C, C++ PROGRAMLAMA DİLİ

TEMEL KAVRAMLAR

C ve C++ Programlama Dillerinin Özellikleri

- C programlama dili, 1970'li yıllarda Dennis Ritchie tarafından UNIX işletim sistemi için geliştirilmiş olup, düşük seviyeli donanım işlemlerini yapabilme gücü ile yüksek seviyeli programlama kolaylığını birleştiren yapısıyla programlama dünyasında bir dönüm noktası olmuştur.

- C dili, sistem programlama, işletim sistemi geliştirme, gömülü sistemler, donanım sürücüleri ve performans odaklı uygulamalar için tercih edilmektedir çünkü donanım kaynaklarına doğrudan erişim imkânı verir.

- C++ dili, C'nin üzerine nesne yönelimli programlama (object-oriented programming – OOP) özelliklerinin eklenmesiyle 1980'li yıllarda Bjarne Stroustrup tarafından geliştirilmiştir ve bu sayede hem yapısal programlama hem de nesne yönelimli programlama paradigmasını destekleyen hibrit bir dil hâline gelmiştir.

- C++ dilinde sınıflar (class), kalıtım (inheritance), çok biçimlilik (polymorphism), enkapsülasyon (encapsulation) ve soyutlama (abstraction) gibi nesne yönelimli kavramlar yer almakta; bu da dilin büyük ve karmaşık yazılımların geliştirilmesine uygun hâle gelmesini sağlamaktadır.

- C'nin temel avantajı, hız ve verimliliktir; çünkü derlenen kod makine koduna çok yakın bir düzeyde çalışır. C++'ın avantajı ise kodun modüler, yeniden kullanılabilir ve sürdürülebilir olmasını sağlayan nesne yönelimli yapıları içermesidir.

- Her iki dil de taşınabilirlik (portability) özelliği sayesinde farklı işletim sistemlerinde çalışabilmekte, standart kütüphaneleriyle geniş bir uygulama yelpazesine hitap etmektedir.

- Günümüzde C dili, özellikle gömülü sistemlerde ve donanım yazılımlarında tercih edilirken, C++ dili oyun motorları, büyük ölçekli masaüstü yazılımlar, finansal sistemler ve performans kritik uygulamalarda kullanılmaktadır.

Programın Başlangıç Noktası: main() Fonksiyonu

- C ve C++ dillerinde yazılmış her programın çalışmaya başladığı nokta main() fonksiyonudur.

- Derleyici, kaynak kodu derledikten sonra işletim sistemi programı çalıştırdığında, kontrol ilk olarak main() fonksiyonuna aktarılır ve bu fonksiyonun içindeki komutlar sırasıyla yürütülür.

- main() fonksiyonunun dönüş tipi genellikle int olarak tanımlanır, çünkü programın sonunda işletim sistemine bir çıkış kodu döndürülür. Bu çıkış kodu, programın başarıyla çalışıp çalışmadığını gösterir.

- 0 değeri genellikle programın başarılı şekilde sonlandığını ifade eder.

- 0 dışındaki değerler ise hata veya farklı durum kodlarını göstermek için kullanılabilir.

- main() fonksiyonu, parametresiz tanımlanabileceği gibi, komut satırı argümanları almak üzere şu şekilde de tanımlanabilir:

```
int main(int argc, char *argv[])
```

argc (argument count): Komut satırından programa geçirilen argümanların sayısını belirtir.

argv (argument vector): Komut satırından girilen argümanları tutan karakter dizisi (string) dizisidir.

- Bu yapı sayesinde, programlar kullanıcıdan komut satırı aracılığıyla dinamik giriş alabilir ve daha esnek bir şekilde çalışabilir.

- main() fonksiyonunun programın kalbi olduğu unutulmamalıdır; tüm algoritmalar, kontrol yapıları ve fonksiyon çağrıları bu fonksiyon üzerinden başlatılır.

Derleyici, Yorumlayıcı, Ön İşlemci (Preprocessor) Kavramı

- Derleyici (compiler), C veya C++ dillerinde yazılmış kaynak kodu (source code) makine diline (machine code) çeviren yazılımdır. Derleme süreci sırasında sözdizimi (syntax) hataları kontrol edilir, kod optimize edilir ve çalıştırılabilir dosya oluşturulur.

- C ve C++ dilleri derlenen (compiled) diller sınıfındadır; bu nedenle yazılan kod doğrudan makine koduna çevrilerek çalıştırılır ve bu da yüksek hız sağlar.

- Yorumlayıcı (interpreter), derleyiciden farklı olarak kodu satır satır çalıştıran bir yazılımdır. C ve C++ dilleri doğrudan yorumlanmaz, ancak bazı özel durumlarda hata ayıklama veya eğitim amaçlı yorumlayıcı araçlar kullanılabilir.

- Ön işlemci (preprocessor), derleyiciden önce çalışan ve kaynak kod üzerindeki bazı yönergeleri işleyen bir aşamadır.

- Ön işlemci, # ile başlayan yönergeleri işler.
- Örneğin #include komutu, başka dosyaların programın içine eklenmesini sağlar; #define makro tanımlamalarını yapar.
- Derleme süreci şu aşamalardan oluşur:

Ön işleme (Preprocessing): Makrolar açılır, dosya dahil etme işlemleri yapılır.

Derleme (Compilation): Kaynak kod, ara dil veya makine koduna dönüştürülür.

Bağlama (Linking): Kütüphaneler ve fonksiyonlar birleştirilir, çalıştırılabilir dosya elde edilir.

Makro, Inline Fonksiyon, On İşlemci Blokları

- Makrolar, #define komutu ile tanımlanan ve derleme öncesinde yerlerine geçen sabitler veya kod parçalarıdır.

Örnek:

```
#define PI 3.14

#define SQUARE(x) (x * x)
```

- Makrolar, tekrar eden sabit değerlerin veya basit işlemlerin kod içinde kolayca kullanılmasını sağlar.

- Ancak fazla kullanıldığında kod karmaşasına ve hata ayıklama zorluklarına yol açabilir.
- Inline fonksiyonlar, inline anahtar kelimesi ile tanımlanan ve çağrıldıkları yerde fonksiyon çağrısı yerine doğrudan fonksiyon gövdesinin yerleştirilmesini sağlayan fonksiyonlardır.
- Bu yöntem, özellikle küçük ve sık kullanılan fonksiyonlarda fonksiyon çağrı maliyetini ortadan kaldırarak performansı artırır.

Örnek:

```
inline int kare(int x) {
    return x * x;
}
```

- Ön işlemci blokları, koşullu derleme yapıları oluşturmak için kullanılır.
- #ifdef, #ifndef, #endif gibi yapılar sayesinde belirli koşullar sağlandığında kodun derlenip derlenmeyeceği kontrol edilir.
- Bu özellik, büyük projelerde farklı platformlara uygun kod yazmayı kolaylaştırır.

Örnek:

```
#ifdef WINDOWS
   printf("Windows ortamında çalışıyor.\n");
#else
   printf("Windows dışındaki bir ortamda çalışıyor.\n");
#endif
```

Ek Önemli Alt Başlıklar

- Derleyici Hataları ve Çalışma Zamanı Hataları
- Derleyici hataları, kod yazılırken sözdizimi veya tür uyuşmazlığı gibi nedenlerden dolayı oluşur ve programın derlenmesini engeller.
- Çalışma zamanı hataları ise program çalıştırılırken ortaya çıkar; örneğin, sıfıra bölme veya dizinin sınırları dışına çıkma gibi durumlar bu tür hatalardır.
- Kütüphaneler ve Header Dosyaları
- C dilinde stdio.h, stdlib.h, math.h gibi kütüphaneler; C++ dilinde ise <iostream>, <string>, <vector> gibi header dosyaları sıkça kullanılır.
- Bu kütüphaneler, hazır fonksiyonlar ve veri yapıları sağlayarak programcının işini kolaylaştırır.

- Yorum Satırları
- Kodun daha anlaşılır olabilmesi için açıklama eklemek amacıyla kullanılır.
- C'de ve C++'ta // tek satırlık, /* ... */ ise çok satırlık yorumlar için kullanılır.

VERİ TİPLERİ VE DEĞİŞKENLER

Temel Veri Tipleri: int, float, double, char, bool

int (integer): Tamsayıları temsil eden veri tipidir. İşaretli (signed) olarak kullanıldığında pozitif ve negatif tam sayıları, işaretsiz (unsigned) olarak kullanıldığında yalnızca pozitif tam sayıları ifade eder. Bellekte kapladığı alan genellikle 4 bayttır, fakat derleyiciye ve sistem mimarisine bağlı olarak 2 bayt veya 8 bayt da olabilir. Örneğin, int x = 25; ifadesi bellekte tam sayı saklamayı sağlar.

float (floating point): Ondalıklı sayıları temsil eden veri tipidir ve genellikle 4 bayt bellek alanı kaplar. Yaklaşık 6–7 basamak duyarlılıkla ondalıklı sayıları ifade eder. Örneğin, float pi = 3.14; ifadesi ondalıklı bir değeri belleğe atar.

double (double precision floating point): Daha yüksek hassasiyetli ondalıklı sayıları ifade eder. Genellikle 8 bayt bellek alanı kaplar ve yaklaşık 15–16 basamak duyarlılığa sahiptir. double alan = 1234.56789; ifadesi çok daha kesin hesaplamaların yapılmasına olanak tanır. Bilimsel hesaplamalarda ve mühendislik uygulamalarında sık tercih edilir.

char (character): Tek bir karakteri veya ASCII tablosundaki bir işareti temsil eder. Bellekte genellikle 1 bayt yer kaplar. char harf = ‘A’; ifadesinde tek tırnak içinde belirtilen karakter bellekte ASCII değeri üzerinden saklanır.

bool (boolean): Mantıksal doğru (true) veya yanlış (false) değerini tutmak için kullanılan veri tipidir. C dilinde bool veri tipi standart olarak bulunmaz, stdbool.h kütüphanesi eklenerek kullanılabilir. C++ dilinde ise bool yerleşik bir veri tipidir ve true değeri bellekte 1, false değeri ise 0 olarak saklanır.

Signed ve Unsigned Tipler

- Signed tipler, hem pozitif hem de negatif değerleri ifade edebilir. Örneğin, signed int tipinde –2,147,483,648 ile +2,147,483,647 arasında değer saklanabilir.

- Unsigned tipler, yalnızca sıfır ve pozitif değerleri temsil eder. Örneğin, unsigned int tipinde 0 ile 4,294,967,295 arasında değer saklanabilir.

- Unsigned tipler, özellikle negatif değerlerin kullanılmadığı durumlarda tercih edilir ve böylece aynı bellek alanında daha geniş bir pozitif aralık elde edilir.

- C ve C++ dillerinde int, char, short, long gibi veri tipleri signed veya unsigned olarak tanımlanabilir.

Örnek:

```
unsigned int yas = 25;

signed char karakter = -120;
```

- Signed–unsigned dönüşümleri dikkatli kullanılmalıdır çünkü negatif bir değerin unsigned değişkene atanması beklenmedik büyük değerler oluşturabilir.

8 Bit, 16 Bit, 32 Bit, 64 Bit Değer Aralıkları

- Veri tiplerinin alabileceği değer aralıkları, bellekte kapladıkları bit sayısına göre değişir. Genel olarak:

8 bit signed: –128 ile +127 arası

8 bit unsigned: 0 ile 255 arası

16 bit signed: –32,768 ile +32,767 arası

16 bit unsigned: 0 ile 65,535 arası

32 bit signed: –2,147,483,648 ile +2,147,483,647 arası

32 bit unsigned: 0 ile 4,294,967,295 arası

64 bit signed: –9,223,372,036,854,775,808 ile +9,223,372,036,854,775,807 arası

64 bit unsigned: 0 ile 18,446,744,073,709,551,615 arası

- Bu aralıklar, kullanılan işletim sistemi ve derleyiciye bağlı olarak değişebilir; bu yüzden standart kütüphanelerde stdint.h (C) ve <cstdint> (C++) dosyaları kullanılarak int8_t, uint16_t, int32_t, uint64_t gibi kesin bit boyutlu veri tipleri tercih edilmelidir.

- Böylece farklı platformlarda taşınabilir (portable) yazılım geliştirmek mümkün olur.

Değişken Tanımlama Kuralları (Geçerli/Geçersiz İsimler)

- C ve C++ dillerinde değişken isimleri belirlenirken bazı kurallara dikkat edilmelidir:
- İsimler bir harf veya alt çizgi (_) karakteri ile başlamalıdır, rakam ile başlayamaz.
- Değişken isimlerinde harf, rakam ve alt çizgi kullanılabilir, ancak boşluk veya özel karakterler (@, #, ! gibi) kullanılamaz.
- Büyük ve küçük harfler farklı değişken isimleri olarak kabul edilir. Örneğin, sayi ile Sayi iki farklı değişkendir.
- Değişken isimleri anlamlı seçilmeli ve kodun okunabilirliğini artırmalıdır. Örneğin, ogrenciSayisi ifadesi x ifadesine göre daha anlaşılırdır.
- Anahtar kelimeler (int, float, for, if gibi) değişken ismi olarak kullanılamaz.

Geçerli örnekler:

```
int ogrenciSayisi;

float ortalama_not;

char ad1;
```

Geçersiz örnekler:

```
int 1sayi;      // Rakamla başlayamaz

float double;   // double anahtar kelimedir

char isim-soy;  // Özel karakter içeremez
```

Otomatik Tür Belirleme: auto

- C++11 standardı ile birlikte gelen auto anahtar kelimesi, bir değişkenin türünü derleyicinin otomatik olarak belirlemesine olanak tanır.
- Bu özellik, özellikle karmaşık tiplerin kullanıldığı durumlarda kodu sadeleştirir ve daha okunabilir hâle getirir.

Örnek:

```
auto x = 10;       // int olarak algılanır

auto y = 3.14;     // double olarak algılanır

auto z = "Merhaba"; // const char* olarak algılanır
```

- auto, kodun bakımını kolaylaştırır ve tür değişikliklerinde **daha az** hata yapılmasını sağlar.

- Ancak gereksiz kullanıldığında kodun anlaşılabilirliğini azaltabilir; bu nedenle türün açıkça belirtilmesi gereken durumlarda kullanılmamalıdır.

Ek Önemli Alt Başlıklar

Tür Dönüşümleri (Type Casting)

- C ve C++ dillerinde bir veri tipinden diğerine dönüşüm yapılabilir.
- int tipinden double tipine dönüşüm genellikle veri kaybına yol açmaz, ancak double tipinden int tipine dönüşüm ondalık kısmın kaybolmasına neden olur.
- Tür dönüşümleri static_cast, dynamic_cast, const_cast ve reinterpret_cast gibi operatörlerle güvenli bir şekilde yapılabilir (C++).

Sabitler (Constants)

- Sabitler const anahtar kelimesi veya #define makrosu ile tanımlanabilir.
- Sabitler program boyunca değeri değişmeyen değişkenlerdir.
- Örneğin: const double PI = 3.14159;

Enumerations (enum)

- enum anahtar kelimesi, sabitleri gruplayarak daha anlamlı hâle getirir.

Örnek:

```
enum Renk {KIRMIZI, YESIL, MAVI};

Renk r = MAVI;
```

volatile anahtar kelimesi

- Donanım kontrollü değerlerin saklandığı değişkenlerde kullanılan volatile, derleyiciye bu değişkenin değeri dış kaynaklarla değiştirilebileceğini belirtir.

register değişkenleri

- Çok sık kullanılan değişkenlerin işlemci register'larında saklanması için öneri sunar. Modern derleyicilerde otomatik optimizasyon yapıldığı için artık nadiren kullanılır.

OPERATÖRLER

- C ve C++ dillerinde operatörler, değişkenler ve sabitler üzerinde işlem yapmak için kullanılan özel sembollerdir. Programlamada matematiksel, mantıksal ve yapısal işlemlerin en temel araçlarıdır. Aşağıda her operatör grubu detaylı biçimde ele alınmıştır.

Atama Operatörleri (=, +=, -=, *=, /=)

- Atama operatörleri, sağ tarafta bulunan değeri veya ifadenin sonucunu sol tarafta belirtilen değişkene atamak için kullanılır.

= (eşittir): Basit atama operatörüdür. Sağdaki değeri soldaki değişkene aktarır.

```
int x;

x = 10;  // x değişkenine 10 atanır.
```

+=: Mevcut değerin üzerine yeni değer ekleyerek sonucu değişkene atar.

```
x += 5;  // x = x + 5 anlamına gelir.
```

-=: Mevcut değerden yeni değeri çıkararak sonucu değişkene atar.

```
x -= 3;  // x = x - 3
```

*=: Mevcut değeri yeni değerle çarpar ve sonucu değişkene atar.

```
x *= 2;  // x = x * 2
```

/=: Mevcut değeri yeni değere böler ve sonucu değişkene atar.

```
x /= 4;  // x = x / 4
```

- Bu operatörler kodu daha kısa, okunabilir ve bakımı kolay hâle getirir.

Aritmetik Operatörler (+, -, *, /, %)

- Aritmetik operatörler, matematiksel işlemleri gerçekleştirmek için kullanılır.

+ (toplama): İki sayının toplamını döndürür.

```
int a = 5, b = 3;

int c = a + b;  // c = 8
```

- (çıkarma): Bir sayıdan diğerini çıkarır.

```
int c = a - b;  // c = 2
```

* (çarpma): İki sayıyı çarpar.

```
int c = a * b;  // c = 15
```

/ (bölme): Bir sayıyı diğerine böler. Tamsayı bölmesinde sonuç küsurat kısmı atılır, ondalıklı hesaplama için float veya double kullanılmalıdır.

```
int c = a / b;    // c = 1 (tamsayı bölme)

float d = 5.0/3;  // d = 1.666...
```

% (mod): İki sayının bölümünden kalanı verir. Yalnızca tamsayılar için geçerlidir.

```
int c = a % b;  // c = 2 (5 mod 3 = 2)
```

Karşılaştırma Operatörleri (==, !=, <, >, <=, >=)

- Karşılaştırma operatörleri, iki değeri kıyaslamak için kullanılır ve sonuç olarak true (doğru) veya false (yanlış) döndürür.

==: İki değer eşitse doğru döner.

```
if (a == b) { /* eşitlik durumu */ }
```

!=: İki değer eşit değilse doğru döner.

```
if (a != b) { /* eşit değil */ }
```

<: Soldaki değer sağdakinden küçükse doğru döner.

>: Soldaki değer sağdakinden büyükse doğru döner.

<=: Soldaki değer sağdakine küçük veya eşitse doğru döner.

>=: Soldaki değer sağdakine büyük veya eşitse doğru döner.

- Bu operatörler, özellikle koşul ifadelerinde (if, while, for) sıklıkla kullanılır.

Mantıksal Operatörler (&&, ||, !)

- Mantıksal operatörler, koşullar arasında bağ kurmak için kullanılır. Sonuçları boolean (true/false) değeridir.

&& (ve): İki koşul da doğruysa sonucu doğru döner.

```
if (a > 0 && b > 0) { /* her ikisi de pozitif */ }
```

|| (veya): Koşullardan **en az** biri doğruysa sonucu doğru döner.

```
if (a > 0 || b > 0) { /* en az biri pozitif */ }
```

! (değil): Bir koşulun tersini alır. Koşul doğruysa yanlış, yanlışsa doğru döner.

```
if (!(a > b)) { /* a b'den büyük değilse */ }
```

- Mantıksal operatörler genellikle karar yapılarında ve döngü koşullarında kullanılmaktadır.

Artırma ve Azaltma Operatörleri (++, --)

- Bu operatörler, değişkenin değerini bir artırmak veya bir azaltmak için kullanılır.

++ artırma operatörü: Değeri 1 artırır.

-- azaltma operatörü: Değeri 1 azaltır.

Bu operatörler iki farklı kullanım biçimine sahiptir:

Ön ek (prefix): ++x → Değişken önce artırılır, sonra işlemde kullanılır.

Son ek (postfix): x++ → Değişken önce işlemde kullanılır, sonra artırılır.

Örnek:

```
int x = 5;

int y = ++x;  // x = 6, y = 6

int z = x--;  // z = 6, x = 5
```

Üye Erişim Operatörleri (., ->, ::)

- C++ dilinde nesne yönelimli programlama kapsamında kullanılan üye erişim operatörleri, sınıf veya yapıların üyelerine erişmek için kullanılır.

. (nokta operatörü): Bir nesne veya yapı üzerinden doğrudan üye erişimi sağlar.

```
struct Ogrenci { int yas; };

Ogrenci o;

o .yas = 20;  // nokta operatörü ile erişim
```

-> (ok operatörü): Işaretçiler (pointer) üzerinden üye erişimini sağlar.

```
Ogrenci *p = &o;

p->yas = 21;  // ok operatörü ile erişim
```

:: (scope resolution operatörü): Global değişkenlere erişmek, sınıf üyelerine ulaşmak veya static değişkenleri çağırmak için kullanılır.

```
int x = 10;

namespace N { int x = 20; }

int main() {

   cout << ::x;   // global x'e erişim

   cout << N::x;  // namespace içindeki x'e erişim

}
```

- Bu operatörler, özellikle C++'ta nesne tabanlı yazılımların temelini oluşturur.

Ek Önemli Alt Başlıklar

Bit Düzeyinde Operatörler (Bitwise Operators)

- & (and), | (or), ^ (xor), ~ (not), << (sola kaydırma), >> (sağa kaydırma) gibi operatörler, sayıları bit düzeyinde manipüle etmek için kullanılır. Özellikle gömülü sistemlerde ve düşük seviyeli programlamada önemlidir.

Üçlü Koşul Operatörü (Ternary Operator ?:)

- Kısa if-else yapıları için kullanılır.

```
int a = 5, b = 10;

int buyuk = (a > b) ? a : b;  // buyuk = 10
```

KARAR YAPILARI VE DÖNGÜLER

- Programlama dillerinde karar yapıları ve döngüler, algoritmaların akışını kontrol eden temel yapılardır. C ve C++'ta bu yapılar, koşulların değerlendirilmesi ve işlemlerin belirli kurallara göre tekrar edilmesi için kullanılır.

if, else if, else

- **if yapısı**, bir koşulun doğru olup olmadığını kontrol eder ve koşul doğruysa içerisindeki kod bloğunu çalıştırır.

```
int yas = 18;
if (yas >= 18) {
.... cout<<"Reşitsiniz.";
}
```

- **else if yapısı**, ilk if koşulu sağlanmadığında başka koşulların kontrol edilmesini sağlar. Bu sayede çok yönlü karar mekanizmaları kurulabilir.

```
if (yas < 12) {
    cout << "Çocuk kategorisi.";
}
else if (yas < 18) {
    cout << "Genç kategorisi.";
}
else {
    cout << "Yetişkin kategorisi.";
}
```

- **else yapısı**, yukarıdaki hiçbir koşul sağlanmazsa devreye girer. Böylece koşullar dışında kalan tüm olasılıklar yakalanmış olur.
- if–else yapısı, programlamada en temel **karar kontrol mekanizmasıdır** ve kullanıcıdan alınan verilerin işlenmesinde, algoritmalarda farklı dallara ayrılmada kullanılır.

switch-case Yapısı

- **switch-case yapısı**, bir değişkenin değerine bağlı olarak farklı işlemler gerçekleştirmek için kullanılır. Özellikle çok sayıda olasılık varsa, kodun okunabilirliğini artırır.
- Kullanımı:

```
int gun = 3;

switch (gun) {

   case 1: cout << "Pazartesi"; break;

   case 2: cout << "Salı"; break;

   case 3: cout << "Çarşamba"; break;

   case 4: cout << "Perşembe"; break;

   case 5: cout << "Cuma"; break;

   default: cout << "Hafta sonu"; break;

}
```

- **case etiketleri**, kontrol edilen değişkenin değerine göre çalıştırılacak kodları belirtir.
- **break deyimi**, seçilen case bloğunun çalıştırıldıktan sonra switch yapısından çıkmasını sağlar. Eğer break kullanılmazsa program diğer case bloklarını da çalıştırmaya devam eder (**fall-through** davranışı).
- **default etiketi**, hiçbir case koşulu sağlanmadığında çalıştırılır ve genellikle hata yakalama veya alternatif işlem yapmak için kullanılır.

Döngüler: for, while, do-while

- Döngüler, belirli koşullar sağlandığı sürece kod bloklarının tekrar tekrar çalıştırılmasını sağlar.

for Döngüsü

- **for döngüsü**, özellikle belirli sayıda tekrar yapılacak işlemler için uygundur.

Yapısı:

```
for (başlangıç; koşul; artış) {
   // tekrar edilecek işlemler
}
```

Örnek:

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

while Döngüsü

- **while döngüsü**, koşul doğru olduğu sürece çalışır. Koşul baştan kontrol edildiği için, koşul ilk seferde sağlanmazsa döngü hiç çalışmaz.

Örnek:

```
int i = 0;
while (i < 5) {
    cout << i << " ";
    i++;
}
```

do-while Döngüsü

- **do-while döngüsü**, koşul sona bırakıldığı için döngü gövdesi **en az bir kez** çalıştırılır. Bu özelliğiyle while döngüsünden ayrılır.

Örnek:

```
int i = 0;

do {

   cout << i << " ";

   i++;

} while (i < 5);
```

Döngü Kontrol Deyimleri: break, continue

- **break deyimi**, döngü içerisinden çıkmak için kullanılır. Belirli bir koşul sağlandığında döngünün kalan kısmı çalıştırılmadan sonlandırılır.

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

- **continue deyimi**, döngünün mevcut yinelemesini sonlandırarak bir sonraki yinelemeye geçmesini sağlar. Döngü tamamen sonlanmaz, sadece o adım atlanır.

```
for (int i = 0; i < 10; i++) {
   if (i % 2 == 0) continue;
   cout << i << " ";
}
// Çıktı: 1 3 5 7 9
```

Ek Önemli Alt Başlıklar

İç içe döngüler (nested loops)

- Bir döngünün içinde başka bir döngü kullanılabilir. Bu yapı genellikle çok boyutlu diziler üzerinde işlem yapılırken tercih edilir.

```
for (int i = 0; i < 3; i++) {
   for (int j = 0; j < 3; j++) {
      cout << "(" << i << "," << j << ") ";
   }
}
```

Sonsuz Döngüler

- Eğer döngü koşulu hiçbir zaman sağlanmazsa veya koşul kalıcı olarak doğru kalırsa sonsuz döngü oluşur.

Örneğin: while (true) { /* sürekli çalışır */ }

- Sonsuz döngüler, genellikle sistem yazılımlarında, kullanıcı giriş bekleyen programlarda veya sürekli çalışan servislerde kullanılır.

Döngülerin performansa etkisi

- Döngü yapıları, özellikle büyük veri kümeleri üzerinde çalışırken programın performansını doğrudan etkiler. Bu nedenle döngülerin verimli yazılması ve gereksiz tekrarların önlenmesi önemlidir.

FONKSİYONLAR

- Fonksiyonlar, programlama dillerinde belirli bir görevi yerine getiren bağımsız kod bloklarıdır. Daha modüler, okunabilir, yeniden kullanılabilir ve bakımı kolay programlar geliştirmek için fonksiyonlar büyük önem taşır. C ve C++ dillerinde fonksiyonlar, programların temel yapı taşlarından biridir.

Fonksiyon Tanımlama ve Çağırma

- **Fonksiyon tanımlama**, bir fonksiyonun ismini, dönüş tipini, parametrelerini ve içindeki komutları belirlemek anlamına gelir. Temel yazım şekli şöyledir:

```
dönüş_tipi fonksiyon_adi(parametreler) {
   // fonksiyon gövdesi
   return değer;
}
```

- Fonksiyonların **dönüş tipi** olabilir (`int`, `double`, `char`, `bool` gibi) veya olmayabilir (`void`).

Örnek:

```
int topla(int a, int b) {
   return a + b;
}
```

- Fonksiyonun kullanılabilmesi için **çağrılması (function call)** gerekir. Fonksiyon çağrısı, fonksiyon ismi ve parametrelerin uygun biçimde yazılmasıyla yapılır.

```
int sonuc = topla(5, 7); // sonuc = 12
```

- Fonksiyonlar sayesinde kod tekrarından kaçınılır, aynı işlev farklı yerlerde tekrar tekrar yazılmak yerine tek **bir defa** tanımlanıp çağrılır.

Fonksiyon Prototipleri

- C ve C++'ta, derleyicinin fonksiyonları tanımadan önce kullanabilmesi için **fonksiyon prototipleri** kullanılır.
- Prototip, fonksiyonun dönüş tipini, adını ve parametre türlerini belirtir, ancak gövdesini içermez.

Örneğin:

```
int carp(int, int); // prototip
```

- Bu prototip, fonksiyonun daha sonra nasıl tanımlanacağına dair derleyiciye bilgi verir. Fonksiyon gövdesi genellikle `main()` fonksiyonundan sonra yazıldığında prototip gereklidir.
- Eğer fonksiyon gövdesi `main()` fonksiyonundan önce tanımlanmışsa prototip kullanmaya gerek yoktur.
- Fonksiyon prototipleri, özellikle **büyük projelerde kodun düzenlenmesini kolaylaştırır** ve **derleme sırasında hataların önlenmesine katkı sağlar**.

Parametreler: Değer ile (Pass by Value), Adres ile (Pass by Reference)

- Fonksiyonlara dışarıdan veri aktarılırken parametreler kullanılır. C ve C++ dillerinde bu aktarım iki temel yöntemle yapılır:

Değer ile Parametre Geçirme (Pass by Value)

- Fonksiyona parametre gönderildiğinde, parametrenin **kopyası** oluşturulur ve fonksiyon bu kopya üzerinde işlem yapar.
- Orijinal değişken fonksiyon dışında aynı kalır.

Örnek:

```
void degistir(int x) {
    x = x + 5;
}
int main() {
    int a = 10;
    degistir(a);
    cout << a;  // çıktı: 10
}
```

- Bu yöntemin avantajı, orijinal verinin değişmemesidir. Dezavantajı ise büyük veri yapılarında (ör. büyük diziler) kopyalama maliyetinin yüksek olmasıdır.

Adres ile Parametre Geçirme (Pass by Reference)

- Parametre fonksiyona **referans (adres)** yoluyla gönderilir. Fonksiyon, orijinal değişken üzerinde işlem yapar.
- C dilinde bu yöntem işaretçiler (pointers) ile, C++ dilinde ise referans (&) ile sağlanır.
- Örnek (C++ referans ile):

```
void degistir(int &x) {
    x = x + 5;
}
int main() {
    int a = 10;
    degistir(a);
    cout << a;  // çıktı: 15
}
```

- Bu yöntem, **verimlilik** sağlar çünkü kopyalama yapılmaz. Ancak orijinal veri değiştiği için dikkatli kullanılmalıdır.

Recursive Fonksiyonlar (Örnek: Öklid Algoritması)

- **Recursive (özyineli) fonksiyon**, kendi kendisini çağıran fonksiyondur.
- Genellikle matematiksel problemlerin çözümünde, algoritmaların doğal tanımlarını ifade etmek için kullanılır.
- Recursive fonksiyonlarda mutlaka bir **durdurma koşulu (base case)** bulunmalıdır, aksi halde fonksiyon sonsuz döngüye girer.

Örnek: Faktöriyel Hesabı

```
int faktoriyel(int n) {
    if (n == 0) return 1;       // durdurma koşulu
    else return n * faktoriyel(n - 1);
}
```

Örnek: Öklid Algoritması (EBOB Hesaplama)

```
int ebob(int a, int b) {
    if (b == 0) return a;       // durdurma koşulu
    return ebob(b, a % b);      // recursive çağrı
}
```

- Recursive fonksiyonlar kodun daha kısa ve anlaşılır olmasını sağlar, fakat fazla derin çağrılarda **bellek yığınının taşması (stack overflow)** riski vardır.

C++'ta Fonksiyon Aşırı Yükleme (Overloading)

- C++ dilinde aynı isimli birden fazla fonksiyon tanımlanabilir. Buna **fonksiyon aşırı yükleme (function overloading)** denir.
- Burada fonksiyonların **parametre türleri veya sayıları farklı** olmalıdır.
- Derleyici, fonksiyon çağrısını parametrelerin türüne ve sayısına göre çözümler.

Örnek:

```
int topla(int a, int b) {
    return a + b;
}
double topla(double a, double b) {
    return a + b;
}
```

Çağrı:

```
cout << topla(3, 4);     // int versiyonunu çağırır
cout << topla(3.2, 4.5); // double versiyonunu çağırır
```

- Overloading, kodun daha okunabilir ve esnek olmasını sağlar. Aynı işlevi farklı türlerle gerçekleştirmek mümkündür.

Ek Önemli Alt Başlıklar

Varsayılan Parametreler (Default Arguments)

- C++'ta fonksiyon parametrelerine varsayılan değer atanabilir. Eğer çağrı sırasında değer verilmezse bu varsayılan değer kullanılır.

```
void mesaj(string isim = "Ziyaretçi") {
   cout << "Merhaba " << isim;
}
mesaj();          // çıktı: Merhaba Ziyaretçi
mesaj("Ahmet");   // çıktı: Merhaba Ahmet
```

inline Fonksiyonlar

- Küçük ve sık kullanılan fonksiyonlarda performans kazanmak için `inline` anahtar kelimesi ile tanımlanabilir. Bu durumda fonksiyon çağrısı yerine doğrudan fonksiyonun gövdesi yerleştirilir.

Lambda Fonksiyonlar (C++11 ve sonrası)

- Kısa ve anonim fonksiyonlardır. Özellikle geçici işlemler için tercih edilir.

```
auto kare = [ ](int x) { return x * x; };
cout << kare(5);  // çıktı: 25
```

DİZİLER VE İŞLEMLERİ

- Diziler, aynı veri tipindeki birden fazla elemanı ardışık bellek alanlarında saklamaya yarayan veri yapılarıdır. Programlamada çok sayıda veriyi düzenli şekilde tutmak, üzerinde işlem yapmak ve hızlı erişim sağlamak amacıyla kullanılırlar. C ve C++ dillerinde diziler temel veri yapıları arasında yer alır.

Tek Boyutlu Diziler

Tanım: Tek boyutlu dizi, aynı türden verilerin bir sıra (liste) halinde tutulduğu yapılardır.

Tanımlama biçimi:

```
veri_tipi dizi_adi[eleman_sayisi];
```

Örneğin:

```
int notlar[5];     // 5 elemanlı bir int dizisi

float fiyatlar[3]; // 3 elemanlı bir float dizisi
```

- Dizilerde eleman sayısı sabittir ve tanımlandığında bellekte belirlenen miktarda alan ayrılır.
- Elemanlara indeks (index) üzerinden erişilir. İndeksler 0'dan başlar ve n elemanlı bir dizide son indeks n–1'dir.

```
notlar[0] = 80;

notlar[1] = 90;

cout << notlar[0]; // çıktı: 80
```

- Tek boyutlu diziler, genellikle öğrencilerin notları, ürün fiyatları, sıcaklık ölçümleri gibi doğrusal veri gruplarını saklamak için kullanılır.

Çok Boyutlu Diziler

Tanım: Çok boyutlu diziler, verilerin matris veya tablo şeklinde saklanmasına imkân tanır. En yaygın kullanılanı iki boyutlu dizilerdir.

Tanımlama biçimi:

```
veri_tipi dizi_adi[satir][sutun];
```

Örneğin:

```
int matris[3][3]; // 3x3 boyutunda bir matris
```

Elemanlara satır ve sütun indeksleri ile erişilir:

```
matris[0][0] = 5;

matris[1][2] = 10;

cout << matris[1][2]; // çıktı: 10
```

- Çok boyutlu diziler, özellikle matematiksel işlemlerde (ör. matris çarpımı), grafik işlemlerinde (görüntü işleme), oyun haritalarında veya tablo yapılarında kullanılır.

- İki boyutlu dizilerin ötesinde, üç boyutlu veya **daha fazla** boyutlu diziler de tanımlanabilir, ancak bellek yönetimi açısından dikkatli kullanılmalıdır.

Dizi Elemanlarına Erişim ve Güncelleme

- Dizi elemanlarına erişim indeks numarası ile yapılır. Her eleman bellekte ardışık olarak tutulur, bu yüzden erişim işlemi çok hızlıdır.

Örnek:

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

cout << sayilar[2];   // çıktı: 3

sayilar[2] = 10;      // 3 yerine 10 atanır

cout << sayilar[2];   // çıktı: 10
```

- Eğer diziye sınırların dışında erişilmeye çalışılırsa (out of bounds error), bellek hataları ortaya çıkabilir. C ve C++ dillerinde bu durum kontrol edilmez, dolayısıyla programcı dikkatli olmalıdır.

- Döngüler, dizilerin elemanlarını okumak veya güncellemek için sıkça kullanılır.

```
for (int i = 0; i < 5; i++) {
    sayilar[i] = i * 2;
}
```

- Bu örnekte dizinin tüm elemanları güncellenmiş olur.

Fonksiyonlara Dizi Gönderme

- C ve C++'ta fonksiyonlara dizi gönderilirken, aslında dizinin adı bellekteki ilk elemanın adresini temsil ettiği için referans (adres) yoluyla aktarım yapılır.
- Bu nedenle fonksiyon içinde yapılan değişiklikler, orijinal diziyi de etkiler.

Örnek:

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


int main() {
    int sayilar[5] = {1, 2, 3, 4, 5};
    yazdir(sayilar, 5);
}
```

- Eğer dizinin elemanları fonksiyon içinde değiştirilirse, bu değişiklikler ana programda da geçerli olur:

```
void degistir(int dizi[], int n) {
   for (int i = 0; i < n; i++) {
      dizi[i] = dizi[i] * 2;
   }
}
```

- C++'ta fonksiyonlara dizi göndermenin daha güvenli ve esnek yolu std::vector kullanmaktır. Vektörler dinamik boyutlu olduğu için dizilerin sınırlamalarını aşar.

Ek Önemli Alt Başlıklar

- Dizilerin Başlatılması (Initialization)
- Dizi tanımlanırken değerler doğrudan verilebilir.

```
int dizi[5] = {10, 20, 30, 40, 50};
```

- Eğer bazı değerler verilmezse, verilenlerden sonrası varsayılan olarak 0 ile doldurulur.
- Karakter Dizileri (String ve char dizileri)
- C dilinde metinler, '\0' ile sonlandırılan char dizileri olarak tutulur.

```
char kelime[] = “Merhaba”;
```

- C++ dilinde ise daha gelişmiş std::string sınıfı kullanılabilir.
- Dinamik Diziler
- Sabit boyutlu dizilerin aksine, malloc/free (C) veya new/delete (C++) ile dinamik bellekten dizi oluşturulabilir.
- Bu yöntem, kullanıcıdan alınan girişlere bağlı olarak boyutu değişkenlik gösteren veri yapıları için önemlidir.
- Çok Boyutlu Dizilerde Bellek Kullanımı
- İki boyutlu diziler bellekte satır satır saklanır (row-major order). Bu özellik, dizi üzerinde yapılan işlemlerin performansını etkileyebilir.

İŞARETÇİLER (POİNTERS)

- İşaretçiler, C ve C++ dillerinin en güçlü fakat aynı zamanda en dikkatli kullanılmasını gerektiren özelliklerinden biridir. Bir işaretçi, bellekteki bir adresi saklayan değişkendir. Bu sayede programcı, doğrudan bellek üzerinde işlem yapabilir, dinamik veri yapıları oluşturabilir ve düşük seviyeli donanım işlemlerini gerçekleştirebilir.

İşaretçi Tanımı ve Kullanımı

Tanım: İşaretçi (pointer), başka bir değişkenin bellekteki adresini saklayan değişkendir. Yani işaretçi, doğrudan değeri değil, o değerin adresini tutar.

- Bir işaretçi tanımlanırken veri tipinden sonra * işareti kullanılır:

```
int *p;
```

- Bu ifade, p'nin bir int türünden değişkenin adresini tutabileceğini belirtir.
- Bir değişkenin adresi, & (address-of) operatörü ile alınır.

```
int x = 10;

int *p = &x;   // p işaretçisi x'in adresini tutar
```

- İşaretçinin gösterdiği adresteki değere erişmek için * (dereference) operatörü kullanılır.

```
cout << *p;  // çıktı: 10

*p = 20;     // x'in değeri artık 20 olur
```

- Böylece işaretçiler aracılığıyla bellek üzerinde dolaylı erişim sağlanır.

Bellek Adresleri ile Çalışma

- C ve C++ dillerinde her değişken bellekte belirli bir adreste tutulur. İşaretçiler bu adreslere erişim sağlar.

Örnek:

```
int a = 5;

cout << &a;   // a'nın bellekteki adresini gösterir
```

- Bellek adresleri genellikle 16'lık sayı sistemi (hexadecimal) ile ifade edilir (örn. 0x7ffeefbff5c).

İşaretçiler sayesinde:

- Dinamik bellek yönetimi yapılabilir.
- Donanım adreslerine erişilebilir.
- Fonksiyonlar arasında büyük veri yapıları verimli bir şekilde aktarılabilir.
- Ancak yanlış adreslerle çalışmak (dangling pointers, null pointer dereferencing) programın çökmesine veya bellek hatalarına yol açabilir.

İşaretçiler ve Diziler

- Dizilerin adı, bellekte ilk elemanlarının adresini temsil eder. Bu nedenle diziler ve işaretçiler arasında güçlü bir ilişki vardır.

Örnek:

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

int *p = dizi;   // dizi ismi, dizi[0]'ın adresini temsil eder

cout << *p;      // çıktı: 10

cout << *(p+1);  // çıktı: 20
```

- p+1 ifadesi, işaretçiyi dizinin bir sonraki elemanına taşır. İşaretçi aritmetiği, bellekteki elemanlar arasında gezinebilmek için çok önemlidir.
- İşaretçiler kullanılarak dizi elemanlarına döngüler aracılığıyla erişilebilir:

```
for (int i = 0; i < 3; i++) {
   cout << *(p+i) << " ";
}
```

- Bu yapı, diziler üzerinde daha esnek ve düşük seviyeli işlemler yapılmasını sağlar.

Fonksiyonlara İşaretçi Parametre Geçişi

- Fonksiyonlara parametre aktarımı sırasında işaretçiler kullanılarak, fonksiyonun doğrudan orijinal değişken üzerinde işlem yapması sağlanabilir.

Örnek:

```
void degistir(int *ptr) {
    *ptr = *ptr + 10;   // işaret edilen değeri değiştirir
}

int main() {
    int a = 5;
    degistir(&a);
    cout << a;   // çıktı: 15
}
```

- Bu yöntem, pass by reference mantığının C'deki karşılığıdır.

- Özellikle büyük veri yapılarında (örneğin büyük diziler, yapılar veya matrisler) kopyalama yerine işaretçilerin kullanılması performansı ciddi şekilde artırır.

- C++ dilinde referanslar (&) işaretçilere daha güvenli bir alternatif olarak sunulmuştur, fakat işaretçiler hâlen sistem programlamada ve düşük seviyeli işlemlerde vazgeçilmezdir.

Ek Önemli Alt Başlıklar

Null Pointer (Boş İşaretçi):

- Bir işaretçi hiçbir adres göstermiyorsa NULL veya modern C++'ta nullptr ile tanımlanır.
- Null pointer, işaretçinin yanlış bellek bölgelerine erişmesini engellemek için güvenlik amacıyla kullanılır.

Dangling Pointer (Sallanmış İşaretçi):

- Geçersiz veya serbest bırakılmış bellek adreslerini göstermeye devam eden işaretçilerdir. Program çökmesine neden olabilir.

Pointer to Pointer (Çifte İşaretçi):

- İşaretçilerin adresini tutan işaretçilerdir.

```
int x = 10;

int *p = &x;

int **pp = &p;

cout << **pp;  // çıktı: 10
```

Dinamik Bellek Yönetimi:

- malloc, calloc, free (C) veya new, delete (C++) ile bellekte çalışma zamanında alan ayırmak mümkündür.
- Bu yapı özellikle dizilerin boyutunun önceden bilinmediği durumlarda tercih edilir.

Void Pointer (Genel Amaçlı İşaretçi):

- void * türü, her türden verinin adresini tutabilir, fakat üzerinde işlem yapılmadan önce uygun türe dönüş-türülmesi gerekir.

SINIFLAR VE NESNE YÖNELİMLİ PROGRAMLAMA (C++)

- C++ dilini C'den ayıran en önemli özellik, nesne yönelimli programlama (object-oriented programming, OOP) desteğidir. OOP, gerçek dünyadaki varlıkları yazılım içerisinde sınıf ve nesneler aracılığıyla modellemeyi sağlar. Bu yaklaşım, büyük ve karmaşık projelerin yönetilebilirliğini artırır, kodun modülerliğini sağlar ve yazılım geliştirme sürecinde yeniden kullanılabilirlik, güvenlik ve sürdürülebilirlik avantajı sunar.

Sınıf Tanımları ve Üyeleri

- Sınıf (class), C++ dilinde nesne yönelimli programlamanın temel yapı taşıdır. Sınıflar, verileri (özellikler) ve bu veriler üzerinde çalışan fonksiyonları (davranışlar) tek bir çatı altında toplar.
- Bir sınıf tanımlaması şu şekilde yapılır:

```
class Ogrenci {
    int yas;          // veri üyesi (data member)
    string isim;      // veri üyesi
public:
    void bilgileriYazdir(); // üye fonksiyon (member function)
};
```

- Sınıflar veri üyeleri (attributes) ve üye fonksiyonlardan (methods) oluşur. Veri üyeleri sınıfın durumunu, üye fonksiyonlar ise davranışlarını temsil eder.

- Sınıflardan türetilen somut varlıklara nesne (object) adı verilir.

```
Ogrenci ogr1;   // Ogrenci sınıfından bir nesne
```

Erişim Belirleyiciler: public, protected, private

- C++ dilinde veri güvenliğini sağlamak için erişim belirleyiciler (access specifiers) kullanılır.

private: Yalnızca sınıf içinden erişilebilir. Varsayılan erişim belirleyicidir.

protected: Sınıf içinden ve ondan türeyen alt sınıflardan erişilebilir.

public: Programın herhangi bir yerinden erişilebilir.

Örnek:

```
class Ogrenci {
private:
    int yas;
protected:
    string bolum;
public:
    string isim;
    void setYas(int y) { yas = y; }
    int getYas() { return yas; }
};
```

- Bu yapı sayesinde sınıf üyelerine kontrollü erişim sağlanır. Veri gizleme (encapsulation) prensibi de bu sayede uygulanır.

Yapıcı (Constructor) ve Yıkıcı (Destructor) Fonksiyonlar

Yapıcı fonksiyon (constructor):

- Sınıftan nesne oluşturulduğunda otomatik olarak çalışan özel fonksiyondur.
- İsmi sınıf adı ile aynı olmalı ve dönüş tipi olmamalıdır.
- Nesneye ilk değer ataması yapmak için kullanılır.

```
class Ogrenci {
public:
   Ogrenci(string ad) {  // yapıcı fonksiyon
      isim = ad;
   }
   string isim;
};
Ogrenci ogr1(“Ahmet”);
```

Yıkıcı fonksiyon (destructor):

- Nesne ömrü sona erdiğinde otomatik olarak çağrılır.
- İsmi ~ işareti ile başlar, parametre almaz, dönüş tipi olmaz.
- Dinamik bellek serbest bırakma gibi temizlik işlemleri için kullanılır.

```
~Ogrenci() {
   cout << "Nesne yok edildi.";
}
```

Üye Fonksiyonlara Erişim

- Sınıfın üye fonksiyonlarına erişmek için nokta (.) operatörü kullanılır.

```
Ogrenci ogr;
ogr.setYas(20);
cout << ogr.getYas();
```

- Eğer nesne işaretçi olarak tanımlanmışsa, ok (->) operatörü kullanılır:

```
Ogrenci *p = new Ogrenci("Ali");
p->setYas(22);
```

- Üye fonksiyonlar sınıf içinde doğrudan tanımlanabilir ya da sınıf dışında tanımlanarak scope resolution (::) operatörü ile bağlanabilir.

```
void Ogrenci::bilgileriYazdir() {

   cout << isim << " " << yas;

}
```

Arkadaş Fonksiyon (friend function)

- Normalde sınıfın private üyelerine yalnızca sınıf içinden erişilebilir. Ancak friend fonksiyon tanımlanarak özel izin verilmiş fonksiyonların bu üyelere erişmesine olanak sağlanır.

Kullanımı:

```
class Ogrenci {
private:
   int yas;
public:
   Ogrenci(int y) { yas = y; }
   friend void yazdir(Ogrenci o); // arkadaş fonksiyon
};
void yazdir(Ogrenci o) {
   cout << "Yaş: " << o.yas;
}
```

- friend fonksiyonlar nesneye ait değildir; sınıf dışındadır ama o sınıfın özel alanlarına erişebilir. Bu özellik dikkatli kullanılmalıdır, çünkü aşırı kullanımı kapsülleme (encapsulation) ilkesini zayıflatabilir.

Çok Biçimlilik (Polymorphism) ve Kalıtım (Inheritance)

Kalıtım (Inheritance):

- Bir sınıfın başka bir sınıftan özellik ve davranışları devralmasıdır.
- Kodun tekrarını azaltır, modülerliği artırır.

Örnek:

```
class Insan {
public:
    string isim;
    void konus() { cout << "Merhaba"; }
};

class Ogrenci : public Insan {
public:
    int numara;
};
```

- Burada Ogrenci sınıfı Insan sınıfından türemiştir ve onun üyelerine erişebilir.

Çok Biçimlilik (Polymorphism):

- Aynı işlevin farklı biçimlerde uygulanabilmesidir.
- C++'ta sanal fonksiyonlar (virtual functions) sayesinde gerçekleştirilir.
- Sanal fonksiyonlar, türeyen sınıflarda yeniden tanımlanabilir (override).

```
class Hayvan {
public:
    virtual void sesCikar() { cout << "Bilinmeyen ses"; }
};

class Kedi : public Hayvan {
public:
    void sesCikar() override { cout << "Miyav"; }
};
```

```
class Kopek : public Hayvan {
public:
    void sesCikar() override { cout << "Hav"; }
};

Hayvan* h1 = new Kedi();
Hayvan* h2 = new Kopek();
h1->sesCikar(); // Miyav
h2->sesCikar(); // Hav
```

- Kalıtım ve polimorfizm birlikte, C++'ta esnek, genişletilebilir ve yeniden kullanılabilir yazılım mimarileri oluşturmayı mümkün kılar.

Ek Önemli Alt Başlıklar

Soyut Sınıflar (Abstract Classes) ve Saf Sanal Fonksiyonlar (Pure Virtual Functions):

- Soyut sınıflar, doğrudan nesne türetilemeyen, yalnızca kalıtım yoluyla kullanılan sınıflardır.
- En az bir saf sanal fonksiyon (=0 ile tanımlanır) içerirler.

```
class Sekil {
public:
    virtual void alanHesapla() = 0; // saf sanal fonksiyon
};
```

Çoklu Kalıtım (Multiple Inheritance):

- C++ dilinde bir sınıf birden fazla sınıftan türeyebilir. Ancak bu durum diamond problem (elmas problemi) gibi karmaşık durumlara yol açabilir.

this İşaretçisi:

- Sınıfın kendi nesnesine işaret eder.
- Üye fonksiyonlar içinde kullanılarak mevcut nesnenin adresine erişim sağlar.

Statik Üyeler (static members):

- Tüm nesneler tarafından paylaşılan sınıf üyeleridir. Nesneye değil, sınıfın kendisine aittir.

BELLEK YÖNETİMİ

- Bellek yönetimi, programların çalışma zamanında ihtiyaç duyduğu verilerin bellekte uygun şekilde saklanması ve serbest bırakılması sürecidir. C ve C++ dillerinde bellek yönetimi, özellikle büyük ve dinamik veri yapılarının kullanıldığı durumlarda hayati önem taşır. Yanlış bellek kullanımı programların yavaşlamasına, bellek sızıntısına ve hatta sistem çökmesine yol açabilir.

Dinamik Bellek Tahsisi: malloc, calloc, free (C)

- C dilinde **statik bellek tahsisi**, değişkenlerin programın derlenme aşamasında bellekte yer almasıyla gerçekleşir. Ancak bazen çalışma zamanında kullanıcıdan alınan bilgilere göre bellek ayırmaya ihtiyaç duyulur. Bu durumda **dinamik bellek tahsisi** devreye girer.
- Dinamik bellek işlemleri için <stdlib.h> kütüphanesi kullanılır.

malloc (memory allocation)

- malloc, belirli bir miktarda ham bellek ayırır ve başlangıç değerlerini doldurmaz.
- Dönüş değeri void * türündedir, bu nedenle uygun türe dönüştürülmesi gerekir.

```
int *p;

p = (int*) malloc(5 * sizeof(int));  // 5 elemanlık int dizisi
```

- Eğer yeterli bellek bulunmazsa NULL döner.

calloc (contiguous allocation)

- calloc, malloc ile benzer işlev görür, ancak ayırdığı belleği sıfır ile doldurur.
- Ayrıca parametre olarak eleman sayısı ve eleman boyutu ayrı ayrı verilir.

```
int *p;

p = (int*) calloc(5, sizeof(int));  // 5 elemanlık int dizisi, hepsi 0 ile başlar
```

free

- malloc veya calloc ile ayrılan bellek, iş bittikten sonra serbest bırakılmalıdır. Aksi halde memory leak (bellek sızıntısı) oluşur.

```
free(p);
```

- free işleminden sonra işaretçinin yeniden kullanılmadan önce `NULL` yapılması iyi bir pratiktir.

new ve delete (C++)

- C++ dilinde dinamik bellek tahsisi için new ve delete operatörleri kullanılır. Bu operatörler, malloc ve free'ye göre daha güvenli ve nesne yönelimli programlama ile uyumludur.

new

- new, bellekte istenilen türden bir değişken veya dizi için alan ayırır ve adresini döndürür. Ayrıca nesneler için yapıcı fonksiyonları da çağırır.

```
int *p = new int;       // tek bir int için bellek

int *dizi = new int[5];  // 5 elemanlı dizi için bellek
```

- new başarısız olursa, varsayılan olarak bir istisna (bad_alloc) fırlatır.

delete

- new ile ayrılan bellek delete ile serbest bırakılmalıdır.

```
delete p;       // tek değişken için

delete[] dizi;   // dizi için
```

- malloc/free'de olduğu gibi delete işleminden sonra işaretçinin nullptr yapılması tavsiye edilir.

Bellek Sızıntısı (Memory Leak)

- Bellek sızıntısı, dinamik olarak ayrılan belleğin serbest bırakılmadan programın sona ermesi veya işaretçinin kaybolması durumunda ortaya çıkar.
- Bellek sızıntısı zamanla kullanılabilir belleğin azalmasına ve sistem performansının düşmesine neden olur.

Örnek:

```
void ornek() {
    int *p = new int[100];
    // delete[] p; yapılmazsa bellek sızıntısı oluşur
}
```

Bellek sızıntılarından kaçınmak için:

- Her malloc/calloc için bir free çağrısı,
- Her new için bir delete,
- Her new[ ] için bir delete[ ] çağrısı yapılmalıdır.

- Modern C++ uygulamalarında bellek yönetimini güvenli hâle getirmek için akıllı işaretçiler (smart pointers) önerilir:

std::unique_ptr → Tek sahiplik modeli.

std::shared_ptr → Paylaşımlı sahiplik modeli.

std::weak_ptr → Geçici referans modeli.

- Bu sınıflar sayesinde bellek yönetimi otomatik olarak yapılır ve sızıntı riski **en az**a iner.

Ek Önemli Alt Başlıklar

Stack ve Heap Bellek Kavramı:

Stack (yığın): Yerel değişkenler için ayrılır, otomatik olarak yönetilir. Fonksiyon bitince değişkenler silinir.

Heap (öbek): Dinamik bellek tahsisi yapılan bölgedir, manuel yönetilmesi gerekir.

Dangling Pointers (Sallanmış İşaretçiler):

- Belleği free veya delete ile serbest bıraktıktan sonra hâlen işaretçinin o bellek alanını göstermesi durumudur. Kullanımı programın çökmesine neden olabilir.

Double Free (Çifte Serbest Bırakma):

- Aynı bellek alanının **iki kez** free/delete edilmesi durumudur. Bu da program hatalarına yol açar.

RAII (Resource Acquisition Is Initialization):

- C++'ta bellek yönetiminde kullanılan bir tekniktir. Bir sınıfın yapıcısında kaynak (bellek, dosya, bağlantı) edinilir, yıkıcısında serbest bırakılır. Bu yöntem güvenli bellek yönetimi sağlar.

GİRDİ/ÇIKTI İŞLEMLERİ

- Girdi/çıktı işlemleri, kullanıcı ile program arasındaki etkileşimin temelini oluşturur. Programların işlevselliği, yalnızca hesaplamalar yapmakla sınırlı değildir; aynı zamanda dış dünyadan veri almak (input) ve işlenen bilgiyi kullanıcıya veya bir dosyaya aktarmak (output) da gereklidir. C ve C++ dillerinde girdi/çıktı işlemleri farklı kütüphaneler ve sözdizimleri aracılığıyla gerçekleştirilir.

printf ve scanf (C)

- C dilinde girdi/çıktı işlemleri için stdio.h kütüphanesi kullanılır.

printf (output)

- printf, ekrana yazı ve değişken değerlerini basmak için kullanılır.
- Format belirleyiciler (format specifiers) ile hangi türdeki verinin nasıl gösterileceği belirlenir.

Örnek:

```
int yas = 20;

float ort = 85.75;

printf("Yaş: %d, Ortalama: %.2f", yas, ort);
```

Kullanılan bazı format belirleyiciler:

- %d → int
- %f → float
- %.2f → ondalıklı sayıyı 2 basamakla yazdırır
- %c → char
- %s → string (char dizisi)
- scanf (input)
- scanf, kullanıcıdan veri almak için kullanılır.
- Kullanılan değişkenin adresini (&) parametre olarak ister.

Örnek:

```
int yas;

scanf("%d", &yas);

printf("Girilen yaş: %d", yas);
```

- scanf kullanılırken boşluk, tab veya enter gibi ayraçlarda veri girişi sonlanır. Metin alırken `gets` veya daha güvenli `fgets` tercih edilebilir.

cout ve cin (C++)

- C++ dilinde girdi/çıktı işlemleri için `iostream` kütüphanesi kullanılır.
- Daha modern, güvenli ve okunabilir sözdizimi sağlar.

cout (output)

- cout, ekrana veri yazdırmak için kullanılır.
- << (insertion) operatörü ile değişken veya sabitler çıktı akışına eklenir.

Örnek:

```
int yas = 20;
double notOrt = 90.5;
cout << "Yaş: " << yas << ", Ortalama: " << notOrt;
```

- endl ifadesi satır sonu ekler ve akışı temizler (flush).

```
cout << "Merhaba" << endl << "Dünya";
```

cin (input)

- cin, kullanıcıdan veri almak için kullanılır.
- >> (extraction) operatörü ile değişkenlere giriş yapılır.

Örnek:

```
int yas;
cout << "Yaşınızı giriniz: ";
cin >> yas;
cout << "Girdiğiniz yaş: " << yas;
```

- cin boşluk ve enter karakterine kadar olan girişi alır. Eğer boşluk içeren string almak gerekiyorsa getline(cin, degisken) fonksiyonu kullanılmalıdır.

```
string isim;
getline(cin, isim);
cout << "Merhaba " << isim;
```

Formatlı Çıktı İşlemleri

- Programlarda verilerin düzenli ve okunabilir biçimde ekrana yazdırılması için **formatlı çıktı** işlemleri kullanılır.

C'de Formatlı Çıktı

- printf fonksiyonunda format belirleyiciler ile sayıların genişliği, ondalık basamak sayısı ve hizalaması ayarlanabilir.

Örnek:

```
printf("%10d", 123);   // 10 karakterlik alanda sağa yaslı yazdırır
printf("%-10d", 123);  // sola yaslı yazdırır
printf("%.3f", 3.14159); // 3 basamaklı ondalık
```

C++'ta Formatlı Çıktı

- C++ dilinde <iomanip> kütüphanesi formatlı çıktı için kullanılır.

Örnekler:

```
#include <iomanip>

double pi = 3.1415926535;

cout << fixed << setprecision(3) << pi; // çıktı: 3.142

cout << setw(10) << 123;          // 10 karakterlik alanda sağa yaslı

cout << left << setw(10) << 123;      // sola yaslı
```

- Bu özellikler sayesinde raporlar, tablolar ve bilimsel veriler düzenli biçimde ekrana basılabilir.

Ek Önemli Alt Başlıklar

Dosya Girdi/Çıktı İşlemleri

- C'de fopen, fscanf, fprintf, fclose fonksiyonlarıyla;
- C++'ta <fstream> kütüphanesiyle (ifstream, ofstream, fstream) dosya üzerinde okuma ve yazma işlemleri yapılabilir.

Buffer (Tampon) ve Akış (Stream) Kavramı

- Girdi/çıktı işlemleri genellikle tampon bellek üzerinden yürütülür, böylece performans artırılır.

Hata Denetimi

- C'de scanf dönüş değerleri kontrol edilmeli, C++'ta ise cin.fail() fonksiyonu ile hatalı giriş olup olmadığı test edilmelidir.

SÜREÇLER VE SİSTEM ÇAĞRILARI

- Programlama dillerinde sadece kullanıcıyla etkileşim değil, aynı zamanda işletim sistemiyle doğrudan iletişim kurma ihtiyacı da vardır. C dilinin sistem programlamada kullanılmasının en önemli nedenlerinden biri, sistem çağrılarını (system calls) desteklemesidir. Sistem çağrıları, işletim sisteminin sunduğu hizmetlere (dosya yönetimi, süreç yönetimi, bellek yönetimi gibi) doğrudan erişim sağlar. Bu bağlamda süreç (process) kavramı, işletim sisteminin en temel birimlerinden biridir.

fork() Fonksiyonu ve Süreç Oluşturma

- Süreç (process): İşletim sistemi tarafından yürütülen, bellekte çalışan her bir program bir süreçtir. Her sürecin kendi adres alanı, program sayacı, yığın ve veri alanı vardır.
- C dilinde Unix/Linux tabanlı işletim sistemlerinde yeni bir süreç oluşturmak için fork() sistem çağrısı kullanılır.
- fork(), mevcut sürecin (ebeveyn – parent) bir kopyasını oluşturur ve bu kopyaya (çocuk – child) yeni bir süreç olarak devam ettirir.

- Kullanım örneği:

```
#include <stdio.h>

#include <unistd.h>


int main() {

   int pid = fork();

   if (pid == 0) {

      printf(“Bu satır çocuk süreç tarafından çalıştırıldı.\n”);

   } else {

      printf(“Bu satır ebeveyn süreç tarafından çalıştırıldı.\n”);

   }

   return 0;

}
```

fork() çağrısı **iki kez** döner:

- 0 değeri, çocuk süreçte (child process) döner.
- 0'dan büyük değer (pid), ebeveyn süreçte döner ve bu değer, çocuk sürecin kimliğini gösterir.
- Negatif değer, yeni süreç oluşturulamadığını (hata) gösterir.
- Bu mekanizma sayesinde aynı anda iki bağımsız süreç oluşur ve paralel olarak çalışmaya başlar.

Paralel Çalışan Süreçler ve Çıktıların Sayısı

- fork() fonksiyonundan sonra, hem ebeveyn hem de çocuk süreç aynı kodun farklı kopyalarını çalıştırır.

Bu nedenle ekrana basılan çıktılar sürecin hangi dallanmayı takip ettiğine göre değişir.

Örnek:

```
#include <stdio.h>

#include <unistd.h>



int main() {
   fork();
   fork();
   printf(“Merhaba Dünya\n”);
   return 0;
}
```

- Bu programın çıktısı **4 kez** "Merhaba Dünya" olacaktır. Çünkü:

» İlk fork() çağrısı iki süreç oluşturur.

» İkinci fork() çağrısı her iki süreçte de çalışır ve toplamda 4 süreç meydana gelir.

- Genel kural: Bir programda n **kez** fork() çağrılırsa, **en fazla 2**^n **adet** süreç oluşur (ilk süreç dahil).

- Paralel süreçlerin çalışması sırasıyla garanti edilmez; işletim sistemi CPU zamanlamasına göre süreçleri sırayla veya aynı anda çalıştırabilir. Bu nedenle çıktıların sırası her çalıştırmada farklı olabilir.

- Bu özellik, özellikle paralel programlama, çoklu işlem (multiprocessing) ve sistem programlaması konularında önem taşır.

Ek Önemli Alt Başlıklar

exec() Sistem Çağrıları:

- fork() ile oluşturulan bir süreç, aynı programın kopyasıdır. Eğer bu süreç tamamen yeni bir program çalıştırmak isterse, exec() ailesi (execv, execl, execvp vb.) fonksiyonları kullanılır.
- fork() + exec() kombinasyonu, yeni bir programı bağımsız bir süreçte çalıştırmak için en yaygın yöntemdir.

wait() Sistem Çağrısı:

- Ebeveyn sürecin, çocuk sürecin bitmesini beklemesi için kullanılır. Böylece süreçler arasında senkronizasyon sağlanır.

getpid() ve getppid():

- getpid(), mevcut sürecin kimliğini (process ID) döndürür.
- getppid(), ebeveyn sürecin kimliğini döndürür.

Süreçlerin Bellek Paylaşımı:

- fork() sonrası ebeveyn ve çocuk süreç birbirinden bağımsız bellek alanlarına sahiptir. Birinde yapılan değişiklik diğerine yansımaz.

- Ancak paylaşımlı bellek (shared memory) mekanizmaları kullanılarak süreçler arasında veri alışverişi yapılabilir.