**Kullanıcının Oluşturduğu Türler(User Defined Types)**

C ile programlamada daha önceden gördüğümüz temel türler(primitive types) dışında da türler bulunmaktadır. Ancak temel türlerin yanı sıra bu türden kullanıcı tarafından oluşturulan türlerdir. Bu türlerin isimleri de alacağı eleman sayısı da tamamen kullanıcının inisiyatifine kalmıştır. Kullanıcının oluşturduğu türler C’de 3 tanedir. Bunlar :

1-) Structures (Yapılar) : struct anahtar kelimesini kullanarak oluşturulan bir tür olup C’nin nesne yönelimli programlamaya en yaklaştığı özelliktir. Kullanıcının oluşturduğu türler arasında en çok kullanılan tür budur.

2-) Unions(Birlikler) : union anahtar kelimesiyle tanımlanan bu user defined type struct ile çok benzerdir. Ancak ilerde ele alacağımız bazı farklılıkları elbette bulunmaktadır.

3-) Enumarations(Numaralandırma) : enum anahtar sözcüğü ile bildirilen bu user defined type yine oldukça sık kullanılmaktadır. enumlar çoğu programlama dilinde de bulunan bir sentakstır.

**Yapılar (Structures)**

Elemanları bellekte ardışıl biçimde tutulan fakat farklı türlerden olabilen veri yapılarına  "yapı (structure)" denilmektedir.Dizi ile yapılar birbilerine çok benzerler.Her ikisinde de elemanları ardışıl olarak bellekte tutulmaktadır. Fakat dizi elemanları aynı türden olduğu halde yapı elemanları farklı türlerinden olabilmektedir. Yapılarla çalışırken önce yapıların bir şablonu bildirilir. Buna "yapı bildirimi" denir. Yapı bildirimi bellekte yer kaplamaz. Yani tanımlama değildir. Yapı bildirimini gören derleyici yapı elemanlarının türlerini ve isimlerini öğrenir.  
Daha sonra bu yapı türünden gerçek nesneler tanımlanır. Yapı bildiriminin genel biçimi şöyledir:  
struct  
<isim>{  
<eleman-bildirimleri>  
};

Bu genel örneği de bir kod örneğine dönüştürmek gerekirse aşağıdaki gibi bir örnekler verilebilir:

struct DATE{

int day, month, year;

};

struct COMPLEX{

double real;

double img;

};

struct SAMPLE {  
    int a;  
    long b;  
    double c;  
};

Yapı isimlerini bazı programcılar büyük harflerle bazıları küçük harflerle harflendirmektedir. Biz kodlarımızı yazarken yapı isimlerini büyük harflerle harflendireceğiz. Yapı bildirimleri yerel ya da global düzeyle yapılabilir. Eğer yapı bildirimleri yerel düzeyde yapılmış ise o yapı ismi ancak o blokta kullanılabilir. Kısaca bu durum değişkenlerde olduğundan farklı değildir. Global olarak bildirilen bir ise yine normal bir değişken gibi her yerde kullanılabilir. Yapı bildirimleri genellikle global düzeyde yapılmaktadır. Çünkü çoğu kez yapıların farklı fonksiyonlarda kullanılması gerekmektedir. Bunun da rahat ve efektif yolu global bildirim yapılarak gerçekleştirilebilir. Her yapı bildirimi aynı zamanda bir tür de belirtmektedir. Yani bir yapı bildirimiyle oluşturulan türler struct anahtar sözcüğü ve yapı ismiyle ifade edilir. İlerde ele alacağımız structların typedef anahtar sözcüğüyle birlikte kullanımı ile bildirim yapılırken typedef anahtar sözcüğüne gerek olmayacaktır. Ancak şimdilik bir struct anahtar sözcüğünü kullanarak bu bildirimleri yapacağız. Örneğin yukarıdaki örnekteki DATE, COMPLEX ve SAMPLE structları için aşağıdaki gibi bir bildirimler yapılmalıdır :

struct DATE d;

struct COMPLEX c;

struct SAMPLE s;

Örneğin burada d’nin türü ne diye sorulduğunda cevabımız struct DATE türünden olmalıdır. Bu d aslında DATE yapısının day, month ve year parçalarından oluşan bir nesnenin bütününü ifade etmektedir. Yapı nesneleri genellikle bütünsel olarak kullanılmaz. Onların parçalarına erişilip parçaları kullanılır. Yapı elemanlarına erişmek için operatör öncelik tablomuzun 1. Öncelik seviyesinde bulunan nokta operatörü(dot operator) kullanılmaktadır. Nokta operatörü iki operandlı araek bir operatördür. Nokta operatörünün sol tarafındaki operand yapı nesnesinin bütünü, sağ taraftaki operand onun bir elemanı olmak zorundadır. Bu operatör sol taraftaki yapının sağ taraftaki elemanına erişmekte kullanılır. Örneğin :

#include <stdio.h>

struct SAMPLE {  
    int a;  
    long b;  
    double c;  
};  
  
int main(void) {  
    struct SAMPLE s;  
    s.a = 10;  
    s.b = 20;  
    s.c = 12.4;  
    printf("%d, %ld, %lf\n", s.a, s.b, s.c);  
    return 0;  
}

Burada s nesnesi struct SAMPLE türündendir. s.a ifadesi ise int türündendir. s.a ifadesi "s nesnesinin a parçası" anlamına gelmektedir. s.b ifadesi long türden, s.c ifadesi ise double türündendir. C standartlarına göre yapı bildiriminde ilk yazılan eleman düşük adreste bulunacak biçimde eleman ardışılığı oluşturulur. Yani örneğimizde s’nin a parçası en düşük adreste bulunmaktadır. Bunu b parçası izler onu da c parçası izler. Nokta operatörü öncelik tablosunun en yüksek öncelik seviyesinde olup, soldan sağa doğru yapılan operatörler grubundadır. Bu sebeple &s.a gibi bir ifadede s.a nesnesinin adresi alınmaktadır. Çünkü ampersand (address of operator) operatör öncelik tablosunun ikinci öncelik seviyesinde bulunmaktadır. Bu sebeple bu ifadede nokta operatörünün önce yapıldığını söyleyebiliriz. Aşağıdaki örnekte adreslerin ardışıl olduğu anlatılmak istenmiştir :

#include <stdio.h>  
  
struct SAMPLE {  
    int a;  
    long b;  
    double c;  
};  
  
int main(void) {  
    struct SAMPLE s;  
    printf("%p, %p, %p\n", &s.a, &s.b, &s.c);  
    return 0;  
}

Anahtar Notlar:Anımsanacağı gibi Kernighan & Ritchie yazım tarzında iki operandlı operatörlerle operandlar arasında birer boşluk karakteri bırakılıyordu. Fakat nokta ve -> operatörleri iki operandlı olmasına karşın bunlarla operandlar arasında boşluk karakteri bırakılmamaktadır. Bir yapı nesnesinin elemanlarına henüz değer atanmadıysa içerisinde ne vardır? İşte eğer yapı nesnesi yerelse onun tüm elemanlarında çöp değerler, global ise sıfır değerleri bulunur. Bir yapı nesnesine küme parantezleri içerisinde ilk değer verebiliriz. (Tıpkı dizilerde olduğu gibi) Örneğin:

#include <stdio.h>  
  
struct SAMPLE {  
    int a;  
    long b;  
    double c;  
};  
  
int main(void) {  
    struct SAMPLE s = {10, 20, 20.5};  
    printf("%d, %ld, %f\n", s.a, s.b, s.c);  
    return 0;  
}

Tıpkı dizilerde olduğu gibi yapının az sayıda elemanına ilkdeğer verebiliriz. Bu durumda geri kalan elemanlar derleyici tarafından sıfırlanır. Fakat yapının fazla sayıda elemanına ilkdeğer  
vermeye çalışırsak derleme aşamasında error oluşur. Aynı türden iki yapı nesnesi birbirlerine atanabilir. Bu durumda yapının karşılıklı elemanları birbirlerine atanmaktadır.

Örneğin :

#include <stdio.h>  
  
struct SAMPLE {  
    int a;  
    long b;  
    double c;  
};  
  
int main(void) {  
    struct SAMPLE s = {10, 20, 30.5};  
    struct SAMPLE k;  
    k = s;  
    printf("%d, %ld, %f\n", s.a, s.b, s.c);  
    printf("%d, %ld, %f\n", k.a, k.b, k.c);  
    return 0;  
}

Atama işleminde yapıların türlerinin aynı olması gerekir. Tür ise isme bağlıdır. İçi aynı olan farklı isimli yapılan birbirlerine atayamayız. Diziler de yapı elemanı olabilir. Bu durumda yapının dizi elemanlarına erişildiğinde bu ifade dizinin tamamını temsil eder, ifade içerisinde kullanıldığında ise yapı içerisindeki dizinin başlangıç adresini belirtir.

#include <stdio.h>  
  
struct PERSON {  
    char name[32];  
    int no;  
};  
  
int main(void) {  
    struct PERSON per;  
    printf("Adi soyadi:");  
    gets([per.name](http://per.name/));  
    printf("No:");  
    scanf("%d", &[per.no](http://per.no/));  
    printf("%s, %d\n", [per.name](http://per.name/), [per.no](http://per.no/));  
    return 0;  
}

Böyle yapı nesnesine ilk değer de verebiliriz :

struct PERSON person = {"Berke Sertel", 123};

Tabi buradaki iki tırnak ifadesi adres anlamına gelmez. Karakterlerin name dizisine kopyalanacağı anlamına gelir. Nokta operatörüyle index of operatörünün aynı grupta soldan sağa öncelikli olduğuna dikkat ediniz. Örneğin person.name[3] ifadesi person.name dizisinin 3.indeksli elemanı anlamına gelir. Yani :

person 🡪 struct PERSON türündendir.

person.name 🡪 char\* türündendir.

person.name[3] 🡪 char türündendir.

Örneğin:

#include <stdio.h>  
  
struct PERSON {  
    char name[32];  
    int no;  
};  
  
int main(void) {  
    struct PERSON per = {"Kaan Aslan", 123};  
    char ch;  
    ch = [per.name](http://per.name/)[3];  
    putchar(ch);  
    return 0;  
}

**Yapı Elemanı Olarak Göstericilerin Kullanılması**

Bir yapının elemanı bir gösterici olabilir. Örneğin :

struct PERSON {  
    char \*name;  
    int no;192  
};

struct PERSON per;

Burada name elemanı bir göstericidir ve bu göstericinin tahsis edilmiş bir alanı gösteriyor olması gerekir.

#include <stdio.h>  
  
struct PERSON {  
    char \*name;  
    int no;  
};

int main(void) {  
    struct PERSON per;  
    [per.name](http://per.name/) = "Kaan Aslan";  
    [per.no](http://per.no/) = 123;  
    printf("%s, %d\n", [per.name](http://per.name/), [per.no](http://per.no/));  
    return 0;  
}

**Yapı Türünden Adresler ve Göstericiler**

Bir yapı nesnesinin adresi alınabilir. Bu durumda elde edilen adresin sayısal bileşeni tüm yapı nesnesinin bellekteki başlangıç adresi, tür bileşeni ise yapı türünden pointerdır. Bir yapı nesnesinin adresi aynı türden bir yapı göstericisine atanabilir. Örneğin :

struct SAMPLE {  
    int a;  
    long b;  
    double c;  
};  
struct SAMPLE s;  
struct SAMPLE \*ps;  
ps = &s;

Bir yapı göstericisi ya da bir yapı türünden adres dereferencing operatörüyle kullanılırsa yapı nesnesinin tamamına erişilir. Yani yukarıdaki örnekteki \*ps demek ile s demek arasında bir fark yoktur. Burada ps struct SAMPLE\* türünden \*ps ise struct SAMPLE türündedir.

**Yapı Göstericisi Yoluyla Yapı Elemanlarına Erişilmesi**

P ile yapı türünden adres a da bu yapının bir elemanı olmak üzere bu adresin gösterdiği yerdeki yapı nesnesinin a elemanına erişmek için (\*p).a ifadesi kullanılır. \*p.a ifadesi ise geçersizdir. Çünkü nokta operatörünün içerik operatörüne göre daha yüksek bir öncelik seviyesindedir. Bu sebeple bu ifadede önce p.a yapılmaya çalışılır ki nokta operatörünün solundaki operand geçersiz olur. Çünkü nokta operatörünün solundaki operand yapı nesnesinin kendisi olmalıdır, adresi olmamalıdır. Örneğin :

#include <stdio.h>  
  
struct SAMPLE {  
    int a;  
    long b;  
    double c;  
};  
  
int main(void) {  
    struct SAMPLE s;  
    struct SAMPLE \*ps;  
    ps = &s;  
    (\*ps).a = 10;  
    (\*ps).b = 20;  
    (\*ps).c = 30;  
    printf("%d, %ld, %f\n", (\*ps).a, (\*ps).b, (\*ps).c);  
    return 0;  
}

**Ok (Arrow) Operatörü**

Ok operatörü -> karakterleriyle elde edilir. Ok operatörü iki operandlı ara ek bir operatördür. Ok operatörünün solundaki operand bir yapı türünden adres, sağındaki operand o yapının bir  
elemanı olmak zorundadır. On operatörü sol taraftaki operandla belirtilen adresteki yapı nesnesinin sağ taraftaki operandla belirtilen elemanına erişmekte kullanılır. Yani: p🡪a ile (\*p).a tamamen eşdeğerdir. Nokta operatörüyle ok operatörünün her ikisi de yapı elemanlarına erişmekte kullanılır. Nokta operatörü nesnenin kendisiyle ok operatörü adresiyle erişim sağlar. Ok operatörü öncelik tablosunun en üst düzeyinde soldan sağa grupta bulunur:

1.  ( ) [ ] . ->             Soldan-Sağa  
2.  + - ++ -- ! sizeof & \*   Sağdan-Sola  
3.  \* / %                     Soldan-Sağa  
4.  + -                       Soldan-Sağa  
        ... ...  
Örneğin:

#include <stdio.h>  
  
struct SAMPLE {  
    int a;  
    long b;  
    double c;  
};  
  
int main(void) {  
    struct SAMPLE s;  
    struct SAMPLE \*ps;  
    ps = &s;  
    ps->a = 10;  
    ps->b = 20;  
    ps->c = 30;  
    printf("%d, %ld, %f\n", ps->a, ps->b, ps->c);  
    return 0;  
}

Örneğin:

#include <stdio.h>  
  
struct POINT {  
    int x, y;  
};

int main(void)  
{  
    struct POINT pt;  
    struct POINT \*ppt;  
    ppt = &pt;  
    ppt->x = 10;  
    ppt->y = 20;  
    printf("(%d, %d)\n", ppt->x, ppt->y);  
    return 0;  
}

s bir yapı türünden nesne a da bu yapının bir elemanı olmak üzere &s🡪a ifadesi geçersizdir. Çünkü burda da operatör önceliğinden dolayı ok operatörü önce yapılmaya çalışılır. Bu sebeple sentaks hatası oluşur. Ancak (&s)🡪 a ifadesi geçerlidir ve s.a demek ile eşdeğerdir. p bir yapı türünden adres ve a da bu yapının bir elemanı olmak üzere &p🡪a ifadesi geçerlidir ve bu ifade p göstericisinin gösterdiği yerdeki nesnenin a parçasının adresi anlamına gelir.

#include <stdio.h>  
  
struct PERSON {  
    char name[32];  
    int no;  
};  
  
int main(void) {  
    struct PERSON per = {"David Guetta", 234};  
    struct PERSON \*pper;  
    pper = &per;  
    printf("%s, %d\n", pper->name, pper->no);  
    printf("%c\n", pper->name[2]);  
    return 0;  
}

**Yapıların Fonksiyonlara Parametre Yoluyla Aktarılması**

Yapıların fonksiyonlara aktarılmasında iki farklı teknik kullanılabilir. Bunlardan birincisi nesnenin kopyalama yoluyla aktarılması tekniği, diğer ise adres yoluyla aktarma tekniğidir. Nesnenin kendisinin aktarılması genel olarak kötü bir tekniktir. Bunun sebebi kopyalama semantiğinin bellek açısından efektif olmamasından kaynaklanmaktadır. Kopyalama semantiği yerine yapıların adresleri kullanılarak call by reference yapmak çok daha iyi bir tekniktir. C programlama dilinde yapı nesneleri üretim tarafından çoğunlukla hatta hemen hemen her zaman adres yoluyla aktarım yapılır.

**Yapı Nesnelerinin Fonksiyonlara Kopyalama Yoluyla Aktarılması**

Bu yöntemde fonksiyonun parametre değişkeni bir yapı türünden yapı nesnesidir. Fonksiyon da aynı yapı türünden bir nesneyle çağrılır. Aynı türden iki yapı nesnesi atanabildiğine göre bu çağırma geçerlidir. Ancak burada aktarım kopyalama yoluyla yapılmaktadır. Örneğin:

#include <stdio.h>  
  
struct PERSON {  
    char name[32];  
    int no;  
};  
  
void foo(struct PERSON per) {  
    printf("%s, %d\n", [per.name](http://per.name/), [per.no](http://per.no/));  
}  
  
int main(void) {  
    struct PERSON x = {"Ali Serce", 234};  
    foo(x);  
    return 0;  
}

Bu yöntem daha önce de söylediğimiz gibi genel olarak kötü bir tekniktir. Çünkü büyük bir yapının bu yöntemde tüm elemanlarını tek tek aktarım sırasında fonksiyona kopyalanır. Üstelik bu yöntemde fonksiyon içerisinde artık biz orijinal nesneye erişemeyiz. Tabi eğer yapı çok küçükse bu teknik kötü bir teknik olmaz. Bu durumda kullanılabilir.

Yapı Nesnelerinin Fonksiyonlara Adres Yoluyla Aktarılması

Bu yöntemde fonksiyonun parametre değişkeni yapı türünden gösterici olur. Fonksiyon da aynı türden bir yapı nesnesinin adresiyle çağrılır. Bu kullanılması gereken doğru tekniktir.

Örneğin:

#include <stdio.h>  
  
struct PERSON {  
    char name[32];  
    int no;  
};  
  
void foo(struct PERSON \*per) {  
    printf("%s, %d\n", per->name, per->no);  
}  
  
int main(void) {  
    struct PERSON x = {"Ali Serce", 234};  
    foo(&x);  
    return 0;  
}

Örneğin:

#include <stdio.h>  
  
struct DATE {  
    int day;  
    int month;  
    int year;  
};  
  
void get\_date(struct DATE \*date)

{  
    printf("Gun:");  
    scanf("%d", &date->day);  
    printf("Ay:");  
    scanf("%d", &date->month);  
    printf("Yil:");  
    scanf("%d", &date->year);  
}

Örneğin:

void disp\_date(struct DATE \*date)

{  
    printf("%02d/%02d/%04d\n", date->day, date->month, date->year);  
}  
  
int main(void) {  
    struct DATE date;  
    get\_date(&date);  
    disp\_date(&date);  
    return 0;  
}

Örneğin:

#include <stdio.h>  
  
struct COMPLEX {  
    double real, imag;  
};  
  
void get\_comp(struct COMPLEX \*comp) {  
    printf("Gercek Kisim:");  
    scanf("%lf", &comp->real);  
    printf("Sanal Kisim:");  
    scanf("%lf", &comp->imag);  
}  
  
void disp\_comp(struct COMPLEX \*comp) {  
    printf("%.0f+%.0fi\n", comp->real, comp->imag);  
}  
  
int main(void) {  
    struct COMPLEX z;  
    get\_comp(&z);  
    disp\_comp(&z);  
    return 0;  
}

**Yapılara Neden Gereksinim Duyulmaktadır?**

1-) Yapılar olgular için mantıksal bir kapı oluşturmaktadır. Yani tarih gibi, sanal sayılar gibi, şahıs bilgileri gibi birbirleriyle ilgili çok sayıda nesne bir yapıyla ifade edilirse daha kolay bir temsil yeteneği elde edilir. Gerçekten de mantıksal bakımdan birbiriyle bağlantılı nesnelerin yapılarla temsil edilmesi okunabilirliği ve anlaşılabilirliği artırmaktadır.

2-) Bir fonksiyona çok sayıda parametre aktarılacaksa onların tek aktarılmaları hem yazılımsal olarak zordur. Hem anlaşılır olmaktan çıkar hem de yavaştır. Bunun yerine çok sayıda parametre bir yapıda toplanıp tek bir parametre biçiminde fonksiyona geçilebilir.

3-) Bir fonksiyonun tek bir geri dönüş değeri vardır. Eğer fonksiyon dış dünyaya çok sayıda  
değer iletecekse bu yapılarla sağlanabilir. Örneğin iletilecek değerler bir yapıyla ifade edilir.  
Fonksiyona yapı nesnesinin adresi geçirilir. Fonksiyon da bu nesnenin içini doldurur.

**Fonksiyonların Geri Dönüş Değerlerinin Yapı Olması Durumu**

Bir fonksiyonun geri dönüş değeri bir yapı türünden olabilir. Bu durumda return ifadesi de aynı türden bir yapı nesnesi olmalıdır. Aksi durumda tanımsız davranış oluşur. Aslında bu yöntem de C dilinde çoğu kez (yapının byte uzunluğu büyükse) iyi bir teknik olarak kabul edilmez. Çünkü burada return işlemi sırasında geçici nesneye bir kopyalama yapılmakta ve geri dönüş değerinin atanması sırasında da aynı sorun oluşmaktadır. Örneğin:

#include <stdio.h>  
  
struct COMPLEX {  
    double real, imag;  
};  
  
void disp\_comp(struct COMPLEX \*comp) {  
    printf("%.0f+%.0fi\n", comp->real, comp->imag);  
}  
  
struct COMPLEX add\_comp(struct COMPLEX \*z1, struct COMPLEX \*z2) {  
    struct COMPLEX result;  
    result.real = z1->real + z2->real;  
    result.imag = z1->imag + z2->imag;  
    return result;  
}  
  
int main(void) {  
    struct COMPLEX z1 = {3, 2};  
    struct COMPLEX z2 = {1, 4};  
    struct COMPLEX result;  
    result = add\_comp(&z1, &z2);  
    disp\_comp(&result);  
    return 0;  
}

Genellikle programcılar çoklu biligiyi böyle almaktansa bir nesne verip fonksiyonun onun içerisine yerleştirme yapmasını tercih ederler. Örneğin:

#include <stdio.h>  
  
struct COMPLEX {  
    double real, imag;  
};  
  
void disp\_comp(struct COMPLEX \*comp) {  
    printf("%.0f+%.0fi\n", comp->real, comp->imag);  
}  
  
void add\_comp(struct COMPLEX \*z1, struct COMPLEX \*z2, struct COMPLEX \*result) {  
    result->real = z1->real + z2->real;  
    result->imag = z1->imag + z2->imag;  
}  
  
int main(void) {  
    struct COMPLEX z1 = {3, 2};  
    struct COMPLEX z2 = {1, 4};  
    struct COMPLEX result;  
    add\_comp(&z1, &z2, &result);  
    disp\_comp(&result);  
    return 0;  
}

Yapılarda Hizalama(Alignment Kavramı)

Modern 32 işlemcilerde bellek bağlantısı bir hamlede 4 byte bilgiyi çekecek biçimde yapılmıştır. Benzer biçimde 64 bit işlemcilerde de bellek bağlantısı bir hamlede 8 byte bilgiyi çekecek biçimde yapılmıştır. Böylece örneğin 32 bit işlemciler aslında 32 adres yoluna değil 30 adres yoluna sahiptir. Tabi makine komutları yine aynı biçimde byte adreslemeli çalışmaktadır.

Aşağıda 32 bit bir işlemcinin bellek erişimi resmedilmiştir:

|  |
| --- |
| 4 byte |
| 4 byte |
| 4 byte |
| 4 byte |

Bu sistemlerde eğer 4 byte'lık bir bilgi (örneğin int türden bir bilgi) 4' ün katlarında değilse makina komutları göreli olarak daha yavaş çalışmaktadır. Çünkü bu 4 byte'ı işlemci iki bus erişimiyle elde eder:

Tabi bir byte bir bilgiye kaçın katlarında olursa olsun tek bus hareketiyle erişilebilmektedir. Peki 2 byte'lık bilgiler bu durumda nasıl saklanacak? Bunların da 2'nin katlarında olması gerekir. Örneğin 2'nin katlarında olmayan 2 byte'lık (örneğin short türden bir nesne) bir nesne aşağıdaki gibi gösterilebilir: İşte derleyiciler işlemcilerin böyle çalıştıklarını bildiği için makine komutları daha hızlı çalışsın diye 4 byte'lık 200 nesneleri 4'ün katlarına, 8 byte'lık nesneleri 8' in katlarına, 2 byte'lık nesneleri 2' nin katlarına, 1 byte'lık nesneleri 1'in katlarına yerleştirmektedir. Derleyiciler bunlara yerel değişkenler için ve global değişkenler için dikkat ederler. Yapı elemanları bellekte ardışıl olacağından ve ilk yazılan elemanın düşük adreste olması gerekeceğinden bu hizalama (alignement) yapılar için nasıl gerçekleştirilecektir?  
İşte derleyiciler yapı elemanlarının arasına boşluklar koyarak o elemanların belli adres katlarında olmasını sağlayabilmektedir. Örneğin:

struct SAMPLE {  
    char a;  
    int b;  
    char c;  
    int d;  
};  
struct SAMPLE s;

32 bit bir işlemcide bu yapı nesnesinin bellekte 10 byte yer kaplayacağı düşünülebilir.Ancak derleyiciler a ile b arasına ve c ile d arasına 3'er byte boşluk bırakarak int olan kısımların 4'ün  
katlarına gelmesini sağlayabilmektedir. Böylece bu yapı nesnesinin sizeof değerinin 16 çıkması programcıyı şaşırtmamalıdır. Örneğin aşağıdaki örnekte SAMPLE yapısının sizeof değeri son durumda 16 byte olarak elde edilecektir.

#include <stdio.h>  
  
struct SAMPLE {  
    char a;  
    int b;  
    char c;  
    int d;  
};

int main(void)

{  
    struct SAMPLE s;  
    printf("%u\n", sizeof(s)); /\* 16 \*/  
    return 0;  
}

Peki yukarıdaki örnekteki yapıyı aşağıdaki gibi düzenlersen sizeof değeri nasıl değişirdi?

#include <stdio.h>  
  
struct SAMPLE {  
    char a;  
    char b;  
    int c;  
    int d;  
};  
  
int main(void) {  
    struct SAMPLE s;  
    printf("%u\n", sizeof(s)); /\* 12 \*/  
    return 0;  
}

Elbette bu durumda padding byte sayısı 2 olacaktır. Çünkü char türden değişkenler birlikte 4 bytelık bellek bloğunun 2 bytelık alanını dolduracak şekilde hizalanacaktır. Bu yüzden boş kalan bayt (padding byte) sayısı 2 olacaktır. Buna göre son durumda bu yapı için ayrılan bellek bloğu totalde 12 byte olacaktır.

Hizalama pek çok derleyicide derleyici seçeneklerinden yönetilebilmektedir. Örneğin Microsoft C derleyicilerinde hizalama Proje ayarlarında C-C++/ Code Generation /Struct Member Alignment ile değiştirilebilmektedir. Eğer hizalama 1 byte hizalama moduna çekilirse derleyici yapı elemanlarını 1'in katlarına yerleştirmeye çalışır. Dolayısıyla elemanlar arasında hiç boşluk bırakmaz. C standartlarına hizalama kavramı bir kurala bağlanmamıştır. Standartlarda yalnızca derleyicinin yapı elemanları arasında boşluk bırakabileceği belirtilmiştir. Peki derleyicinin yapı elemanları arasında boşluk bırakabilmesi erişimde bir soruna yol açar mı? Yanıt hayır. Çünkü derleyici nerelere boşluk bıraktığını bildiği için ona göre erişimi yapmaktadır. Örneğin aşağıdaki yapı için derleyici 4 byte hizalama kullanmış olsun:

struct SAMPLE {  
    char a;  
    int b;  
    char c;  
    int d;  
};

Şimdi p göstericisinin bu yapıyı gösterdiğini düşünelim. Artık derleyici p->b ifadesiyle p adresinden 1 byte sonraya değil 4 byte sonraya erişir. Çünkü araya boşluk bıraktığını zaten kendisi bilmektedir. Yani yukarıdaki yapıyı biz şöyle düşünebiliriz:

struct SAMPLE {  
    char a;  
    char temp1, temp2, temp3;  
    int b;  
    char c;  
    char temp4, temp5, temp6;  
    int d;  
};

**Yapılarda Dinamik Bellek Yönetimi**

Yapılar bellekte ardışıl bir şekilde tutulduğuna göre yapı elemanları için heap bellek kullanılarak dinamik ömürlü yapı nesneleri de yaratılabilir. Dinamik tahsisat yaparken hizalama olasılığı göz önüne almak gerekir. Bu nedenle yapının byte uzunluğunun sizeof operatörüyle elde edilmesi oldukça önemlidir. (Aynı dinamik ömürlü dizilerde olduğu gibi) sizeof operatörü derelyicinin o anada uyguladığı hizalamayı da hesaba katmaktadır. Aşağıdaki örnekte bir tane dinamik ömürlü yapı nesnesi oluşturulurmuştur:

#include <stdio.h>  
#include <stdlib.h>  
  
struct PERSON {  
    char name[32];  
    int no;  
};  
  
int main(void) {  
    struct PERSON \*per;  
    per = (struct PERSON \*) malloc(sizeof(struct PERSON));  
    if (per == NULL) {  
        printf("cannot allocate memory!..\n");  
        exit(EXIT\_FAILURE);  
    }  
    printf("Adi soyadi:");  
    gets(per->name);  
    printf("No:");  
    scanf("%d", &per->no);  
    printf("%s, %d\n", per->name, per->no);  
    free(per);  
    return 0;  
}

**Yapı Dizileri**

Her elemanı bir yapı nesnesi olan dizilere yapı dizileri (array of structures) denilmektedir. Yapı diziler de normal dizilerde olduğu gibi bildirilir. Örneğin:

#include <stdio.h>  
#include <string.h>  
  
struct PERSON {  
    char name[32];  
    int no;  
};  
  
int main(void) {  
    struct PERSON persons[3];  
    int i;  
    strcpy(persons[0].name, "Felipe Melo");  
    persons[0].no = 123;  
    strcpy(persons[1].name, "Fernando Muslera");  
    persons[1].no = 234;  
    strcpy(persons[2].name, "Didier Drogba");  
    persons[2].no = 678;  
    for (i = 0; i < 3; ++i)  
        printf("%s, %d\n", persons[i].name, persons[i].no);  
    return 0;  
}

Anımsanacağı gibi dizi isimleri tüm diziyi temsil etmektedir. Ancak bunlar işleme sokulduğunda derleyici tarafından dizinin başlangıç adresine dönüştürülmektedir. Yani biz dizi isimleri kullanıldığında nesne belirtmezler. Biz bir yapı dizisinin ismini aynı yapı türünden bir yapı göstericisine atayabiliriz. Aynı aşağıdaki örnekte gösterildiği gibi:

#include <stdio.h>  
#include <string.h>  
  
struct PERSON {  
    char name[32];  
    int no;  
};  
  
int main(void) {  
    struct PERSON persons[3], \*per;  
    int i;  
    strcpy(persons[0].name, "Felipe Melo");  
    persons[0].no = 123;  
    strcpy(persons[1].name, "Fernando Muslera");  
    persons[1].no = 234;  
    strcpy(persons[2].name, "Didier Drogba");  
    persons[2].no = 678;  
    per = persons;  
    for (i = 0; i < 3; ++i) {  
        printf("%s, %d\n", per->name, per->no);  
        ++per;  
    }  
    return 0;  
}

Bir yapı göstericini bir artırdığımızda göstericinin içerisindeki adres yapı uzunluğu kadar artmaktadır. Yapı dizilerine de küme parantezleri ile ilkdeğer verilebilir. Bu durumda eleman olan her yapı nesnesi için de ayrı bir küme parantezi kullanılır. Örneğin:

struct PERSON persons[3] = {  
        {"Felipe Melo",   123},  
        {"Fernando Muslera",    345},  
        {"Didier Drogba", 654}  
};

Aslında bu tür durumlarda, İçteki küme parantezleri zorunlu değildir. Örneğin yukarıdaki ilk değer verme şöyle yapılabilirdi:  
struct PERSON persons[3] = {  
        "Felipe Melo", 123, "Fernando Muslera", 345, "Didier Drogba", 654  
};  
Fakat bu biçimde ilkdeğer verme hem okunabilirliği azaltmakta hem de aradaki bir  
değer unutulduğunda diğer tüm değerlerin yanlış elemanlara atanması gibi bir soruna yol  
açabilmektedir. Örneğin:

struct POINT {  
    int x;  
    int y;  
};  
struct POINT points[5] = {  
        {1, 2},  
        {4, 7},  
        {6, 8},  
        {7, 9},  
        {3, 4}  
};

Örneğin yukarıdaki örnekte ilk noktada y koordinatı yazılmadığı takdirde y koordinatı 0 olarak alınacaktı. Örneğin aşağıdaki kodda bu durum yaşanmaktadır:

struct POINT points[5] = {  
        {1},  
        {4, 7},  
        {6, 8},  
        {7, 9},  
        {3, 4}  
};

Aşağıdaki kodda gösterildiği gibi yapı elemanları yerel değişkenler gibi de kullanılabilir. Bu üretimde çok tavsiye edilen bir yöntem değildir.

#include <stdio.h>  
  
struct POINT {  
    int x;  
    int y;  
};  
  
int main(void) {  
    struct POINT points[5] = {  
            {1, 2},  
            {4, 7},  
            {6, 8},  
            {7, 9},  
            {3, 4}  
    };  
    int i;  
    for (i = 0; i < sizeof(points) / sizeof(\*points); ++i)  
        printf("(%d, %d)\n", points[i].x, points[i].y);  
    return 0;  
}

Bir yapı dizisini de fonksiyona argüman olarak göndermek mümkündür. Bunun için yine onun başlangıç adresini ve uzunluğu fonksiyona geçiririz. Örneğin aşağıdaki örnekte bu yapılmıştır.

#include <stdio.h>  
#include <string.h>  
  
struct PERSON {  
    char name[32];  
    int no;  
};  
void sort\_persons\_byname(struct PERSON \*per, int size);  
  
void sort\_persons\_byno(struct PERSON \*per, int size);  
  
void disp\_persons(struct PERSON \*per, int size);  
  
int main(void) {  
    struct PERSON persons[] = {  
            {"Selami Hakyemez",      123},  
            {"Ahmet Hamdi Tanpinar", 523},  
            {"Hulisi Sen",           323},  
            {"Siracettin Bilyap",    654},  
            {"Ali Ipek",             234}  
    };  
    sort\_persons\_byname(persons, 5);  
    disp\_persons(persons, 5);  
    printf("-----------\n");  
    sort\_persons\_byno(persons, 5);  
    disp\_persons(persons, 5);  
    return 0;  
}  
  
void sort\_persons\_byname(struct PERSON \*per, int size) {  
    int i, k;  
    struct PERSON temp;  
    for (i = 0; i < size - 1; ++i)  
        for (k = 0; k < size - 1 - i; ++k)  
            if (strcmp(per[k].name, per[k + 1].name) > 0) {  
                temp = per[k];  
                per[k] = per[k + 1];  
                per[k + 1] = temp;  
            }  
}  
  
void sort\_persons\_byno(struct PERSON \*per, int size) {  
    int i, k;  
    struct PERSON temp;  
    for (i = 0; i < size - 1; ++i)  
        for (k = 0; k < size - 1 - i; ++k)  
            if (per[k].no > per[k + 1].no) {  
                temp = per[k];  
                per[k] = per[k + 1];  
                per[k + 1] = temp;  
            }  
}  
  
void disp\_persons(struct PERSON \*per, int size) {  
    int i;  
    for (i = 0; i < size; ++i)  
        printf("%s, %d\n", per[i].name, per[i].no);  
}

**Yapı Bildirimleriyle Nesne Tanımlamasın Birlikte Yapılması**

Bir yapı bildirilirken bildirim noktalı virgül ile kapatılmayıp bir değişken listesi de yazılırsa aynı zamanda o yapı türünden nesneler de tanımlanmış olur. Örneğin:

struct POINT {  
    int x, y;  
} pt1, pt2;

Bu bildirimin aşağıdaki bildirimden bir farkı yoktur.

struct POINT {  
    int x, y;  
};  
struct POINT pt1, pt2;

Örneğin:

struct PERSON {  
    char name[32];  
    int no;  
} per = {"Berke Sertel", 123}, \*pper, persons[] = {{"Kaya Sertel", 123}, {"Erişcan Melih", 345} };

Burada “per” struct PERSON türünden bir nesnedir. “pper” struct PERSON türünden bir göstericidir. Persons ise struct PERSON [] türünden bir dizisidir. Bu biçimde bildirilmiş değişkenler yapı global olarak bildirildiyse global, yerel olarak bildirildiyse yerel biçimdedir. C standartlarına göre eğer yapı bildirimi ile aynı zamanda o yapı türünden nesne tanımlanıyorsa bu durumda yapıya isim verilmeyebilir. Örneğin:

struct {  
      char name[32];  
      int no;  
        } x, y;

ifadesi geçerlidir. Ancak eğer yapı türünden değişken tanımlanıyorsa yapıya isim vermek zorunludur. Örneğin aşağıdaki bildirimde bir sentaks hatası oluşur:

struct { //geçersiz  
      char name[32];  
      int no;  
        };

İsimsiz yapıların hepsi farklı bir tür kabul edilir. Yani örneğin:  
struct {  
      char name[32];  
      int no;  
        } x;  
struct {  
      char name[32];  
      int no;  
        } y;

Bu türler derleyici için aynı tür olarak kabul edilmemektedir. Bu yüzden bir örneğin bu türleri birbirine atayamayız.

**İç İçe Yapı Bildirimleri**

Bir yapının elemanları başka bir yapı türünden olabilir. Bu tür durumlara iç içe yapı bildirimleri denilmektedir. Örneğin:

struct DATE {  
            int day, month, year;  
        };  
  
struct PERSON {  
            char name[32];  
            struct DATE bdate;  
            int no;  
        };

Veya:

#include <stdio.h>  
  
struct DATE {  
        int day, month, year;  
        };

struct PERSON {  
       char name[32];  
       struct DATE bdate;  
       int no;  
       };

int main(void)  
{  
    struct PERSON per = {"Batuhan Menküer", {12, 10, 1970}, 2000};  
    printf("%s, %d/%d/%d, %d\n", [per.name](http://per.name/), per.bdate.day, per.bdate.month, per.bdate.year, [per.no](http://per.no/));  
            return 0;  
  
}

İç içe yapılarda ilkdeğer verilirken iç yapı ayrıca küme parantezine alınabilir ya da alınmayabilir. Ancak okunabilirlik bakımından iç yapıya ilişkin değerlerin küme parantezleri içerisine alınması tavsiye edilir. İç içe yapıların alternatif bildirim biçimi de vardır. Bu biçimde iç yapı dış yapının fiziksel olarak içerisinde bildirilir. Örneğin:  
struct PERSON {  
     char name[32];  
     struct DATE {  
         int day, month, year;  
          } bdate;  
         int no;  
        };  
Burada dış yapı bildirimi içerisinde hem iç yapı bildirilmiş hem de o yapı türünden değişken bildirimi yapılmıştır. Bu biçimdeki bildirimde içeride bildirilen yapı (örneğimizde struct DATE) ayrıca dış yapıdan bağımsız olarak da kullanılabilir. Örneğin:  
   struct PERSON per;  
   struct DATE date;  //geçerli  
Yani bu iki iç içe yapı bildirim biçimi semantik olarak tamamen eşdeğerdir. Genel olarak birinci biçimin daha okunabilir olduğunu söylenebiliriz.

**Typedef bildirimi ile yapıların bildirilmesi**

Bir yapı nesnesi bildirilir normalde her zaman struct anahtar sözcüğü kullanılmalıdır. Ancak hem okunabilirlik algılanabilirlik açısından her zaman struct anahtar sözcüğünün kullanmak üretimde neredeyse hiç kullanılmamaktadır. Bunun yerine struct anahtar sözcüğü her zaman kullanmanın engellenmesi için yapılar typedef bildirimleriyle birlikte kullanılmaktadır. Üretimde typedef bildirimsiz bir struct tanımlaması görmek neredeyse imkansızdır. C++ dilinde typedef bildirimi yapmakla yapmamak arasında bir fark yoktur. Orda yapılar struct sözcüğü olmadan da bildirilebilirler. Ancak C’de eğer bildirim struct anahtar sözcüğü olmadan yapılmak isteniyorsa mutlaka typedef bildirimiyle birlikte kullanılmalıdır. Örneğin Aşağıdaki örnekte bir sentaks hatası oluşur:

struct Person

{

char name[24];

char surname[24];

int ID;

};

İnt main(void)

{

Person person; //error

}

Ancak yukarıdaki sentaks hatası aşağıdaki gibi ortadan kaldırılabilir:

typedef struct

{

char name[24];

char surname[24];

int ID;

}Person;

int main(void)

{

Person person; //legal

}

Bu durumda sentaks tamamen legaldir. Ancak bir önceki kodun da C++ kurallarına göre geçerli olduğuna da unutmamak gerek. Genel olarak typedef bildirimlerinin üretimde en sık kullanıldığı yerlerden bir tanesi yapılardır.

**Birlikler (Unions)**

Birlikler yapılara oldukça benzemektedir. Hatta sentaks olarak neredeyse hiçbir fark yoktur. Sadece normalde struct anahtar sözcüğü yerine union anahtar sözcüğü kullanılmaktadır. Genel sentaks aşağıdaki gibidir.

<union anahtar sözcüğü> <değişken ismi>{

<birlik elemanları>

};

Anahtar Notlar : typedef anahtar sözcüğü ile yapıların birlikte kullanıldığını ve bu sentaksın üretimde sıkça kullanıldığını söylemiştik. Ancak popüler kullanım alanlarından biri de birliklerdir. İlerde ele alacağımız enumlar konusunda da typedef bildirimlerini sıkça kullanacağız. Yapılarla birliklerin typedef bildirimi ile kullanımı oldukça benzerdir. Örneğin aşağıdaki örnekte bir birlik typedef bildirimiyle birlikte kullanılmıştır. Bu durumda yapılara benzer olarak bu struct değil union anahtar sözcüğüne gereksinim duyulmaz.

**Yapılar ile Unionların farkı**

Yapılar genellikle (hizalama durumundaki değişiklikler dışında) içinde bulunan tüm elemanların sizeof u kadar yer kaplarken unionlarda bu durum farklıdır Unionlar her zaman içinde bulunan birlik elemanlarından en büyük sizeof değerine sahip olan değişkenin sizeof u kadarlık sizeof değerine sahiptir.Bu da demektir ki birlik elemanlarından birinin değerinin değiştirilmesi demek hepsinin değişmesi anlamına gelmektedir. Bu görünürde dezavantaj gibi görülse de bellekten tasarruf sağlamak açısından oldukça işe yaramaktadır.

Bu fark açıklamasına göre aşağıdaki örnekteki kod çalıştırıldığında ekran çıktısı tahmin etmeye çalışınız:

#include <stdio.h>  
  
typedef struct{  
  
 int a;  
 char ch;  
 char name[20];  
  
}SData;

typedef union{  
 int a;  
 char ch;  
 char name[20];  
}UData;  
  
int main(){  
  
 printf("Size of SData:%d\n",sizeof(SData));//40  
 printf("Size of UData:%d\n",sizeof(UData));//20  
  
 return 0;  
}

Birliklere ilk değer verilirken 1 ve yalnız bir initializer kullanılabilir. Birden fazla değğişken kullanımı anlamsız olduğu için warning’e sebep olur. Bu bir sentaks hatası olmasa da bu şekilde bir tanımlama asla tavsiye edilmez.

typedef union{  
 int a;  
 char ch;  
 char name[20];  
}UData;  
  
int main(){  
  
 UData mydata={345};  
 UData mydata2={.a=12496,.name="Berke"};//warning  
  
 return 0;  
}

**Unionlar Ne İşe Yarıyor?**

Öyle durumlar var ki biz bir veriyi farklı iki şekilde temsil etmek istiyoruz. Aşağıdaki örnekte olduğu gibi:

#include <stdio.h>  
#include <stdint.h>  
  
typedef union{  
 uint32\_t uval;  
 struct{  
 uint16\_t min;  
 uint16\_t sec;  
 };  
  
}UDur;

int main(void){  
  
 UDur mydur={72458271};  
  
 printf("%u dakika\n",mydur.min);  
 printf("%u saniye\n",mydur.sec);  
}

Bu örnekte yüksek anlamlı byte değerleri dakika parametresinde kalan byte değerleri ise saniye parametresinde tutulacaktır.

Ancak tek kullanım alanı bu olmayabilir bazen iki bilgiden duruma göre yalnızca birine ihtiyaç var ise bu gibi durumlarda bellek tasarrufu için de kullanılabilir. Mesela aşağıdaki örnekte kişinin kadın veya erkek olmasına bağlı olarak askerlik bilgisine veya kızlık soyadı bilgisi duruma göre ihtiyaç duyulabilir. Kadınlar için anne kızlık soyadı bilgisi var iken askerlik bilgisi gereksizdir aynı şekilde yalnızca askerlik bilgisine ihtiyaç duyulan erkek kişi için kızlık soyadı bilgisi gereksizdir o sebepten bu gibi durumlar için union kullanımı bellekten tasarruf için oldukça kullanışlıdır.

typedef struct{  
  
 char name[24];  
 int no;  
 double wage;  
 union{  
 char maiden name[24];  
 struct{  
 int status;  
 char palace[20];  
 int id;  
 };  
 };  
};

Anahtar Notlar: Unionlarda tahmin edilebileceği gibi eğer değişkenlerden biri değiştirilirse bu durumda tüm değişkenler değişmiş olur. Örneğin aşağıdaki örnekte b değiştirildiğinde a da değişecektir.

#include <stdio.h>  
  
typedef union{  
  
 int a;  
 int b;  
  
}Data;  
  
int main(){  
  
 Data d;  
  
 d.a=10;  
 d.b=20;  
  
 printf("%d",d.a);//20  
  
}

Enumarations(Numaralandırma)

3. ve son user defined type kategorisindeki elemanımız enumlar üretim tarafında sıkça kullanılmaktadır. Türkçesi numaralandırmadır ve adından da anlaşılacağı gibi elemanları sırası ile numaralandırılır. syntax olarak birlikler ve yapılara çok benzemektedir. Genel syntax özelliği şöyledir:

<enum anahtar sözcüğü> <isim>{  
 <elemanlar>  
};  
typedefli:  
typedef <enum anahtar sözcüğü> {  
 <elemanlar>  
}<isim>;  
  
genel yapısından da anlaşılabileceği gibi yapılar ve birliklerle syntax olarak aynıdır denilebilir. Ancak enum elemanları değişken olarak değil sabit olarak düşünülmedir. #define ön işlemci komutuyla bir macronun define edilmesine benzetilebilir.

enum içerisindeki elemanlara değer verilebilir ve ona göre numaralandırma  
devam eder ancak hiçbir değer verilmezse normal dizi indisleri gibi değer verilir. Örneğin 5 elemanlı bir enum için birinci eleman 0'dır sonraki elemanlar bir bir artarak devam etmektedir bu durumda son elemanın değer 4 olur. enuma ilk değer verilmesi durumunda diğer elemanların değerleri ona bağlı olarak numaralandırılır.  
  
#include <stdio.h>  
  
typedef enum{  
 BLUE,  
 RED=0,  
 PURPLE,  
 WHITE=3,  
 GREY,  
}COLORS;  
  
int main(){  
  
 printf("Blue:%d\n",BLUE);//0  
 printf("Purple:%d\n",PURPLE);//1  
 printf("Grey:%d\n",GREY);//4  
  
}  
  
C'de aynı isimde farklı iki enum elemanı yazımı geçersizdir.  
Not: C++ dilinde bu errore sebep olmaz!! Bu kısmı ilerde ele alacağız.

typedef enum{  
 RED,//error  
 GREEN,  
 YELLOW,  
}TRAFFIC\_LIGHTS;  
  
  
typedef enum{  
 BLUE,  
 RED,//error  
 PURPLE,  
 WHITE,  
 GREY,  
}COLORS;  
  
Aşağıdaki örnekte enumın uzunluğu 3 gibi görünse de aslında sizeof değeri 4 olacaktır.

#include <stdio.h>  
  
typedef enum {  
 C, CPP, Java  
}Xenum;  
  
int main()  
{  
 Xenum var;  
 printf("%d",sizeof(var));  
 return 0;  
  
}  
Enumların yüzde ile yazımı geçersizdir. Bu kodun derlenmesini etkilemese de çalışma zamanı içinde hataya sebep olur.  
#include <stdio.h>  
  
typedef enum{  
 Male, Female  
}SEX;  
  
int main()  
{  
 SEX var = Male;  
 SEX var1 = Female;  
 printf("%s %s",var, var1);  
 return 0;  
}  
İlk eleman ilk değer verilmediği sürece her zaman 0 dan başlar. Kısacası ikinci elemanın değeri hiçbir zaman ilk elemanı etkilemez ancak tersi yanlış bir ifadedir.  
  
#include <stdio.h>  
  
typedef enum{  
 Male, Female=-2  
}SEX;  
  
int main()  
{  
 SEX var = Male;  
 SEX var1 = Female;  
 printf("%d %d",var, var1);  
 return 0;  
}  
  
Aşağıdaki örnekte bir kart oyunu için örnek bir enum tanımlanması yapılmıştır. Bu tanımlama Poker oyunu için özeldir isminden de anlaşılacağı gibi ancak elbette oyundan oyuna farklılık gösterebilir.

typedef enum{HEART,DIAMOND,SPADE,CLUB}SUIT;  
  
typedef enum{  
 TWO=2,  
 THREE,  
 FOUR,  
 FIVE,  
 SIX,  
 SEVEN,  
 EIGHT,  
 NINE,  
 TEN,  
 JACK,  
 QUEEN,  
 KING,  
 ACE,  
}POKER;