOffsetAttribute* öznitelik sınıfı ilgili elemanın belli *offset*'te bulunmasını sağlamaktadır. Örneğin:

```
[StructLayout(LayoutKind.Explicit)]
public struct DWord
{
    [FieldOffset(0)] public byte b0;
    [FieldOffset(1)] public byte b1;
    [FieldOffset(2)] public byte b2;
    [FieldOffset(3)] public byte b3;
    [FieldOffset(0)] public uint dword;
}

//...

DWord dw = new DWord();

dw.b0 = 0x12;
dw.b1 = 0x34;
dw.b2 = 0x56;
dw.b3 = 0x78;

Console.WriteLine("{0:X4}", dw.dword);
// Little Endian: 78563412, Big Endian: 12345678
```

2.4. Sınıflar (Classes)

Sınıflar nesne yönelimli programlama (*object oriented programming*) tekniğinin yapı taşlarını oluşturan veri yapılarıdır. Nesne yönelimli programlama tekniğinin tek bir cümleyle tanımını yapmak olanaksız olsa da eksik bir biçimde "sınıflar kullanılarak program yazma tekniğidir" denebilir. Sınıflar nesne yönelimli programlama tekniğinde belli bir konudaki gerçek bir nesneyi ya da kavramı temsil etmek için kullanılırlar. Bir proje nesne yönelimli olarak modellenecekse önce projeye konu olan bütün gerçek nesneler ve kavramlar sınıflarla temsil edilir; sonra sınıflar arasındaki ilişkiler belirlenerek kodlama yapılır. Sınıflar hem veri elemanlarını hem de fonksiyonları tutan veri yapılarıdır. Sınıflar *C*'deki yapıların fonksiyon içeren biçimlerine benzetilebilir. Sınıfların veri elemanları (*data*'ları) sınıfların fonksiyonları tarafından ortak bir biçimde kullanılırlar. Sınıfların veri eleman organizasyonu tamamen yapılara benzemektedir. Sınıfların fonksiyonları sınıf nesnesi içerisinde yer kaplamazlar, fonksiyonlar mantıksal olarak sınıfla ilişkilendirilmişlerdir. Bir sınıf fonksiyonlara ya da veri elemanlarına sahip olmak zorunda değildir. Sınıfların veri elemanları o dillerin standartlarında açıkça belirtilmese de tıpkı yapılarda olduğu gibi nesne içerisinde ardışıl bir biçimde yerleştirilirler. Örneğin: [2]

```
class Sample {
public:
    void Foo();
    void Bar();
    void Tar();
    //...
private:
    int m_a;
    int m_b;
    int m_c;
};
```

```
Sample s;
```

---

[1]: Derleyicilerin dizi uzunluklarını derleme aşamasında belirlemeleri daha etkin kod üretimini mümkün hale getirmektedir. Tipik olarak derleyiciler yerel bir dizi için yer ayırırken yığın göstericisini (*stack pointer*) dizinin *byte* uzunluğu kadar geri çekerler. Bazı dillerde dizi uzunluklarının sabit ifadeleriyle belirlenmesi biçiminde bir zorunluluk yoktur. (Örneğin *C99*'da yerel dizi uzunlukları değişkenlerle belirtilebiliyor). Bu dillerde diğer yerel değişkenlerin yerlerinin belirlenmesi ek bir hesap ya da bellek alanı gerektirdiğinden etkinliğin azalacağı söylenebilir.

[2]: Nesne yönelimli programlama pek çok ayrıntısı olan bir programlama modelidir. Burada amacımız nesne yönelimli programlama modeli hakkında bilgi vermek değil, bir veri yapısı olarak sınıflar konusuna değinmektir. Dolayısıyla bu bağlamda bu kadar bilgiyi yeterli görüyoruz.