Nesne Yönelimli Programlama

Diziler ve Karakter Dizileri

Dizilere Giriş

- **Dizi (Array):** Diziler, her değer için ayrı değişkenler bildirmek yerine birden fazla değeri tek bir değişkende saklamak için kullanılır.
- **Temel Yapı:**
  - **Veri tipi:** Dizinin sakladığı elemanların tipi.
  - **Boyut:** Dizide kaç eleman saklanacağını belirtir.

Örnek:

```
int sayilar[5]; // 5 elemanlı bir tamsayı dizisi
```

Dizi Boyutu ve Sabit Boyut Kısıtı

- Dizi boyutu **derleme zamanında** sabitlenir. Bu nedenle dizi tanımlandıktan sonra boyutu değiştirilemez.
- Dinamik diziler ile bu sınırlama aşılabilir. Ancak dinamik dizilerin kullanımı sırasında bellek yönetimi sorumluluğu geliştiriciye aittir.

Dizi ile İlişkili Sınır Hataları

- **Out-of-Bounds (Sınır Aşımı) Hatası:** Dizinin geçerli indeks aralığı dışında bir elemanına erişim hatasına yol açar.

Örnek:

```
int sayilar[3] = {1, 2, 3};
cout << sayilar[5];  // Tanımsız davranış (undefined behavior)
```

> **Not:** C++ dilinde sınır kontrolleri yapılmaz. Bu durum, kritik uygulamalarda bellek hatalarına yol açabilir.

Dizi Elemanlarına Erişim

- Dizinin elemanlarına **indis** ile ulaşılır. (İndisler 0'dan başlar.)

Örnek:

```
sayilar[0] = 10;  // İlk elemana 10 değeri atanır
cout << sayilar[0]; // İlk eleman yazdırılır: 10
```

- **Döngü ile Dizi İşleme:**

```
for (int i = 0; i < 5; i++) {
    cout << sayilar[i] << " ";
}
```

Dizileri Başlatma

- Dizi oluşturulurken değer atayabiliriz.

```
1   int sayilar[5] = {1, 2, 3, 4, 5};
```

- **Eksik Başlatma:** Geri kalan elemanlar 0 ile doldurulur.

```
1   int sayilar[5] = {1, 2}; // [1, 2, 0, 0, 0]
```

Çok Boyutlu Diziler

- **2D Diziler:** Matrisi temsil eder.

*	Sütün 1	Sütün 2	Sütün 3
Satır 1	1	2	3
Satır 2	4	5	6

```
int matris[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

- **Erişim:**

```
cout << matris[1][2]; // 6 değerini yazdırır
```

Karakter Dizileri (C-String)

- **Karakter Dizisi:** Bir karakter dizisi, karakterlerin sıralı bir yapısıdır.

```
char isim[6] = {'A', 'h', 'm', 'e', 't', '\0'}; // NULL ile biter
```

- Alternatif olarak:

```
char isim[] = "Ahmet"; // Otomatik olarak '\0' eklenir
```

Karakter Dizilerinde İşlemler

- **strlen():** Karakter dizisinin uzunluğunu bulur (null karakteri hariç).

```
cout << strlen(isim); // 5 yazdırır
```

- **strcmp():** İki karakter dizisini karşılaştırır.

```
strcmp("Ahmet", "Ahmet"); // 0 döner (eşit)
```

Diziler ve Fonksiyonlar

- **Fonksiyonlara Diziyi Geçirme:** Dizi, referans olarak aktarılır.

```
void yazdir(int dizi[], int boyut) {
    for (int i = 0; i < boyut; i++) {
        cout << dizi[i] << " ";
    }
}
```

Kullanım:

```
int sayilar[3] = {10, 20, 30};
yazdir(sayilar, 3); // 10 20 30
```

Karakter Dizileri ile Fonksiyonlar

- Fonksiyonlara **karakter dizisi** geçirme:

```
void mesajYazdir(char mesaj[]) {
    cout << mesaj;
}
```

Kullanım:

```
char mesaj[] = "Merhaba";
mesajYazdir(mesaj); // Merhaba
```

Dizi Kopyalama ve Atama

- Diziler **doğrudan atama** işlemi ile kopyalanamaz.

Yanlış Kullanım:

```
int dizi1[3] = {1, 2, 3};
int dizi2[3];
dizi2 = dizi1;  // HATA! Direkt atama yapılmaz.
```

Doğru Kullanım (Döngü ile Kopyalama):

```
for (int i = 0; i < 3; i++) {
    dizi2[i] = dizi1[i];
}
```

> **Not:** C++ dilinde diziler, doğrudan kopyalama işlemini desteklemez. Bunun yerine döngüler veya `std::copy` fonksiyonu kullanılabilir.

Diziler Neden Tercih Edilir?

Diziler, **veri yönetimi** açısından büyük kolaylık sağlar. Aynı veri tipindeki çok sayıda veriyi **tek bir yapıda** tutmak, hem kodu daha okunabilir kılar hem de performans açısından avantaj sunar. İşte dizilerin tercih edilme sebeplerini açıklayan bazı ana noktalar:

1. Kolay ve Düzenli Veri Yönetimi

- **Sorun:** Her bir değeri ayrı bir değişkende saklamak, çok sayıda veri ile çalışırken zorlaşır.
  - Örneğin: Bir sınıftaki 50 öğrencinin notlarını ayrı değişkenlerde saklamak hem hataya açık hem de karmaşıktır:

```
int not1, not2, not3, ..., not50;  // Karmaşık ve yönetimi zor
```

- **Çözüm:** Dizi kullanarak veriyi **tek bir yapıda** saklamak:

```
int notlar[50];  // Tek yapıda 50 öğrencinin notları
```

2. Kodun Daha Temiz ve Okunabilir Olması

- Kodda **aynı veri türünde** birden fazla değişken kullanmak, hem okunabilirliği hem de sürdürülebilirliği artırır.
- Dizi kullanımı ile **döngüler** sayesinde daha sade ve anlaşılır kodlar yazılır.

Örneğin, 50 öğrencinin notlarının toplamını bulmak:

```
1   int toplam = 0;
2   for (int i = 0; i < 50; i++) {
3       toplam += notlar[i];
4   }
```

Bu yapı, manuel toplam hesaplamaya göre çok daha temiz ve **hata riskini azaltır**.

3. Dinamik Verilerle Kolay İşlem Yapma

- Diziler, döngüler ve fonksiyonlarla rahatlıkla işlenebilir. Bu, çok büyük veri kümelerinde bile hızlı işlem yapmayı sağlar.

Örneğin, **sensor verilerini** işlemek:

```
float sensorVerileri[10];
for (int i = 0; i < 10; i++) {
    cout << "Sensor " << i + 1 << " verisini girin: ";
    cin >> sensorVerileri[i];
}
```

Her sensör için ayrı değişken bildirmek yerine tek bir diziyle verileri yönetmek çok daha pratiktir.

4. Performans ve Bellek Yönetimi Avantajı

- **Statik diziler**, bellekte **ardışık olarak saklanır**. Bu sayede belleğe erişim daha hızlıdır, çünkü CPU **önbelleği (cache)** ardışık veri erişimini optimize eder.
- **Bellek yönetimi:** Tek tek değişken oluşturmak yerine dizi yapısı kullanarak belleğin daha verimli kullanılması sağlanır.

Dizilerin Bellekte Temsili

- İlk elemanın adresi dizinin **başlangıç adresi** olur ve diğer elemanlar birer **offset** ile bulunur.

Örneğin:

- `int sayilar[3] = {10, 20, 30};`
  - `sayilar[0]` : 0x1000
  - `sayilar[1]` : 0x1004 (4 byte sonrası)
  - `sayilar[2]` : 0x1008 (8 byte sonrası)

> **Not:** `sizeof(int)` = 4 byte olduğu için her eleman arası fark 4 byte'tır.

5. İşlevselliğin Artması – Döngülerle İşlem Yapma

- Dizi yapıları, **döngüler ve fonksiyonlarla** birlikte kullanıldığında işlevselliği artırır. Böylece aynı işlemi tekrar tekrar yapmak yerine **otomatikleştirilebilir**.

Örneğin, 100 kişilik bir ekipte maaş artışı hesaplamak:

```
float maaslar[100];
for (int i = 0; i < 100; i++) {
    maaslar[i] *= 1.1;  // Her maaşı %10 artır
}
```

6. Kolay Fonksiyon Kullanımı

- Diziler, fonksiyonlara **parametre olarak** aktarılabilir. Bu, aynı fonksiyonla birçok veri üzerinde işlem yapmayı sağlar.

Örneğin:

```
void diziYazdir(int dizi[], int boyut) {
    for (int i = 0; i < boyut; i++) {
        cout << dizi[i] << " ";
    }
}
```

Bu şekilde, aynı fonksiyonla birçok farklı diziyi ekrana yazdırabilirsiniz.

Örnek Uygulamalar

Örnek 1: Dizi Elemanlarının Toplamı

```
int toplam(int dizi[], int boyut) {
    int toplam = 0;
    for (int i = 0; i < boyut; i++) {
        toplam += dizi[i];
    }
    return toplam;
}
```

Örnek 2: Girilen Metni Ters Çevirme

```
void tersCevir(char str[]) {
    int uzunluk = strlen(str);
    for (int i = uzunluk - 1; i >= 0; i--) {
        cout << str[i];
    }
}
```

Kullanım:

```
char isim[] = "Ahmet";
tersCevir(isim); // temhA
```

`std::array` ile Modern Dizi Kullanımı

- C++'ın modern standartları ile birlikte `std::array` sınıfı, klasik C tarzı dizilerin yerini alır. Bu yapı, daha güvenli ve fonksiyonel bir dizi kullanımı sağlar.

Örnek – `std::array`:

```cpp
#include <array>
#include <iostream>
using namespace std;

array<int, 3> sayilar = {1, 2, 3};

int main() {
    for (int i = 0; i < sayilar.size(); i++) {
        cout << sayilar[i] << " ";
    }
    return 0;
}
```

- **Avantajları:**
  - **Boyut bilgisi:** `size()` fonksiyonu ile dizinin boyutuna erişilebilir.

Dizi İle Çalışırken Kullanışlı Fonksiyonlar

- `std::copy` : Bir diziyi başka bir diziye kopyalamak için.

```cpp
#include <iostream>
#include <algorithm>  // std::copy için gerekli
#include <iterator>   // std::begin ve std::end için gerekli

int main() {
    // Kaynak ve hedef diziler
    int kaynak[] = {1, 2, 3, 4, 5};
    int hedef[5];

    // std::copy kullanarak kaynak diziyi hedefe kopyala
    std::copy(std::begin(kaynak), std::end(kaynak), std::begin(hedef));

    // Hedef dizinin elemanlarını ekrana yazdır
    std::cout << "Hedef Dizi: ";
    for (int i : hedef) {
        std::cout << i << " ";
    }

    return 0;
}
```

- `std::fill`: Tüm dizi elemanlarını aynı değerle doldurur.

```cpp
#include <algorithm>

int dizi[5];
std::fill(dizi, dizi + 5, 10);  // Tüm elemanlara 10 atanır
```

Array

`std::array`, **C++11** ile birlikte eklenen, sabit boyutlu ve tür güvenli (type-safe) bir dizi sınıfıdır. **Standart kütüphanede (** `<array>` **)** bulunur ve klasik C dizilerinden (**raw arrays**) daha güvenlidir.

`std::array` Sınıfı

`std::array`, C++'ta **sabit boyutlu** ve **tür güvenli** bir dizi yapısıdır. **C++11** ile eklenmiştir ve klasik C dizilerine kıyasla daha güvenli bir alternatif sunar. `<array>` başlık dosyasında tanımlıdır.

Neden `std::array` Kullanılır?

- Klasik C dizileri ile bellek sınır hataları oluşabilir.
- `std::array`, boyut bilgisi ve sınır kontrolleri sunarak güvenlik sağlar.
- **Sabit boyutlu** verilere ihtiyaç duyulduğunda kullanılır.
- **STL algoritmalarıyla** uyumludur.

Başlık Dosyası ve Temel Kullanım

```
#include <array>  // std::array kullanımı için gerekli

std::array<int, 5> myArray = {1, 2, 3, 4, 5};
```

Üye Fonksiyonlar

Fonksiyon	Açıklama
size()	Dizinin eleman sayısını döndürür.
at(index)	Belirtilen indeksin elemanına güvenli erişim sağlar.
front()	İlk elemanı döndürür.
back()	Son elemanı döndürür.
fill(value)	Bütün elemanları belirtilen değer ile doldurur.
data()	Ham gösterici (pointer) döndürür.

Kullanım Örnekleri

Temel Kullanım

```cpp
#include <iostream>
#include <array>
using namespace std;

int main() {
    std::array<int, 5> numbers = {1, 2, 3, 4, 5};

    // Elemanlara erişim
    cout << "İlk eleman: " << numbers.front() << endl;
    cout << "Son eleman: " << numbers.back() << endl;

    // Boyut bilgisi
    cout << "Dizi boyutu: " << numbers.size() << endl;

    return 0;
}
```

Çıktı:

```
İlk eleman: 1
Son eleman: 5
Dizi boyutu: 5
```

`at()` Fonksiyonu ile Güvenli Erişim

```cpp
#include <iostream>
#include <array>
using namespace std;

int main() {
    std::array<int, 3> arr = {10, 20, 30};

    try {
        cout << arr.at(2) << endl;   // 30
        cout << arr.at(5) << endl;   // Hata: out_of_range
    } catch (const out_of_range& e) {
        cerr << "Hata: " << e.what() << endl;
    }

    return 0;
}
```

Çıktı:

```
Hata: array::at: __n (which is 5) >= _Nm (which is 3)
```

`fill()` Fonksiyonu ile Dizi Doldurma

```cpp
#include <iostream>
#include <array>
using namespace std;

int main() {
    std::array<int, 4> arr;
    arr.fill(7);  // Tüm elemanları 7 ile doldur

    for (int i : arr) {
        cout << i << " ";
    }
    return 0;
}
```

Çıktı:

```
Kodu kopyala
7 7 7 7
```

Bellek Yönetimi

- `std::array` nesneleri **stack** üzerinde saklanır ve hızlı bellek erişimi sağlar.
- Boyutu sabittir ve dinamik olarak değiştirilemez.
- Dinamik boyut yönetimi gereken durumlarda `std::vector` tercih edilir.

`std::array` ile Kapsam Kontrolü

`std::array` kullanıldığında, fonksiyonlar bittiğinde dizinin kapsadığı bellek otomatik olarak serbest bırakılır. Ek bir **yıkıcı (destructor)** çağrısı gerekmez.

C Dizilerine Göre Avantajları

- **Boyut bilgisi korunur**: `std::array` ile boyut derleme zamanında belirlenir.
- **Sınır kontrolleri** yapılabilir: `at()` fonksiyonu ile güvenli erişim sağlar.
- **STL algoritmaları** ile birlikte çalışabilir.

Vektörler (std::vector) – Yeniden Boyutlandırılabilir Diziler

C++'ta **standart diziler (arrays)** sabit boyutludur. Diziyi tanımlarken boyutunu belirtmeniz gerekir ve çalışma zamanında bu boyut değiştirilemez. Ancak **std::vector** gibi **yeniden boyutlandırılabilir diziler** sayesinde, dinamik veri ekleme ve çıkarma işlemleri kolayca yapılabilir. Vektörler, C++'ın **Standard Template Library (STL)** kapsamında sağlanan ve dizilere alternatif sunan güçlü veri yapılarıdır.

Vektörlerin Avantajları

1. **Dinamik Bellek Yönetimi:**
   - Vektörler, çalışma zamanında boyutlarını **otomatik olarak genişletip daraltabilir**.
   - Eklenen yeni elemanlarla birlikte, vektörün boyutu içsel olarak artırılır (genellikle iki katına çıkar).
2. **Eleman Ekleme ve Çıkarma:**
   - Dizilerde eleman ekleme ve çıkarma işlemleri manuel olarak yapılsa da, vektörler bunu **hazır fonksiyonlar** ile yapar.
3. **Esnek Kullanım:**
   - Dizilerde önceden boyut belirleme zorunluluğu varken, vektörler veri büyüklüğü değişse bile çalışmaya devam eder.
4. **STL Fonksiyonları ile Kolay Entegrasyon:**
   - Vektörler, STL algoritmaları ile kolayca kullanılabilir (örneğin `sort()`, `find()` gibi).

Vektörlerin Temel Kullanımı

Öncelikle, vektör kullanmak için `<vector>` **başlık dosyasını** dahil etmeniz gerekir.

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    // Boş bir tamsayı vektörü tanımlıyoruz
    vector<int> sayilar;

    // Vektöre eleman ekleme
    sayilar.push_back(10);
    sayilar.push_back(20);
    sayilar.push_back(30);

    cout << "Vektör elemanları: ";
    for (int i = 0; i < sayilar.size(); i++) {
        cout << sayilar[i] << " ";
    }
    cout << endl;

    // Eleman çıkarma (son elemanı silme)
    sayilar.pop_back();
```

```
int main() {
//devam
    cout << "Son eleman silindikten sonra: ";
    for (int i = 0; i < sayilar.size(); i++) {
        cout << sayilar[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Çıktı:

```
Vektör elemanları: 10 20 30
Son eleman silindikten sonra: 10 20
```

Vektör İşlemleri

1. Eleman Ekleme – `push_back()`

- Vektörün sonuna yeni bir eleman eklemek için kullanılır.

```
1   sayilar.push_back(40);
```

2. Eleman Silme – `pop_back()`

- Vektörün **sonundaki elemanı** siler.

```
1   sayilar.pop_back();
```

3. Boyut Öğrenme – `size()`

- Vektörün kaç eleman içerdiğini döner.

```
cout << "Vektör boyutu: " << sayilar.size() << endl;
```

4. Eleman Güncelleme ve Erişim

- Diziler gibi, vektör elemanlarına **indeksleme** ile erişilebilir ve güncellenebilir.

```
sayilar[1] = 50;  // İkinci elemanı 50 yap
cout << "İkinci eleman: " << sayilar[1] << endl;
```

Vektörlerin Özellikleri

1. **Bellek Yönetimi:**
   - Vektörler kapasitesi dolduğunda, **arka planda yeni bir bellek bloğu** ayırır ve tüm veriyi yeni bloğa taşır.
   - Genişleme maliyeti olsa da, bu işlemler C++ tarafından optimize edilmiştir.
2. **İçerisindeki Elemanlar Ardışık Olarak Saklanır:**
   - Vektör elemanları, bellekte ardışık olarak tutulur. Bu sayede, diziler gibi hızlı erişim sağlanır.

3. **Eleman Eklemek İçin Alternatif – `insert()`**

   - Belirli bir pozisyona eleman ekleyebilirsiniz:

```
sayilar.insert(sayilar.begin() + 1, 25);  // İkinci pozisyona 25 ekler
```

4. **Tüm Elemanları Silmek – `clear()`**

   - Vektörün içindeki tüm elemanları siler.

```
sayilar.clear();
cout << "Vektör boyutu: " << sayilar.size() << endl;  // Çıktı: 0
```

Örnek: Sensör Verilerini Toplamak

Bir mekatronik sistemde, 10 farklı sensörden gelen veriyi dinamik olarak saklamak ve analiz etmek isteyebiliriz. Sensör sayısı değişebilir; bu durumda **vektör kullanımı** idealdir.

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<float> sensorVerileri;
    float veri;
    cout << "Sensör verilerini girin (çıkmak için -1):" << endl;
    while (true) {
        cin >> veri;
        if (veri == -1) break;
        sensorVerileri.push_back(veri);  // Yeni veri ekleniyor
    }
    cout << "Toplam sensör verisi: " << sensorVerileri.size() << endl;
    cout << "Veriler: ";
    for (float v : sensorVerileri) {
        cout << v << " ";
    }
    cout << endl;

    return 0;
}
```

Vektör (std::vector) ve Dizi (Array) Arasındaki Farklar

C++ dilinde **vektörler** (`std::vector`) ve **diziler** farklı veri yapıları olarak kullanılır. Her iki yapı da veri elemanlarını ardışık olarak saklar. Ancak kullanım amaçları ve özellikleri bakımından farklılık gösterirler. İşte vektörler ve diziler arasındaki temel farklar:

1. Boyut

- **Dizi:** Boyut, tanımlanırken belirlenir ve **sabit** kalır.

```
int dizi[5]; // Boyutu 5 olan sabit bir dizi
```

- **Vektör:** Dinamik boyutludur, çalışma zamanında **büyüyebilir veya küçülebilir**.

```
vector<int> vektor;
vektor.push_back(10);  // Eleman ekleme
```

2. **Bellek Yönetimi**

- **Dizi:** Statik bellekte veya yığın üzerinde saklanır. Dizi için ayrılan bellek miktarı, önceden belirlenir.
- **Vektör:** Dinamik olarak belleği yönetir. Gerekli olduğunda otomatik olarak **bellek genişletir veya küçültür**.

3. Kullanım Kolaylığı

- **Dizi:** Bellek yönetimi ve boyutlandırma **kullanıcı tarafından** yapılır.
- **Vektör:** Kullanımı daha kolaydır; eleman ekleme ve çıkarma işlemleri için hazır fonksiyonlar sağlar (ör. `push_back()`, `pop_back()`).

4. **Fonksiyonel Destek**

- **Dizi:** Temel işlemler sağlar. Kullanıcı elle işlem yapar.

```
int dizi[3] = {1, 2, 3};
for (int i = 0; i < 3; ++i) {
    cout << dizi[i] << " ";
}
```

- **Vektör:** STL (Standart Şablon Kütüphanesi) ile daha fazla fonksiyon desteği sağlar. Örneğin, sıralama:

```
vector<int> vektor = {3, 1, 2};
sort(vektor.begin(), vektor.end());  // Vektör elemanlarını sıralar
```

5. Boyut Kontrol Fonksiyonları

- **Dizi:** Boyut statik olarak bilinir ve `sizeof()` ile öğrenilebilir.

```
int dizi[5];
cout << "Dizi Boyutu: " << sizeof(dizi) / sizeof(dizi[0]) << endl;
```

- **Vektör:** Boyut, çalışma zamanında `size()` ile öğrenilir.

```
vector<int> vektor = {1, 2, 3};
cout << "Vektör Boyutu: " << vektor.size() << endl;
```

6. Eleman Ekleme ve Çıkarma

- **Dizi:** Sabit boyutlu olduğundan **ekleme/çıkarma işlemi doğrudan mümkün değildir**. Eleman eklemek için boyut artırma işlemi elle yapılır.

```
// Yeni eleman eklemek için manuel olarak yeni dizi oluşturulur
int dizi[3] = {1, 2, 3};
int yeniDizi[4];
for (int i = 0; i < 3; ++i) {
    yeniDizi[i] = dizi[i];
}
yeniDizi[3] = 4;  // Yeni eleman eklendi
```

- **Vektör:** `push_back()` ve `pop_back()` gibi fonksiyonlarla **dinamik eleman ekleme ve çıkarma** mümkündür.

```
vector<int> vektor = {1, 2, 3};
vektor.push_back(4);  // 4 eklenir
vektor.pop_back();    // Son eleman çıkarılır
```

7. **Performans**

- **Dizi:** Bellek yönetiminde daha **hızlıdır** çünkü statik boyutlu ve sabit bir yapıdır.
- **Vektör:** Daha esnek olmasına rağmen, dinamik bellek yönetimi nedeniyle dizilere göre **biraz daha yavaştır**.

8. **Bellek Kullanımı**

- **Dizi:** Sabit boyutlu olduğu için bellekte **gereksiz yer kaplayabilir**. Bellek verimli kullanılmaz.
- **Vektör:** Boyut dinamik olarak ayarlandığı için belleği daha **verimli** kullanır.

9. **Kapsam ve Ömür**

- **Dizi:** Lokal olarak tanımlandığında, kapsam dışına çıkıldığında belleği serbest bırakılır.
- **Vektör:** Vektör sınıfı destructor'ı çağrıldığında belleği otomatik olarak temizler.

10. **Özet Tablo**

Özellik	Dizi (Array)	Vektör (std::vector)
Boyut	Sabit	Dinamik
Bellek Yönetimi	Statik	Dinamik
Eleman Ekleme/Çıkarma	Mümkün değil (manuel)	`push_back()`, `pop_back()`
Performans	Daha hızlı	Biraz daha yavaş
Bellek Kullanımı	Sabit boyutlu	Verimli
STL Fonksiyonları	Desteklenmez	STL ile uyumlu

Sonuç

Diziler ve vektörler, veri saklama için önemli araçlardır. Diziler, sabit ve önceden bilinen boyutlar için idealdir. Ancak **dinamik veri yönetimi** ve esneklik gerektiğinde **vektörler** çok daha uygun bir seçimdir. Özellikle **mekatronik projelerde**, veri miktarının önceden tahmin edilemediği durumlarda **vektörler** büyük avantaj sağlar.

10. Dizi ve Karakter Dizileri ile Ödev Konusu

- **Ödev 1:** Kullanıcıdan n tane sayı alıp bir diziye kaydedin ve bu sayıları artan sırada sıralayan bir program yazın.
- **Ödev 2:** Kullanıcıdan bir kelime alın ve bu kelimenin palindrom olup olmadığını kontrol eden bir program yazın.