# ce artı artı

```cpp
#include <iostream>

using namespace std;
int main() {
  cout << "Hello World!";
  return 0;
}
```

# Data Types

## Integer:

Tam sayı - 2 or 4 bytes

```cpp
int x = 5;
```

## Float:

Kesirli sayı - 4 bytes

```cpp
float pi;
pi = 3,14;
```

## Double:

Daha büyük kesirli sayılar için - 8 bytes

```cpp
double pi_but_cooler = 3,14159265358979323846264338327950288841971;
```

## Char:

Tek karakter için - 1 byte

```cpp
char harf = 'c';
```

## Void:

Yok, hiç, boş... - 0 byte

## Boolean:

Doğru-yanlış koşullu ifadesi - 1 byte

```cpp
bool condition = false;
```

---

# Bükülmüş Data Types

## Short:

Küçük tam sayılar için (range -32768 to 32767)

## Long:

Büyük tam sayılar için

## Long long:

Baya büyük tam sayılar için (örneğin astronomik hesaplamalarda kullanılıyor)

## Signed int:

Int tipinin aynısı

## Unsigned int:

Sadece pozitif tam sayılar için

```
long b = 4523232;
long int c = 2345342;
long double d = 233434.56343;
short d = 3434233; // Error! out of range
unsigned int a = -5;    // Error! can only store positive numbers or 0
```

# Matemetiksel Zamazingolar

## Max:

```
cout << max(5, 36);
```

## Min:

```
cout << min(5, 36);
```

## Square Root (Karekök):

```
#include <cmath>
cout << sqrt(100);
```

## Pow (Üs alma):

```
int x = pow(2, 4);
```

# Conditional Statements (Koşullu İfadeler)

## If - else if - else

```
if (condition) {
// Condition eğer doğru ise buradaki kodlar çalışır
```

```
}
else if (condition){
// Eğer if içindeki ifade yanlış ise buraya bakar, buranın condition'ı doğru ise buradaki kodlar çalışır
}
else if (condition){
// Eğer yukarıdaki else if içindeki ifade yanlış ise buraya bakar, buranın condition'ı doğru ise buradaki
kodlar çalışır
}
else{
// İfade hiçbir koşula uymazsa buradaki kodlar çalışır
}
```

### Switch-case

```
int num = 2;
switch (num) {
    case 0:
        std::cout << "Zero";
        break;
    case 1:
        std::cout << "One";
        break;
    case 2:
        std::cout << "Two";
        break;
    case 3:
        std::cout << "Three";
        break;
    default:
        std::cout << "Other";
        break;
}
```

## Loops (Döngüler)

### While Loop:

```
while (condition)
{
// condition true olduğu sürece buradaki kodlar döngüye girer
}
```

### Do-While Loop:

```
do
{
// condition yanlış olsa bile bir kez buradaki kodlar çalışır
} while (condition);  // daha sonra burada condition kontrol edilir eğer true ise döngü devam eder,
false ise döngüden çıkar
```

#### Pro tips:

Do-while döngüsüne giren condition'ın false olduğunu varsayalım. Ancak false olsa bile başka bir özelliği varsa döngüde tutmak istiyoruz. Böyle bir durumda do-while döngüsünün içine if koşulu ile condition'da istediğimiz özellik var mı bakarız. Varsa condition'ı yine "do" içinde true yaparız ve "while" kısmına geldiğinde döngü döner. Eğer istediğimiz özellik yoksa zaten başta false olduğundan döngüye hiç girmez.

## For Loop:

```
for (int i = 0; i < count; i++) // (döngü parametresinin başlangıç değeri; koşul; her döndüğünde parametrenin ne olacağı)
{
// Kodlar
}
```

## Break:

Döngü dönerken break satırına geldiğinde döngüyü kırar, yani döngüden çıkar. Break anahtar kelimesi genelde koşullu ifadeler ile kullanılır.

```
int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            break;
        }
        cout << i;
    }
    // Çıktı: 0 1 2 3 4
    ```

### Continue:
Döngü dönerken continue satırına geldiğinde döngünün o dönüşünde çalışacak kodları atlar ve döngü bir sonraki döngü parametresi ile devam eder. Continue anahtar kelimesi genelde koşullu ifadeler ile kullanılır.

```cpp
int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            continue;
        }
        cout << i;
    }
    // Çıktı: 0 1 2 3 4 6 7 8 9      (5'i atladı)
    ```
## Operatörler

#### İlişkisel Operatörler

|  |  |
| --- | --- |
| `a == b` | a, b'ye eşit mi? |
| `a` != `b` | a, b'ye eşit değil mi? |
| `a < b` | a, b'den küçük mü? |
| `a > b` | a, b'den büyük mü? |
| `a` <= `b` | a, b'den küçük-eşit mi? |
| `a` >= `b` | a, b'den büyük-eşit mi? |

#### Atama Operatörleri

|  |  |
| --- | --- |
| `a += b` | a = a + b |
| `a -= b` | a = a - b |
| `a *= b` | a = a * b |
| `a /= b` | a = a / b |

```
| `a %= b` |  a = a % b |

#### Logical (Mantıksal) Operatörler

|             |                               |
| ----------- | ----------------------------- |
| `a && b`    | İkisi true ise _(AND)_        |
| `x` \|\| `y` | Herhangi biri true ise _(OR)_ |
| `!z`        | z false ise _(NOT)_           |

#### Bitsel (Bitwise) Operatörler
Sayıların her biti için ilgili bitler arasında işlem yapar

|           |                                                |
| --------- | ---------------------------------------------- |
| `a & b`   | Binary AND                                     |
| `a` \| `b` | Binary OR                                      |
| `a ^ b`   | Binary XOR (Birbirinden farklı ise 1 döndürür) |
| `~ a`     | 1' Komplementi                                 |
| `a << b`  | Binary Sola Kaydırma                           |
| `a >> b`  | Binary Sağa Kaydırma                           |
```cpp
int x = 7;  //00000111
int y = 12; //00001100
int z;

z = x & y;
```

## Fonksiyonlar

```cpp
#include <iostream>
int add(int a, int b) {
    return a + b;  // return anahtar kelimesi fonksiyonun ne döndüreceğini belirtir. Eğer void bir
fonksiyon ise değer dönmez
}

int main() {
    std::cout << add(10, 20);
}
```

### Overloading:

C dil ailesinde aynı isimli fakat farklı argümanlar alan fonksiyonlar tanımlamak mümkündür. Fonksiyonlardan birini çağıracağımız zaman verdiğimiz argüman tanımlı olan hangi fonksiyona uyuyor ise o çalışır.

```cpp
#include <stdlib.h>
#include <iostream>
using namespace std;
int topla(int a, int b) {
    return a + b;
}
int topla(int a, int b, int c) {
    return a + b + c;
}

int main() {
    int toplam = topla(3, 3, 3);
    cout << toplam; // çıktı 9 olacaktır
```
```

```
    return 0;
}
```

## Arrays (Diziler)

```
int myNum[3] = {10, 20, 30};

string games[4] = {"hotel E.G.O", "yuppie psycho", "omori", "the count lucanor"};

cout << games[0]; // çıktı hotel E.G.O
myNum[2] = 40;   // 2. index'i değiştirdik
```

```
int main() {
    int a[3] = { 1,2,3 };
    for (int i : a) { // a array'i boyunca döner
        cout << i << "\n";
    }

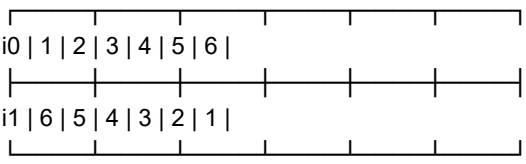
    return 0;
}
// veya
int main() {
    int a[3] = { 1,2,3 };

    for (int i = 0; i < sizeof(a)/4; i++) {   //4'e bölmemizin nedeni int 4 byte yer tutar sizeof(a) bu nedenle 12 değerini döndürür
        cout << a[i] << "\n";                // 4'e bölerek array içindeki element sayısına erişebiliriz
    }

    return 0;
}
```

## Multidimensional Arrays

```
j0  j1  j2   j3   j4   j5
```




```
int x[2][6] = {
    {1,2,3,4,5,6}, {6,5,4,3,2,1}
};
for (int i = 0; i < 2; ++i) {
    for (int j = 0; j < 6; ++j) {
        std::cout << x[i][j] << " ";
    }
}
// Outputs: 1 2 3 4 5 6 6 5 4 3 2 1
```

Minicik not: "i++" ile "++i" farklı şeylerdir. İlkinde önce i'yi döndürür sonra değerini arttırır, ikinci gösterimde ise önce i'yi arttırır sonra döndürür. Ancak bu iki kullanım for döngüsü içinde aynı sonucu veriyor. Bazı yerlerde nadiren kullanılabiliyor.

# Pointer'lar

Programlama dillerinde **işaretçi**, bellek adresi tutmak için tasarlanmış bir türdür.

İşaretçiyi anlamada, bir benzetim olarak kitap ayracı örneği verilebilir. Ayraç bir kitap sayfasına yerleştirilir, burada sayfa numarası bir bellek adresi, ayraç ise bir işaretçidir. Sayfa içeriğine erişmek için doğrudan ayracın bulunduğu sayfa açılabilir.

Bir değişken tanımlandığında, bellekte ona bir yer ayrılır. İşaretçiler bu bellek alanının başlangıç adresini tutar. a tam sayı değişkeni tanımlandığında, atandığı adresin başlangıcından değişkenin türünün boyu kadar bellekte yer kaplar. Değişkene atanan değer, ayrılan bu bellek alanında tutulur.

```
int a = 3;

int *ptr = &a;

cout << "address of variable : " << &a;
cout << "address of variable : " << ptr;

cout << "value of variable : " << a;
cout << "value of variable : " << *ptr;
//değeri değiştirme
*ptr = 9
```

## & Operatörü

& operatörü önüne geldiği değişkenin bellekteki adresini döndürür. Bu değişken int, string, array hatta pointer bile olabilir. *(x, a'yı tutan bir pointer olsun: &a a'nın adresini, &x ise a'yı tutan x pointer'ının adresini verir)*
`&a` **ifadesi:** Bu ifade, `a` değişkeninin bellekteki **adresini** döndürür. Yani, `&a` ifadesi `a` nın adresini işaret eden bir değerdir.

## * Operatörü

Eğer bir işaretçinin (örneğin `ptr` nin) değerini almak istersek `*ptr` şeklinde erişiriz , bu `ptr` işaretçisinin işaret ettiği **adresteki değere** erişmenizi sağlar. `*ptr` , işaret edilen adresteki gerçek değeri temsil eder.

## `int *ptr` tanımı:

Bu ifade, `ptr` adında bir işaretçi (pointer) değişkeni tanımlar. Bu işaretçi, bir `int` türündeki bir değerin **adresini** tutmak için kullanılır. Buradaki `*` işaretçi tanımlarken kullanılır ve bu `ptr` 'nin bir işaretçi olduğunu belirtir. Bu, `ptr` 'nin kendisinin bir adres olduğunu gösterir.

## * Operatörünün 2 farklı kullanımı:

- **Pointer Tanımlama:** `int *ptr;` ifadesiyle bir pointer tanımlıyoruz. Bu, `ptr` nin bir `int` türündeki değerin adresini tutacağını belirtir.
- **Dereferencing (Değer Erişimi):** `*ptr` ifadesiyle, `ptr` nin işaret ettiği adresteki değere erişiyorsunuz.

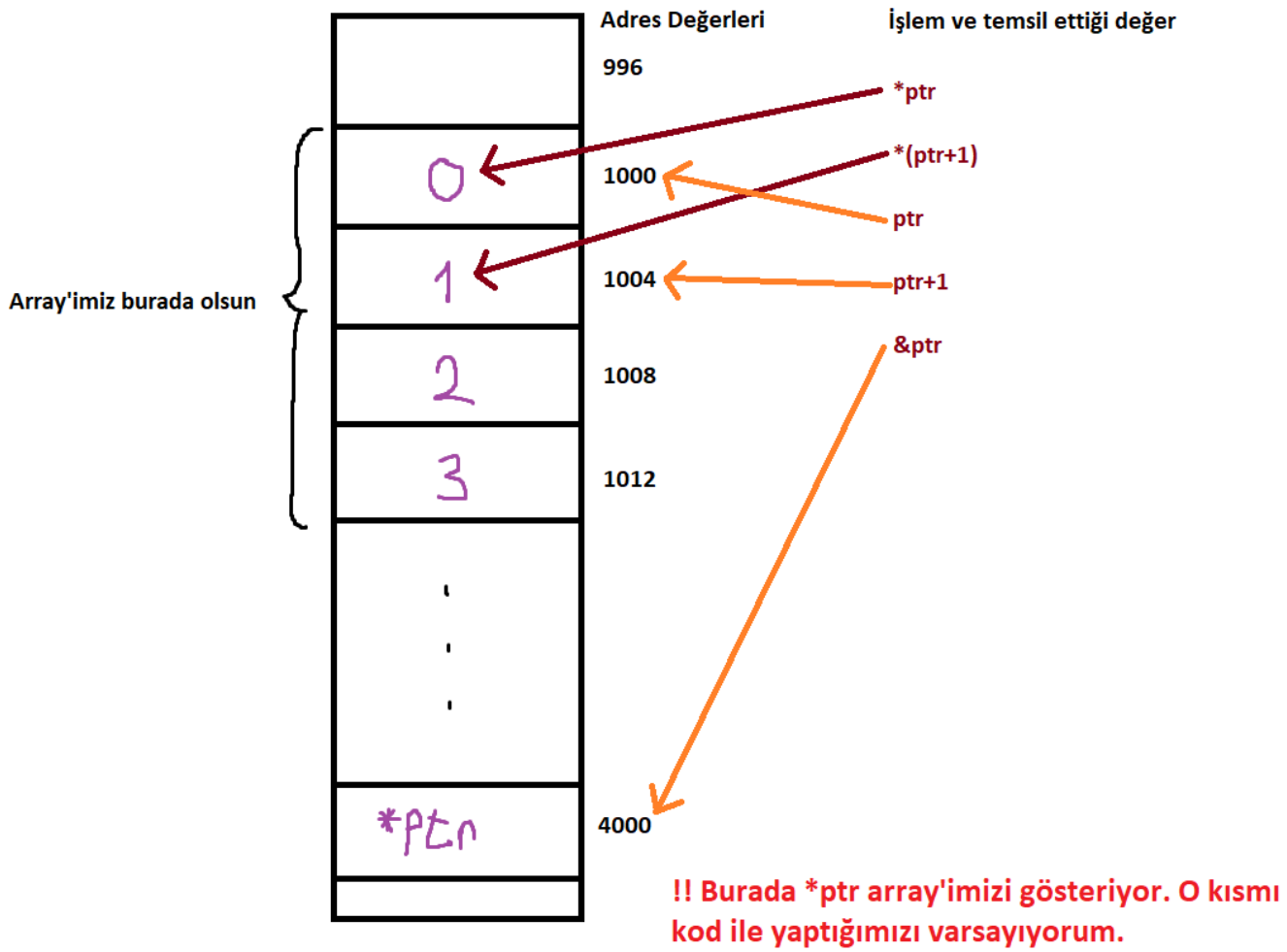
```
void topla(int *ptr) {
    *ptr += 3;
}
void main()
{   int a = 5;
    topla(&a);
    cout << a;
}
```

## Pointer ve array çorbası:

```
int numbers[] = { 0 , 1 , 2 , 3 };
int *ptr = numbers;
cout << *ptr<< endl;
cout << ptr << endl;
cout << ptr+1 << endl; //games array'inin başlangıç adresinden
cout << *(ptr+1) << endl;
```

urada ptr+1 belleğin içinde oluşturduğumuz arrayin bloğunu 1 arttırır. Örnekte numbers adında 4 elemanlı array tanımladık ve arrayin tipi int olduğundan her blok 4 byte. 4 değerimiz olduğundan bellekte 16 byte yer kaplıyor. Array'in başlangıç adresinin 1000 olduğunu varsayalım, bu durumda 1000. adreste "0" değeri tutuluyor, 1004. adreste "1" değeri tutuluyor vs... Örnekte bu arrayi ptr adında bir pointer ile tutuyoruz.

Şimdi bu bilgilerimiz ile pointer'larla oynayalım:
"*ptr" yi yazdırdığımızda array'in başlangıç adresindeki değeri, yani 0'ı verir. Array'in hepsini bu şekilde alamadığımıza dikkat edelim. *(onun için döngü ile tek tek geçmemiz lazım)*
"ptr" ve "ptr+1" i birlikte görelim. Bir pointer tanımladığımızda (bu örnek için *ptr) yıldızsız hali yani "ptr", neyi tutmasını istediysek onun başlangıç adresini döndürür. Örneğin arrayin başlangıç adresi 1000 ise "ptr" 1000 değerini döndürür. "ptr+1" ise arrayin bir sonraki elemanının adres değerini döndürür. Yukarıda açıkladığım üzere bu değer de 1004'tür.
Peki 1004 adresinde tutulan yani array'imizin 2. elemanı(1. index'i)'nı nasıl alabiliriz. Pointer'ımızın işaret ettiği adresteki değeri almak için "*" işareti kullanıyorduk. Öyleyse *(ptr+1) yaparsak array'in 2. elemanına ulaşmış oluruz. Aynı şeyler ptr+2,ptr+3 için de geçerli




## Structures (struct, yapılar):

Yapılar aynı array'ler gibi verileri depolamak için kullanılan bir türdür. Ancak array'lerin aksine oluşturduğumuz yapı içerisine farklı tip verileri saklayabiliriz (int, float, string)

Yapı oluşturmak için "struct" anahtar kelimesini kullanırız. En sona da yapı değişkeninin adını yazıyoruz. Yapı içerisindeki değişkenlerimize erişmek için ise: "yapıAdı.değişkenAdı". Yapıdaki değişkenlere değer atamak için: "yapıAdı.değişkenAdı = değer".

```
struct {
    int numbers;
    string names;
} myStruct;

myStruct.names = "Lavuk";

cout << myStruct.names;
```

## Birden fazla değişken ile aynı yapıyı kullanma:

Bazen bir yapının içerisindeki özellikleri farklı değişkenler için kullanmak isteriz. Bunun için yapının sonuna daha fazla değişken ismi ekleyebiliriz.

```
struct {
    string dersAdi;
    int dersKredisi;
} ders1, ders2;

ders1.dersAdi = "Haberleşme";
ders1.dersKredisi = 5;

ders2.dersAdi = "Devre Analizi";
ders2.dersKredisi = 4;

cout << ders2.dersKredisi;
```

Yukarıda yaptığımızın aynısını farklı bir yol ile yapmamız mümkün.

## Yapıyı tür olarak tanımlama:

"Struct" anahtar kelimemizin yanına yapımızın adını girerek o yapıyı bir tür olarak tanımlayabiliriz. Bu sayede kodun herhangi bir yerinde bu yapıdan farklı değişkenler üretebiliriz.

```
struct Ders {
    string dersAdi;
    int dersKredisi;
};

Ders ders1; // Ders yapımızdan ders1 adlı değişken oluşturduk
ders1.dersAdi = "Haberleşme";
ders1.dersKredisi = 5;

Ders ders2; // Yine aynı yapıyı kullanarak farklı bir değişken oluşturduk
ders2.dersAdi = "Devre Analizi";
ders2.dersKredisi = 4;

cout << ders2.dersKredisi;
```

Yazarın Notu(benim fikrim olduğundan sakat bi fikir olabilir evde denemeyiniz!) : En son gösterdiğim yapıyı tür olarak tanımlama daha kullanışlı çünkü kod yazarken bir yapı oluşturduğumuzda kaç değişkenin o yapıyı kullanacağını genelde önceden bilmeyiz. Bu sebeple istediğimiz her yerden(başka bir script içerisinden de ulaşabiliriz "include" ile) ihtiyacımız olduğunda değişken üretmek daha pratik ve düzenli bir kullanım gibi.