Nesne Yönelimli Programlama

Yapılar ve Birlikler

Yapılar (Struct) ile Veri Gruplama

C++ dilinde `struct` (yapı), birden fazla veriyi tek bir birim halinde gruplandıran özel bir veri tipidir. Yapılar, birden fazla veri alanını (üyeyi) tek bir koleksiyon içinde tutmamıza olanak sağlar ve genellikle farklı türdeki verilerin bir arada kullanılması gerektiğinde kullanılır.

Bir Yapı, tek bir ad altında bir araya getirilmiş bir grup veri öğesidir. Üye olarak bilinen bu veri öğeleri farklı türlere ve farklı uzunluklara sahip olabilir.

Yapılar, **nesne yönelimli programlama (OOP)** kavramlarının daha öncesinde, C dilinde veri gruplama için yaygın olarak kullanılmıştır. C++ dilinde ise, **nesnelerin** tanımlanmasının dışında, **veri kapsülleme** ve **soyutlama** gibi OOP özellikleriyle daha geniş kapsamlı kullanımlar kazanmıştır.

Söz Dizimi 1:

```
struct type_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
} object_names;
```

Söz Dizimi 2:

```
struct name {
  member1_type member1_name;
  member2_type member2_name;
  member3_type member3_name;
};
```

Yapıların Temel Özellikleri

- **Bütünleşik Veri Yönetimi:** Yapılar, farklı türdeki verileri bir arada tutmak için kullanılır. Örneğin, bir `Employee` yapısı hem ad, soyad, maaş gibi farklı türdeki verileri bir arada depolayabilir.
- **Bellek Yapısı:** Yapılar, üyelerinin her birine belirli bir bellek bloğu ayırır. Her üye bağımsız olarak bellekte saklanır ve yapının toplam belleği, üyelerinin toplam belleğine eşittir.
- **Üyelerin Erişimi:** Yapı üyeleri, varsayılan olarak **public** erişim belirleyicisine sahiptir. Bu, dışarıdan doğrudan erişilebileceği anlamına gelir.

Örnek:

```cpp
#include <iostream>
struct koordinat {
  int x;
  int y;
};
int main() {
  // k1 değişlenlerinin tanımlanması
  struct koordinat k1;
  k1.x = 2;
  k1.y = 2;
  // k1 değişlenlerinin tanımlanması ve başlatılması
  struct koordinat k2 = { 0, 1 };
  std::cout << "x1 = " << k1.x << ", y1 = " << k1.y << "\n";
  std::cout << "x2 = " << k2.x << ", y2 = " << k2.y << "\n";
  return 0;
}
```

Yapı Dizisi

Elemanları yapı tipinde olan bir diziye yapı dizisi denir. Genellikle programda birden fazla yapı değişkenine ihtiyaç duyduğumuzda kullanışlıdır.

`struct yapi_adi dizi_adi [dizi_eleman_sayisi];` şeklinde tanımlanır.

```cpp
#include <iostream>
struct koordinatlar {
  int x;
  int y;
};
int main() {
  struct koordinatlar k[4];
  int i;
  for (i = 0; i < 4; i++)
    std::cin >> k[i].x >> k[i].y;

  for (i = 0; i < 4; i++) {
    std::cout << "x" << i+1 << "=" << k[i].x << "\t";
    std::cout << "y" << i+1 << "=" << k[i].y << "\n";
  }
  return 0;
}
```

Yapılar İçinde Yapılar

Bir yapı elemanı (üye) karmaşık ya da basit olabilir. Basit elemanlar `int, float, char` veya `double` gibi C++'ın temel veri türlerinden herhangi biri olabilir. Bununla birlikte, bir yapı, diziler, yapılar vb. gibi kendisi karmaşık olan bir unsurdan oluşabilir.

Böylece, bir yapının bir elemanı bir dizi veya kendi içinde bir yapı olabilir. Bu tür karmaşık elemanlardan oluşan bir yapıya karmaşık yapı denir.

Örnek:

```cpp
#include <iostream>
#include <string>

struct Adres {
    std::string sokak;
    std::string sehir;
    std::string postaKodu;
};

struct Ogrenci {
    int numara;
    std::string isim;
    Adres adres;  // İç içe yapı kullanımı
};
```

```cpp
int main() {
    // Ogrenci yapısından bir nesne oluşturuyoruz
    Ogrenci ogrenci1;
    ogrenci1.numara = 12345;
    ogrenci1.isim = "Ahmet Yılmaz";

    // Adres bilgilerini atıyoruz
    ogrenci1.adres.sokak = "Atatürk Cad.";
    ogrenci1.adres.sehir = "İstanbul";
    ogrenci1.adres.postaKodu = "34000";

    // Bilgileri ekrana yazdırıyoruz
    std::cout << "Öğrenci Numarası: " << ogrenci1.numara << std::endl;
    std::cout << "Öğrenci İsmi: " << ogrenci1.isim << std::endl;
    std::cout << "Sokak: " << ogrenci1.adres.sokak << std::endl;
    std::cout << "Şehir: " << ogrenci1.adres.sehir << std::endl;
    std::cout << "Posta Kodu: " << ogrenci1.adres.postaKodu << std::endl;

    return 0;
}
```

Yapılar İçinde Pointerlar

Tıpkı diğer veri türlerinde olduğu gibi, yapılar da kendi türlerinde bir işaretçi (pointer) ile gösterilebilir. Bu, özellikle büyük veri yapıları ile çalışırken veya bellekte yer tasarrufu sağlamak gerektiğinde oldukça kullanışlıdır.

Yapı işaretçileri, yapıyı bellekte işaret eder ve doğrudan bu adres üzerinden verilere erişim sağlar.

Örnek :

```cpp
#include <iostream>
#include <string>

struct Ogrenci {
    int numara;
    std::string isim;
};

int main() {
    // Ogrenci türünde bir nesne oluşturuyoruz
    Ogrenci ogrenci1;
    ogrenci1.numara = 12345;
    ogrenci1.isim = "Ahmet Yılmaz";

    // Ogrenci türünde bir işaretçi tanımlıyoruz ve ogrenci1'in adresini atıyoruz
    Ogrenci* ogrenciPtr = &ogrenci1;

    // İşaretçi üzerinden yapının üyelerine erişiyoruz
    std::cout << "Öğrenci Numarası: " << ogrenciPtr->numara << std::endl;
    std::cout << "Öğrenci İsmi: " << ogrenciPtr->isim << std::endl;

    return 0;
}
```

Örnek :

```cpp
#include <iostream>
#include <string>

struct Ogrenci {
    int numara;
    std::string isim;
};

int main() {
    // Dinamik olarak bir Ogrenci nesnesi oluşturuyoruz
    Ogrenci* ogrenciPtr = new Ogrenci; // `new` ile bellekte alan tahsis edilir

    // İşaretçi üzerinden yapının üyelerine değer atıyoruz
    ogrenciPtr->numara = 12345;
    ogrenciPtr->isim = "Ahmet Yılmaz";

    // İşaretçi üzerinden yapının üyelerine erişiyoruz ve ekrana yazdırıyoruz
    std::cout << "Öğrenci Numarası: " << ogrenciPtr->numara << std::endl;
    std::cout << "Öğrenci İsmi: " << ogrenciPtr->isim << std::endl;

    // Bellekten tahsis edilen alanı serbest bırakıyoruz
    delete ogrenciPtr;

    return 0;
}
```

Yapılar ve Yapıcılar

Yapılar, fonksiyonel yapılar olabilir. C++'da, `struct`'lar için **yapıcı (constructor)** fonksiyonlar tanımlanabilir. Bu, nesne oluşturulurken üyelerin başlangıç değerleri atamak için kullanılır.

Constructor Örneği:

```
struct Calisan {
    int TCKN;
    std::string Adi;
    double Maas;

    // Constructor
    Calisan(int _Tckn, std::string _ad, double _maas) {
        TCKN = _Tckn;
        Adi = _ad;
        Maas = _maas;
    }
};
```

Yapıcı kullanarak, bir `Calisan` nesnesi oluşturulurken üyelerin başlangıç değerlerini belirlemek mümkündür.

```
Calisan Personel_1(101, "Ahmet", 50000.0);
```

Yapıcı (constructor)

C++ dilinde **constructor** (yapıcı), bir yapının (`struct` veya `class`) **başlatılması** ve **ilk değerlerinin atanması** için kullanılan özel bir fonksiyondur. Constructor, bir yapıdan nesne yaratıldığında **otomatik olarak çağrılır** ve nesnenin başlangıç durumunu ayarlar.

Yapılarda Constructor Özellikleri

1. **İsmi yapı ile aynı olmalıdır.** Constructor, yapı ismiyle aynı isme sahip bir fonksiyondur. Bu sayede, `struct` veya `class` nesnesi oluşturulduğunda otomatik olarak çağrılır.

2. **Geri dönüş türü yoktur.** Constructor'ların dönüş türü yoktur; bu yüzden `void` veya başka bir dönüş türü belirtilmez.

3. **Nesne oluşturulurken otomatik olarak çağrılır.** Constructor, nesne bellekte yaratıldığında veya başlatıldığında otomatik olarak çağrılır. Böylece nesnenin ilk durumunun ayarlanması sağlanır.

Constructor Çeşitleri

C++ dilinde constructor'ların çeşitli türleri vardır:

- **Varsayılan Constructor (Default Constructor):** Parametre almayan bir yapıcıdır ve varsayılan değerlerle başlatma yapar.
- **Parametreli Constructor:** Yapının üyelerini belirli değerlere başlatmak için parametre alan yapıcıdır.
- **Kopya Constructor (Copy Constructor):** Bir nesneyi başka bir nesneden kopyalayarak başlatmak için kullanılır.

```cpp
#include <iostream>

struct Point {
    int x;
    int y;

    // Varsayılan Constructor (parametresiz)
    Point() : x(0), y(0) {
        std::cout << "Default Constructor called\n";
    }

    // Parametreli Constructor
    Point(int xVal, int yVal) : x(xVal), y(yVal) {
        std::cout << "Parameterized Constructor called\n";
    }

    // Kopya Constructor
    Point(const Point& other) : x(other.x), y(other.y) {
        std::cout << "Copy Constructor called\n";
    }
};
```

```cpp
int main() {
    // Varsayılan constructor kullanımı
    Point p1;
    std::cout << "p1: (" << p1.x << ", " << p1.y << ")\n";

    // Parametreli constructor kullanımı
    Point p2(5, 10);
    std::cout << "p2: (" << p2.x << ", " << p2.y << ")\n";

    // Kopya constructor kullanımı
    Point p3 = p2;
    std::cout << "p3: (" << p3.x << ", " << p3.y << ")\n";

    return 0;
}
```

Yıkıcı (Destructor)

Destructor (yıkıcı), bir nesne bellekte işini bitirdiğinde çağrılan özel bir fonksiyondur. Bellek temizliği, kaynak serbest bırakma (örneğin, açılmış dosyalar, dinamik olarak ayrılmış bellek gibi) gibi işlemler destructor içinde yapılır. Destructor, nesne yaşam döngüsünün sonunda otomatik olarak çalışır.

Yapılarda (struct) bir destructor tanımlamak için şunlara dikkat edilir:

1. Destructor ismi yapının ismiyle aynı olmalı ve başına `~` karakteri eklenmelidir.
2. Destructor, parametre almaz ve dönüş değeri olmaz.
3. Destructorlar, **otomatik olarak** çağrılır, doğrudan çağrılmazlar.

Örnek:

```cpp
#include <iostream>
#include <string>

struct Calisan {
    int TCKN;
    std::string Adi;
    double Maas;
    // Constructor
    Calisan(int _Tckn, std::string _ad, double _maas) {
        TCKN = _Tckn;
        Adi = _ad;
        Maas = _maas;
        std::cout << "Calisan nesnesi oluşturuldu: " << Adi << std::endl;
    }
    // Destructor
    ~Calisan() {
        std::cout << "Calisan nesnesi yok ediliyor: " << Adi << std::endl;
    }
};

int main() {
    Calisan calisan1(12345678901, "Ahmet Yılmaz", 5000.0);
    // Scope sonunda destructor otomatik olarak çağrılacak
    return 0;
}
```

Örnek:

```cpp
#include <iostream>
#include <string>

struct Calisan {
    int TCKN;
    std::string Adi;
    double Maas;

    // Constructor
    Calisan(int _Tckn, std::string _ad, double _maas) {
        TCKN = _Tckn;
        Adi = _ad;
        Maas = _maas;
        std::cout << "Calisan nesnesi oluşturuldu: " << Adi << std::endl;
    }

    // Destructor
    ~Calisan() {
        std::cout << "Calisan nesnesi yok ediliyor: " << Adi << std::endl;
    }
};
```

```cpp
int main() {
    // new ile dinamik bellek tahsisi yapılıyor
    Calisan* calisan1 = new Calisan(12345678901, "Ahmet Yılmaz", 5000.0);

    // Nesneyle ilgili işlemler burada yapılabilir

    // delete ile bellekte ayrılan alanı serbest bırakıyoruz (destructor çağrılır)
    delete calisan1;

    return 0;
}
```

Birlikler (Union)

C++'da `union` (birlik), aynı bellek alanını paylaşan, birbirinden farklı türdeki verilerin tek bir veri yapısında toplanmasını sağlayan bir özelliktir. Birlikler, **belirli bir anda yalnızca bir üyenin kullanılmasına olanak tanır**. Bu özellik, belleğin verimli bir şekilde kullanılmasını sağlamak için kullanılır. C++'daki `union` veri tipi, **yapılar** (struct) ile benzer şekilde çalışır, ancak tüm üyelerinin aynı bellek alanını paylaştığı için bellek yönetimini farklı bir biçimde ele alır.

Birliklerin Temel Özellikleri

- **Bellek Paylaşımı:** Birlikte tanımlanan tüm üyeler, aynı bellek alanını paylaşır. Bu nedenle, yalnızca bir üye aktif olarak kullanılabilir.
- **Bellek Verimliliği:** Bellek açısından verimli bir yapıdır çünkü üyelerin hepsi aynı bellek alanını kullanır, ancak yalnızca en büyük üye kadar bellek ayrılır.
- **Erişim:** Birlik üyelerine sadece bir tanesi aynı anda erişilebilir. Diğer üyeler geçerli değildir ve bellek içeriği o an kullanılan üye ile değişir.

Birlik Tanımı

```
union Veri {
    int intDeger;
    float floatDeger;
    char charDeger;
};
```

- `Veri` adında bir birlik tanımlanmış ve üç farklı türde üye içermektedir: `int`, `float` ve `char`.
- Ancak bu üç üye **aynı bellek alanını paylaşır**.

Birlik Nesnesi Kullanımı:

Birlikten bir nesne oluşturduğumuzda, bu nesne içinde yalnızca bir üye kullanılabilir:

```
Veri veri1;

veri1.intDeger = 10;  // intDeger'e değer atandı
std::cout << "Tam Sayi: " << veri1.intDeger << std::endl;  // Değer doğru bir şekilde yazdırılır

data.floatDeger = 5.25f;  // floatDeger'ya değer atandı
std::cout << "Float: " << veri1.floatDeger << std::endl;  // integer değeri artık geçerli değildir

std::cout << "Tam Sayi: " << veri1.intDeger << std::endl;  // Burada belirsiz bir sonuç alırız
```

Yukarıdaki örnekte, `intDeger` ve `floatDeger` farklı türler olmasına rağmen **aynı bellek alanını kullanır**. Bu nedenle, `floatDeger`'ya değer atandığında, `intDeger`'nın değeri geçerli olmayacaktır.

Bellek Kullanımı

Birliklerin üyeleri, aynı bellek alanını paylaştıkları için, birlik oluşturulduğunda yalnızca en büyük üye kadar bellek tahsis edilir. Örneğin, aşağıdaki kodda `float` türü 4 byte, `int` türü de 4 byte alır, ancak `union` yalnızca 4 byte bellek kullanır.

```
union Veri {
    int intDeger;   // 4 bytes
    float floatDeger; // 4 bytes
    char charDeger;  // 1 byte
};

Veri veri1;
```

Birlikler ve Yapıcılar (Constructors)

C++'da, `union` türünde bir yapıcı fonksiyon tanımlamak mümkündür, ancak yalnızca **ilk üye** için geçerlidir. Çünkü bir zamanda yalnızca bir üye kullanılabilir.

Birlikte Yapıcı Kullanma:

```
union Veri {
    int intDeger;
    float floatDeger;
    char charDeger;

    // Yapıcı
    Veri(int sayi) {
        intDeger = sayi;  // Sadece intDeger başlatabiliyoruz
    }
};
```

Bu örnekte, `union` yapıcısı sadece `intDeger` üyelerini başlatır. Bu, yapının özelliklerinden biridir çünkü yalnızca bir üye aktif hale gelebilir.

Örnek:

```
union Data {
    int intValue;
    float floatValue;
    char charValue;
};
```

```
Data data;
data.intValue = 10;
std::cout << "Integer: " << data.intValue << std::endl;
data.floatValue = 5.25f;
std::cout << "Float: " << data.floatValue << std::endl; // intValue'nin değeri artık geçerli değildir
```

Avantajları ve Kullanım Alanları

1. **Bellek Verimliliği**

Birlikler, aynı bellek alanını paylaşan üyeleri sayesinde belleği verimli bir şekilde kullanır. Bu, özellikle bellek sınırlı sistemlerde, mikrodenetleyicilerde veya taşınabilir cihazlarda önemlidir.

2. **Farklı Türlerde Veri Yönetimi**

Birlikler, aynı veri parçasını farklı türlerde temsil etmenizi sağlar. Örneğin, aynı veri parçası bir zamanlar `int`, bir zamanlar `float` veya `char` olabilir, ancak bunların her biri için farklı türde bellek tahsisi yapılmaz.

3. **Düşük Seviyeli Programlama**

Birlikler, düşük seviyeli yazılım geliştirme, özellikle **sistem programlama** ve **gömülü sistemler** için yaygın bir kullanıma sahiptir. Bu tür sistemlerde, belleği etkin bir şekilde kullanmak ve donanım ile doğrudan etkileşimde bulunmak gerekebilir.

Birliklerin Sınırlamaları ve Riskleri

1. **Yalnızca Bir Üye Aktif Olabilir**

Birlikler, aynı bellek alanını paylaştıkları için yalnızca bir üye aynı anda geçerli olabilir. Diğer üyeler üzerinde işlem yapmak, belirsiz ve hatalı sonuçlar doğurur.

2. **Hafıza Erişimi**

Birlikler, türler arasında geçiş yapılırken **hatalı bellek erişimlerine** yol açabilir. Bir üye üzerinde işlem yaparken, diğer üyelerin değerlerine erişmek mümkün değildir ve bu bellek hatalarına yol açabilir.

Bit Alanları (Bit Fields)

Bit alanları, yapılar veya birlikler içinde belirli sayıda bit ile sınırlı veri alanları oluşturmamızı sağlar. Bit alanları, özellikle belleğin çok kıymetli olduğu durumlarda faydalıdır.

Bit Alanı Tanımı

```
struct Flags {
    unsigned int isVisible : 1;
    unsigned int isEnabled : 1;
    unsigned int isAdmin : 1;
};
```

Kullanımı

- Bit alanları sadece `int`, `unsigned int`, `char` gibi tamsayı türlerinde kullanılabilir.
- Bit alanlarının bellek boyutu, tüm alanların toplamı kadardır, bu nedenle bellek verimli kullanılır.

Örnek:

```
Flags flags;
flags.isVisible = 1;
flags.isEnabled = 0;
flags.isAdmin = 1;

std::cout << "isVisible: " << flags.isVisible << std::endl;
std::cout << "isEnabled: " << flags.isEnabled << std::endl;
std::cout << "isAdmin: " << flags.isAdmin << std::endl;
```

Birlikler ve Bit Düzeyinde İşlemler

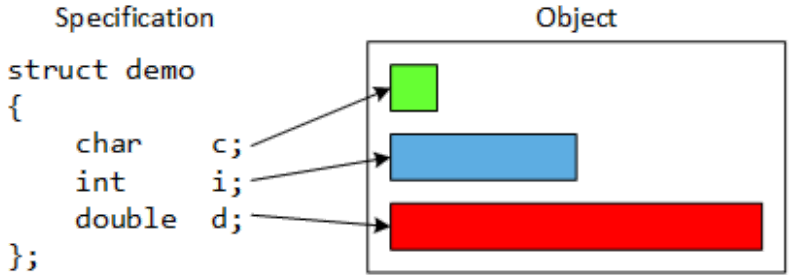
Birlikler, özellikle **bit düzeyindeki işlemler** için yaygın olarak kullanılır. Aşağıda, `union` kullanarak 32-bit değerleri 4 farklı bayta ayıran bir örnek gösterilmektedir:

```cpp
#include <iostream>
#include <cstdint>


union Data {
    uint32_t value;  // 32-bit değeri tek bir uint32_t olarak tut
    uint8_t bytes[4]; // 4 byte'a ayır
};

int main() {
Data data;
data.value = 0x12345678;

std::cout << "Byte 1: " << std::hex << (int)data.bytes[0] << std::endl;
std::cout << "Byte 2: " << std::hex << (int)data.bytes[1] << std::endl;
std::cout << "Byte 3: " << std::hex << (int)data.bytes[2] << std::endl;
std::cout << "Byte 4: " << std::hex << (int)data.bytes[3] << std::endl;

return 0;
}
```

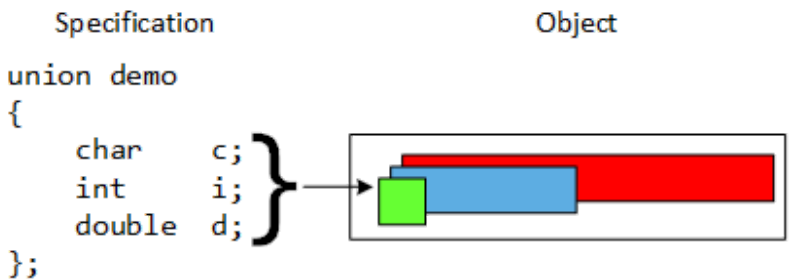
Bit Alanlarının Sınırlamaları

- Bit alanları, platformdan platforma farklılık gösterebilir.
- Bit alanlarına pointer ile erişim genellikle desteklenmez, doğrudan erişim gereklidir.

Struct



Union



C++'da `typedef`, `using`, ve `enum` Kavramları

C++ dilinde `typedef`, `using` ve `enum` önemli kavramlardır. Bu kavramlar, daha okunabilir, anlaşılır ve esnek kod yazılmasına yardımcı olur. Her biri farklı amaçlarla kullanılır, ancak hepsi C++ dilinde yazılım geliştirme sürecinde oldukça faydalıdır.

`typedef` Nedir?

C++'da `typedef`, mevcut bir türün başka bir isimle kullanılmasını sağlar. Bu, özellikle karmaşık türleri (örneğin, işaretçiler veya fonksiyon işaretçileri) daha basit ve anlamlı hale getirmek için kullanılır.

`typedef` Kullanımı

```
typedef int Sayi;  // Sayi ismi artık int türünü ifade eder
Sayi x = 10;  // x, int türünde bir değişken olarak kullanılır
```

Yukarıdaki örnekte, `typedef int Sayi;` ifadesi, `int` türüne yeni bir isim (`Sayi`) atamaktadır. Artık, `Sayi` kelimesi `int` ile eşdeğer olur.

Karmaşık Türler İçin `typedef`

Özellikle işaretçiler veya fonksiyon işaretçileri gibi karmaşık türlerde `typedef` kullanmak çok faydalıdır:

```
typedef int* IntPointer;  // int türü işaretçisi için yeni bir isim
IntPointer p = nullptr;  // p, int türünde bir işaretçidir
```

`using` Nedir?

C++11 ile tanıtılan `using` ifadesi, `typedef` ile benzer şekilde çalışır ancak genellikle daha modern ve daha anlaşılır bir yazım tarzı sağlar.

`using` Kullanımı

```
1   using Sayi = int;  // Sayi ismi artık int türünü ifade eder
2   Sayi x = 10;  // x, int türünde bir değişken olarak kullanılır
```

Karmaşık Türler İçin `using`

`using` ile karmaşık türlerin tanımlanması `typedef`'e benzer şekilde yapılabilir:

```
1   using IntPointer = int*;  // int türü işaretçisi için yeni bir isim
2   IntPointer p = nullptr;  // p, int türünde bir işaretçidir
```

`typedef` ve `using` arasındaki farklar genellikle stil ve okunabilirlik ile ilgilidir. `using` genellikle daha açık ve anlaşılır olduğu için tercih edilmektedir.

`enum` Nedir?

`enum`, **sınıflandırılmış sabitler** oluşturmak için kullanılan bir türdür. Genellikle bir dizi ilişkili sabit değeri adlandırmak için kullanılır. `enum`, belirli bir isim kümesine değerler atamayı sağlar ve bu değerler genellikle sıralıdır.

`enum` Kullanımı

```
enum Color {
    Red,    // Red = 0
    Green,  // Green = 1
    Blue    // Blue = 2
};
```

Yukarıdaki örnekte, `Color` adında bir enum tanımlanmıştır. `Red`, `Green` ve `Blue` değerleri sırasıyla `0`, `1` ve `2` değerlerine atanır.

`enum` ile Farklı Değerler Atama

```
enum Color {
    Red = 10,
    Green = 20,
    Blue = 30
};
```

Bu örnekte, `Red`, `Green`, ve `Blue` sabitleri belirli değerler alır.

`enum class` (C++11 ve Sonrası)

C++11 ile tanıtılan `enum class`, daha güvenli bir enum türüdür. Her bir enum sınıfının kendi isim alanına sahip olmasını sağlar.

```
enum class Color {
    Red = 1,
    Green = 2,
    Blue = 3
};

Color c = Color::Red;  // `Color::Red` şeklinde erişilir
```

`typedef`, `using`, ve `enum` ile `struct` ve `union` İlişkisi

1. `typedef` ve `using` ile `struct` İlişkisi

`typedef` ve `using` kullanılarak, `struct` türleri daha anlamlı ve daha kısa isimlerle tanımlanabilir. Bu, özellikle büyük ve karmaşık veri yapılarını daha anlaşılır kılar.

`typedef` ile `struct` Kullanımı

```
typedef struct {
    int id;
    std::string name;
} Employee;  // Employee artık bu struct'ı ifade eder

Employee emp1;  // Employee tipinde bir değişken tanımlanabilir
emp1.id = 101;
emp1.name = "Ahmet";
```

`using` ile `struct` Kullanımı

```
using Employee = struct {
    int id;
    std::string name;
};

Employee emp1;  // Employee tipi ile bir değişken oluşturulabilir
emp1.id = 102;
emp1.name = "Ayşe";
```

2. `typedef`, `using` ve `enum` ile `struct` İlişkisi

`enum` ve `typedef` / `using`, `struct` içinde kullanılabilir. `enum` tipi bir `struct` içinde üyeler olarak tanımlanabilir ve `typedef` / `using` ile daha anlaşılır hale getirilebilir.

`enum` ile `struct` Kullanımı

```
enum class Department {
    HR = 1,
    IT = 2,
    Sales = 3
};

struct Employee {
    int id;
    std::string name;
    Department dept;  // enum sınıfı kullanılıyor
};

Employee emp1;
emp1.id = 103;
emp1.name = "Ali";
emp1.dept = Department::IT;  // enum değerine erişim
```

`typedef` ve `using` ile `enum` Kullanımı

```
typedef enum {
    Red = 1,
    Green = 2,
    Blue = 3
} Color;

struct Item {
    int id;
    Color color;  // typedef ile enum tipi kullanımı
};

Item item1;
item1.id = 104;
item1.color = Red;
```

Bu kullanım, özellikle enum türünün ve `struct` üyelerinin daha okunabilir olmasını sağlar.

3. `typedef`, `using` ve `enum` ile `union` İlişkisi

`typedef` ve `using`, `union` türleriyle de ilişkilidir ve aynı şekilde verimlilik sağlar. `enum` türü de `union` üyeleri olarak kullanılabilir.

`typedef` ile `union` Kullanımı

```
typedef union {
    int intValue;
    float floatValue;
    char charValue;
} Data;

Data data;
data.intValue = 10;  // intValue'ya değer atanır
```

`using` ile `union` Kullanımı

```
using Data = union {
    int intValue;
    float floatValue;
    char charValue;
};

Data data;
data.floatValue = 5.25f;  // floatValue'ya değer atanır
```

`enum` ile `union` Kullanımı

```
enum class DataType {
    IntType = 1,
    FloatType = 2,
    CharType = 3
};

union Data {
    int intValue;
    float floatValue;
    char charValue;
    DataType type;  // enum türü bir union üyesi olarak kullanılabilir
};

Data data;
data.intValue = 10;
data.type = DataType::IntType;
```

Bu örnekte, `enum` türü `union`'ın bir üyesi olarak kullanılmıştır. Bu tür kombinasyonlar, belleği daha verimli kullanmanın yanı sıra daha esnek veri yapıları oluşturmayı sağlar.

Hizalama(alignment) ve Dolgu (padding)

C++'da yapıların (struct) bellekte nasıl sıralandığı, **bellek verimliliği** ve **performans** açısından oldukça önemlidir. Yapılar, üyelerinin türlerine göre belirli bir düzenle bellekte yer alır. Bu düzenlemelerde kullanılan **dolgu (padding)** ve **hizalama (alignment)** kavramları, bellek yönetimini ve erişim hızını etkileyebilir.

Yapıların Bellekte Sıralanması

Bir yapı (`struct`), birden fazla üyeden oluşur ve bu üyeler bellekte sırasıyla yer alır. Ancak, her üyenin bellekte nasıl sıralandığı, bazı durumlarda **dolgu** ve **hizalama** işlemleri ile belirlenir. Yapılar, bellekteki üyelerin hizalanmasını optimize etmek için genellikle **belirli sınırlarla hizalanır**.

Temel Bellek Düzeni

Bir yapıdaki üyelerin sıralanması, genellikle şu şekilde olur:

1. Yapının ilk üyesi bellekte ilk sırada yer alır.
2. Diğer üyeler, sırasıyla yapı içerisinde tanımlandığı gibi yer alır.
3. Ancak, üyelerin türüne göre bellekte bazı boşluklar (dolgu) eklenebilir, bu da üyeler arasındaki bellek sırasını değiştirebilir.

```
1   struct Example {
2       char a;    // 1 byte
3       int b;     // 4 bytes
4       char c;    // 1 byte
5   };
```

Yapıların Bellekte Sıralanması ve Bellek Tahsisi

1. **Bellekte Art Arda Sıralanma**:

   - C++'da bir yapı (`struct`) tanımlandığında, içindeki her bir üye bellekte sıralı olarak yer alır. Bu, yapının bellekte kesintisiz bir blok olarak tahsis edildiği ve her üyenin, bir önceki üyenin hemen ardından yer aldığı anlamına gelir.

   - Yapının ilk üyesi bellekteki ilk adrese yerleştirilir, ardından gelen her üye, bir öncekinin kullandığı alanın sonuna yerleştirilir.

   - **Örneğin**, aşağıdaki `struct` yapısını ele alalım:

```
struct Kisi {
    char cinsiyet;      // 1 byte
    int yas;            // 4 byte
    double boy;         // 8 byte
};
```

`Kisi` yapısında ilk olarak `cinsiyet` değişkeni 1 byte kaplar. Ardından gelen `yas` 4 byte yer kapladığı için 1 byte'lık bir **dolgu** (padding) eklenir. `boy` ise 8 byte yer kaplar. Bu sayede tüm veri yapısı **alignment** (hizalama) kurallarına uygun hale getirilir.

Örnekler