Object Oriented Programming

Yahya Yeşilyurt - 25.05.2024

OOP01

Quality Characteristics of a Software System

ISO (International Organization for Standardization) ve IEC (International Electrotechnical Commission) kalite modelleri için standartları belirler.

ISO/IEC 25010: *Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.*

Bu standart iki tane kalite modeli içerir.

- **Quality in use model (Kullanımda kalite modeli):** Bu sistemin dış kalitesidir; belirli kullanım bağlamlarında paydaşlar (müşteriler, doğrudan ve dolaylı kullanıcılar vb.) üzerindeki etkidir.
- **Product Quality (Ürün kalitesi):** bu özellikler yazılım geliştirme ekibiyle ilgilidir.

Quality Attributes of a Software (External and Internal)

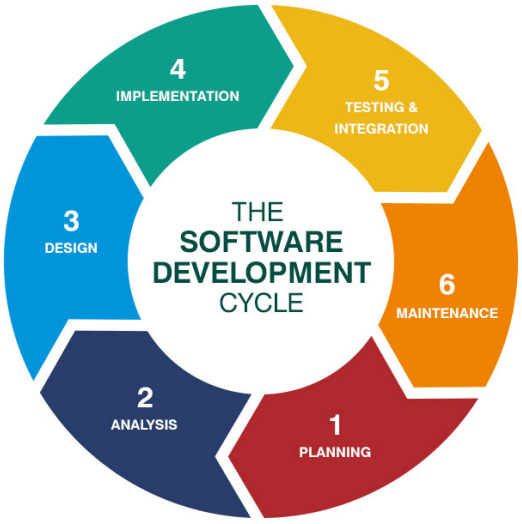
External / User

- Program işini doğru yapmalı. (effectiveness)
- Program gerektiği kadar hızlı çalışmalı. (time constraints)
- Sistem kaynaklarını çok fazla (effectiveness) gereksiz harcamamalı (processor time, memory, disk capacity, network capacity)
- Güvenilir olmalı.
- Kullanışlı ve kullanılabilir olmalı ve hakkında yeterince doküman olmalı (easy to learn and use)
- Güncellemesi, genişletilmesi, adapte edilmesi kolay olmalı (flexibility)

Internal / Software Developer

- İşlevsel olarak tamamlanmış ve doğru olmalı.
- Verimli olmalı (time behavior, resource utilization, capacity).
- Kaynak kodu okunabilir ve anlaşılabilir olmalı (comments, documentation).
- Yeni gereksinimlere göre kolay güncellenebilmeli ve yeni environementlere kolay adapte olabilmeli.
- Doğru hataları bulabilmesi için kolay test edilebilir olmalı.
- Programın modülleri daha sonraki projelerde kullanılabilir olmalı.

Software Development Process



The Unified (Software Development) Process - UP

Birleştirilmiş süreç (UP), object oriented sistemler oluşturmaya yönelik popüler bir yinelemeli yazılım geliştirme sürecidir. Birkaç best practise teşvik eder.

- **Iterative (yinelemeli):** Geliştirme, yineleme adı verilen bir dizi kısa, sabit uzunlukta (örneğin üç haftalık) mini projeler halinde düzenlenir; her birinin sonucu test edilmiş, entegre ve çalıştırılabilir bir kısmi sistemdir. Her yineleme kendi gereksinim analizini, tasarımını, uygulamasını ve test etkinliklerini içerir.
- **Incremental (artımlı), evolutionary (değerlendirici)**
- **Risk driven (risk odaklı)**

What is programming?

Program geliştirme, gerçek dünyadaki durumların modellerini oluşturmayı ve bu modellere dayalı bilgisayar programları oluşturmayı içerir.
Bilgisayar programları, gerçek dünya sorunlarına çözüm oluşturan şeylerin (nesnelerin) bilgisayar dünyası temsillerini içerebilir.

Imperative/Procedural Programming Technique

Imperative (emredici) veya procedural (prosedürel) programlama, bilgisayar programlarının belirli bir sırayla talimatlar vererek çalıştırıldığı bir programlama paradigmasıdır. İşlevsel programlamaya veya nesne yönelimli programlamaya karşıt olarak, bu paradigmada programlar, belirli bir sıra ve yöntemle talimatları gerçekleştirir.

Imperative/procedural programlama hakkında temel bazı kavramlar:

1. **Durum ve Değişkenler**: Programlama, değişkenlerin ve programın durumunun izlenmesine dayanır. Programın çalışması sırasında durum değişebilir ve değişkenler bu durumu temsil eder. Bu durum, programın bir noktasından diğerine taşınır ve değişkenlerin değerleri buna göre güncellenir.

2. **Kontrol Yapıları**: İmperatif programlama, sıralı, koşullu ve döngüsel kontrol yapılarını kullanır. Sıralı yapılar, talimatların belirli bir sıra içinde çalıştırılmasını sağlar. Koşullu yapılar, belirli bir koşulun doğru veya yanlış olmasına bağlı olarak farklı talimatların çalıştırılmasını sağlar. Döngüsel yapılar, belirli bir koşul sağlandığı sürece belirli talimatların tekrarlanmasını sağlar.
3. **Alt Programlar (Fonksiyonlar veya Prosedürler)**: İmperatif programlama, tekrar kullanılabilirlik ve modülerlik için alt programları kullanır. Bu alt programlar, belirli bir görevi yerine getiren talimat kümesidir. Alt programlar, ana program içinde çağrılabilir ve genellikle parametrelerle çalışabilirler.
4. **Bellek Yönetimi**: İmperatif programlama, bellek yönetimini doğrudan ele alır. Bellek, değişkenlerin değerlerini saklamak için kullanılır ve programın yürütülmesi sırasında dinamik olarak ayrılabilir ve serbest bırakılabilir.
5. **Nesnelerin Kullanımı**: İmperatif programlama paradigması genellikle nesne yönelimli programlama (OOP) ile karşılaştırılır. OOP'de programlar nesneler aracılığıyla organize edilirken, imperative programlama daha çok prosedürel bir yaklaşımı benimser. Bu, verilerin ve işlevlerin daha ayrık bir şekilde organize edilmesine yol açar.

İmperatif/Prosedürel programlama, birçok modern programlama dilinde kullanılan temel bir paradigmadır ve C, Pascal ve Fortran gibi birçok eski ve yeni programlama dilinde bulunabilir. Bu yaklaşım, özellikle düşük seviyeli sistem programlaması gibi alanlarda yaygın olarak kullanılır, ancak yüksek seviyeli uygulama geliştirmesi için de kullanılabilir.

İmperatif Programlamanın Dezavantajları:

1. **Karmaşıklık ve Büyüme**: Büyük projelerde, işlevler ve değişkenler arasındaki karmaşa artabilir. Büyüyen bir kod tabanı yönetilmesi zor olabilir.
2. **Bakım Güçlüğü**: İmperatif programlama, sıkça kullanılan kod bloklarının tekrar kullanılmasını zorlaştırabilir. Bu da kodun bakımını zorlaştırabilir ve hata ayıklamayı karmaşık hale getirebilir.
3. **Esneklik Sorunları**: İmperatif programlama, modülerlik ve yeniden kullanılabilirlik konularında OOP kadar esnek olmayabilir. Kod genellikle daha az yeniden kullanılabilir ve değiştirilmesi daha zor olabilir.
4. **Veri Gizliliği Zorlukları**: İmperatif programlama, veri gizliliği ve güvenliği konularında daha zayıf olabilir. Değişkenler ve işlevler sıklıkla genel alanlarda tanımlanır ve erişilebilir olabilir, bu da hatalara yol açabilir.

Nesne Yönelimli Programlamanın Avantajları:

1. **Modülerlik ve Yeniden Kullanılabilirlik**: OOP, modülerlik ve yeniden kullanılabilirlik konularında güçlüdür. Nesneler, kodun tekrar kullanılabilir parçaları olarak tasarlanabilir ve farklı projelerde kolayca kullanılabilir.
2. **Kodu Daha Kolay Anlama**: Nesne yönelimli programlama, gerçek dünya nesnelerine daha iyi bir şekilde karşılık gelir, bu da genellikle kodun daha açık ve anlaşılır olmasını sağlar. Bir nesnenin davranışını anlamak, genellikle sadece nesnenin yöntemlerini incelemekle mümkündür.
3. **Veri Gizliliği ve Güvenlik**: OOP, veri gizliliği ve güvenlik konularında daha güçlüdür. Sınırlı erişim denetimi ve kapsülleme ile değişkenler ve işlevlerin kontrol edilmesi daha kolaydır.
4. **Kodun Daha Kolay Bakımı**: Nesne yönelimli programlama, kodun bakımını kolaylaştırır. Kod, nesnelerin ve sınıfların uygun bir şekilde yapılandırılması sayesinde daha modüler hale gelir, bu

da değişikliklerin yapılmasını ve hataların bulunmasını daha kolaylaştırır.

Sonuç olarak, nesne yönelimli programlama, genellikle büyük ölçekli ve karmaşık projeler için daha uygun olan bir programlama paradigmasıdır. Ancak, ihtiyaçlarınıza ve projenize bağlı olarak, bazen basit, küçük projelerde veya donanım düşük seviyeli programlama gibi belirli durumlarda, imperative/prosedürel programlama tercih edilebilir olabilir.

What is an object?

Real-world object		Software object
Attributes	→	Data
Behavior	→	Functions (methods)

Gerçek dünya nesneleri ile yazılım nesneleri arasında benzerlikler ve ilişkiler vardır çünkü yazılım nesneleri genellikle gerçek dünyadaki nesnelerin modelleridir. İşte bu benzerlikler ve ilişkilerin bazıları:

1. **Varlık ve Özellikler**: Hem gerçek dünya nesneleri hem de yazılım nesneleri, belirli bir varlığı temsil eder. Gerçek dünyada bir araba gibi somut bir nesne olabilir veya bir müşteri gibi soyut bir kavram olabilir. Her iki durumda da, nesnelerin özellikleri veya nitelikleri vardır. Gerçek dünyada bir arabanın rengi, markası, modeli gibi özellikler olabilirken, bir müşterinin adı, adresi gibi özellikleri olabilir.
2. **Davranışlar ve Metodlar**: Hem gerçek dünya nesneleri hem de yazılım nesneleri, belirli davranışları gerçekleştirebilir. Gerçek dünyada bir araba sürülebilir, bir müşteri sipariş verebilir gibi, yazılım nesneleri de benzer şekilde işlevsellik sağlayabilir. Yazılım nesneleri bu davranışları gerçekleştirmek için metodlar veya fonksiyonlar içerebilir.
3. **İlişkiler ve Etkileşim**: Gerçek dünya nesneleri arasında ilişkiler olabilir. Örneğin, bir müşteri bir sipariş verebilir ve bu sipariş bir ürünle ilişkilidir. Benzer şekilde, yazılım nesneleri arasında da benzer ilişkiler olabilir. Bir müşteri nesnesi bir sipariş nesnesi oluşturabilir ve bu sipariş nesnesi bir ürün nesnesiyle ilişkilendirilebilir.
4. **Kalıtım ve Özelleştirme**: Gerçek dünyada, belirli bir nesnenin özellikleri ve davranışları, o nesnenin türetildiği daha genel bir kategoriye dayanabilir. Örneğin, bir kamyonet bir araçtır ve araçların sahip olduğu özellikleri ve davranışları paylaşır. Yazılım nesneleri arasında da benzer bir kalıtım ve özelleştirme kavramı vardır. Bir sınıf, diğer sınıflardan türetilebilir ve bu türetilmiş sınıf, ana sınıfın özelliklerini ve davranışlarını kalıtım yoluyla alabilir ve özelleştirebilir.

Bu benzerlikler ve ilişkiler, yazılım geliştiricilerinin gerçek dünya problemlerini modellemesine ve çözmesine yardımcı olur. Nesne yönelimli programlama paradigması, bu tür benzerlikleri ve ilişkileri kavramak ve yazılım tasarımını gerçek dünya problemlerine daha yakın hale getirmek için kullanılır. Bu sayede, daha modüler, esnek ve yeniden kullanılabilir yazılım çözümleri oluşturulabilir.

Key terms the object oriented approach: Encapsulation - Data Hiding

Nesne yönelimli programlamada (OOP), "encapsulation" (kapsülleme) ve "data hiding" (veri gizleme) önemli kavramlardır. Bu kavramlar, yazılım nesnelerinin iç durumlarını ve davranışlarını korumak ve kontrol etmek için kullanılır.

1. **Encapsulation (Kapsülleme)**:
    - Encapsulation, bir nesnenin veri alanlarını (state) ve bunların üzerindeki işlevselliği (behavior) bir araya getirme işlemidir.
    - Bir nesnenin iç durumu (veri alanları) ve bu durum üzerindeki operasyonlar (metodlar) bir kapsül içine yerleştirilir.
    - Bu durum, dış dünyadan gelen istenmeyen müdahalelere karşı korunur ve sadece belirli arayüzler aracılığıyla erişilebilir hale getirilir.
    - Encapsulation, nesnenin iç detaylarını gizleyerek, dış dünyayla arasında bir sınır oluşturur ve nesnenin iç yapısının dış dünya tarafından değiştirilmesini engeller.
    - Encapsulation, kodun daha modüler ve daha az bağımlı hale gelmesine olanak tanır, böylece yazılımın bakımı ve genişletilmesi kolaylaşır.
2. **Data Hiding (Veri Gizleme)**:
    - Data hiding, bir nesnenin iç durumlarını dış dünyadan gizlemeyi ifade eder.
    - Bir nesnenin veri alanlarına doğrudan erişim sağlayan işlevselliği kısıtlayarak, nesnenin iç detaylarının gizlenmesini sağlar.
    - Genellikle, veri alanları private veya protected erişim düzeyine sahip olacak şekilde tanımlanır ve sadece nesnenin kendi metodları tarafından erişilebilir.
    - Bu, nesnenin iç durumunun istenmeyen bir şekilde değiştirilmesini önler ve nesnenin tutarlılığını ve bütünlüğünü korur.
    - Data hiding, kodun daha güvenli olmasını sağlar çünkü sınırlı erişim, hatalı kullanımı ve yanlışlıkla yapılan değişiklikleri önler.

Özetlemek gerekirse, encapsulation ve data hiding, nesne yönelimli programlamada yazılım nesnelerinin iç durumlarını korumak ve kontrol etmek için kullanılan önemli kavramlardır. Bu kavramlar, yazılımın daha güvenli, modüler ve esnek olmasını sağlar ve karmaşıklığı azaltarak bakımı kolaylaştırır.

Example of an object: a point in a graphics program

```
#include <iostream>

class Point {
private:
    double x;
    double y;
    bool hidden; // Gizlenmiş mi değil mi?

public:
    // Constructor
    Point(double x = 0.0, double y = 0.0) : x(x), y(y), hidden(false) {}

    // Getter and Setter methods
    double getX() const {
```

```
        return x;
    }

    double getY() const {
        return y;
    }

    void setX(double newX) {
        x = newX;
    }

    void setY(double newY) {
        y = newY;
    }

    // Method to move the point
    void move(double dx, double dy) {
        x += dx;
        y += dy;
    }

    // Method to print the point's coordinates
    void print() const {
        std::cout << "Point coordinates: (" << x << ", " << y << ")" << std::
    }

    // Method to hide the point
    void hide() {
        hidden = true;
    }
};

int main() {
    // Creating a Point object
    Point point(3, 4);

    // Printing initial position
    point.print(); // Çıktı: Point coordinates: (3, 4)

    // Moving the point
    point.move(1, -2);
    point.print(); // Çıktı: Point coordinates: (4, 2)

    // Hiding the point
    point.hide();
```

```
    return 0;
}
```

Bu kod, `Point` sınıfını ve onun özelliklerini içerir: `x` ve `y` koordinatları, `move` fonksiyonu (noktayı hareket ettirir), `print` fonksiyonu (noktanın koordinatlarını yazdırır) ve `hide` fonksiyonu (noktayı gizler). `main` fonksiyonunda bu yöntemlerin nasıl kullanılacağı örneklendirilmiştir.

OOP02

Declarations and Definitions in C++

Declarations (Bildirimler):

Bir değişkenin veya fonksiyonun varlığını bildiren, ancak onu tanımlamayan ifadelerdir. Bildirme, bir değişkenin veya fonksiyonun adını, türünü ve bazı durumlarda parametrelerini belirtir.

```
// Function declaration
int add(int x, int y);
```

Yukarıdaki örnek, `add` fonksiyonunun varlığını bildirir, ancak bu fonksiyonun tam olarak nasıl çalıştığını veya ne yaptığını tanımlamaz. Bu, `add` fonksiyonunun başka bir yerde kullanılmadan önce, bu fonksiyonun var olduğunu derleyiciye bildirmek için kullanılabilir.

```
// Variable declaration
extern int count;
```

Yukarıdaki örnek, `count` adlı bir değişkenin varlığını bildirir, ancak bu değişkenin değerini veya nasıl kullanılacağını tanımlamaz. Bu, `count` değişkeninin başka bir dosyada tanımlanmış olduğunu ve bu dosyanın içinde bu değişkene erişim olacağını belirtmek için kullanılabilir.

Definitions (Tanımlamalar):

Bir değişkenin veya fonksiyonun varlığını bildirirken aynı zamanda onu tanımlayan ifadelerdir. Tanımlama, bir değişkenin değerini atar veya bir fonksiyonun işlevselliğini belirtir.

```
// Function definition
int add(int x, int y) {
    return x + y;
}
```

Yukarıdaki örnek, `add` fonksiyonunun ne yaptığını ve nasıl çalıştığını belirtir. Bu, fonksiyonun gerçek işlevselliğini tanımlar ve çağrıldığında hangi işlemleri gerçekleştireceğini belirtir.

```
// Variable definition
int count = 0;
```

Yukarıdaki örnek, `count` adlı bir değişkenin ne olduğunu ve başlangıç değerinin ne olduğunu belirtir. Bu, değişkenin bellekte nasıl yer alacağını ve hangi değere sahip olacağını tanımlar.

Farklar:

- Declaration, varlığın sadece adını, türünü ve bazı durumlarda parametrelerini belirtirken, definition, varlığın tam olarak ne olduğunu veya ne yaptığını belirtir.
- Declaration, bir varlığın var olduğunu derleyiciye bildirmek için kullanılırken, definition, bir varlığın işlevselliğini veya değerini belirtmek için kullanılır.
- Declarationda `extern` anahtar kelimesi kullanılabilirken, definitionda bu kullanım mümkün değildir.

> Tüm definitionlar aynı zamanda bir declarationdur ama declarationlar definition değildir.

Examples:

```
extern int i; // Declaration, the definition is in another file
int i;        // Definition, memory is allocated
--------------------------------------------------------------------------
struct ComplexT;     // Declaration only
struct ComplexT {    // Declaration (type) and definition of complex numbers
    double re{}, im{};
};
ComplexT c1, c2;     // Definition of two complex number variables c1, c2
--------------------------------------------------------------------------
void function(int,int);  // Declaration (its body is the definition)
void function(int, int){ // Definition
    ...
}
--------------------------------------------------------------------------
class Point;     // Declaration only
class Point {    // Declaration and Definition of Point Class
    public:
        void move(int, int);  // Declaration of the function to move the Poin
    private:
        int x{}, y{};         // Definition of the properties: x and y coordi
};
Point point1, point2;     // Definition of two Point objects
```

One Definition Rule (ODR)

C++'taki "One Definition Rule", bir programın her bir fonksiyonunun veya global nesnesinin sadece bir kez tanımlanması gerektiğini belirten bir kuraldır. Bu kural, birden fazla kaynak dosyasında veya birden fazla derleme biriminde tanımlanmış olan aynı isimli fonksiyonlar veya global nesneler arasındaki çakışmaları önler.

One Definition Rule, C++ standartlarına sıkı bir şekilde uygulanır ve genellikle hata ayıklama ve kodun sağlığı açısından önemlidir. One Definition Rule ihlalleri, öngörülemeyen davranışlar veya programın hatalı çalışmasına neden olabilir.

Bu kuralın temel amacı, herhangi bir ismin, birim veya hedef dosyada yalnızca bir kez tanımlanmasını sağlamaktır. Bununla birlikte, belirli durumlarda, aynı ismin birden fazla kez bildirilmesine izin verilebilir, ancak yalnızca bir tanesi tanımlanabilir.

Örneğin, bir header dosyasında bir fonksiyon prototipi (bildirimi) olabilir ve bu fonksiyonun farklı kaynak dosyalarında tanımı (tanımlaması) olabilir. Ancak, bu fonksiyonun birden fazla tanımının (definition) olması bir hata oluşturur.

Örnek:

```
// header file: functions.h
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

int add(int a, int b); // Fonksiyonun bildirimi

#endif
```

```
// kaynak dosyası: functions.cpp
#include "functions.h"

int add(int a, int b) { // Fonksiyonun tanımı
    return a + b;
}
```

```
// farklı bir kaynak dosyası: main.cpp
#include "functions.h"

int main() {
    int result = add(3, 4);
    return 0;
}
```

Yukarıdaki örnekte, `add` fonksiyonu `functions.h` başlık dosyasında bildirilmiş ve `functions.cpp` kaynak dosyasında tanımlanmıştır. Ayrıca, `main.cpp` dosyasında bu fonksiyonun kullanımı sağlanmıştır. Bu şekilde, One Definition Rule ihlali önlenir ve program düzgün şekilde derlenir ve çalışır.

Namespaces

C++ dilinde, "namespaces" (isim alanları), isim çakışmalarını önlemek, kodun organize edilmesini sağlamak ve kodun okunabilirliğini artırmak için kullanılan bir özelliktir. Bir namespace, bir grup ilgili ismin bir araya getirildiği mantıksal bir bölüm veya kapsayıcıdır.

Bir namespace, global bir isim alanı olabilir veya başka bir namespace içinde yer alabilir. İsim alanları, bir `namespace` anahtar kelimesi ve ilgili isimlerin `::` operatörüyle birleştirilmesiyle tanımlanır.

İsim alanları, kodun okunabilirliğini artırır ve büyük projelerde isim çakışmalarını önler. Özellikle büyük kütüphaneler veya çerçeveler kullanıldığında, bu isim alanları önemlidir çünkü farklı kütüphaneler veya bileşenler aynı isimlere sahip olabilirler.

Örnek olarak, bir matematik kütüphanesi içinde `math` adında bir namespace kullanalım:

```
#include <iostream>

// Math namespace tanımı
namespace math {
    const double pi = 3.14159;

    double square(double x) {
        return x * x;
    }
}

int main() {
    // Math namespace'ten pi sabitine erişim
    std::cout << "Pi: " << math::pi << std::endl;

    // Math namespace'ten square fonksiyonuna erişim
    std::cout << "Square of 5: " << math::square(5) << std::endl;

    return 0;
}
```

Bu örnekte, `math` adında bir namespace tanımlanmıştır. İçinde bir `pi` sabiti ve bir `square` fonksiyonu bulunmaktadır. Main fonksiyonunda, bu namespace'ten sabit ve fonksiyonlara erişim sağlanmıştır. `math::` operatörü, `pi` ve `square` isimlerinin namespace içinde olduğunu belirtir.

Bu şekilde, `math` namespace'i içindeki isimler, diğer kod bloklarındaki isim çakışmalarını önler ve kodun okunabilirliğini artırır. Ayrıca, bu namespace kullanımı, projede birden çok isim alanını organize etmek ve kütüphaneleri daha modüler hale getirmek için de kullanılabilir.

using declaration

C++'ta, `using` bildirimi veya `using` declaration, bir isim alanındaki belirli bir ismi, daha kısa bir şekilde kullanmak için kullanılan bir yapıdır. `using` declaration, özellikle uzun isim alanlarıyla çalışırken kodu daha okunabilir hale getirmek için kullanışlıdır.

`using` declaration, bir ismi bir isim alanından genel kapsama taşır. Bu, belirli bir ismi namespace'ten çıkararak, adını doğrudan kullanılabilir hale getirir.

Örnek olarak, `std` isim alanındaki `cout` ve `endl` isimlerini kullanmak için `using` declaration kullanabiliriz:

```
#include <iostream>

int main() {
    using std::cout;
    using std::endl;

    cout << "Hello, world!" << endl;

    return 0;
}
```

Yukarıdaki örnekte, `using` declaration kullanılarak `cout` ve `endl` isimleri `std` isim alanından genel kapsama taşınmıştır. Böylece, `std::` ön ekini her kullanımdan kaldırarak kodu daha okunabilir hale getirir.

Bir `using` declaration, bir dosya veya bir kod bloğu içinde tanımlanabilir. `using` declaration, o belirli kod bloğunda veya dosyada geçerlidir. Bu nedenle, genellikle dosyanın başında veya fonksiyonun başında kullanılır.

Ancak, `using` declaration'ın aşırı kullanımı, kodun okunabilirliğini azaltabilir ve isim çakışmalarına neden olabilir. Bu nedenle, `using` declaration'ın dikkatli bir şekilde kullanılması önerilir.

Bir `using` declaration'ın bir namespace içindeki tüm elemanlara uygulanabileceği bir örnek şu şekilde olabilir:

```
#include <iostream>

namespace math {
    const double pi = 3.14159;

    double square(double x) {
        return x * x;
    }

    double cube(double x) {
        return x * x * x;
    }
}

int main() {
    using namespace math;

    std::cout << "Pi: " << pi << std::endl;
    std::cout << "Square of 5: " << square(5) << std::endl;
    std::cout << "Cube of 3: " << cube(3) << std::endl;

    return 0;
}
```

Bu örnekte, `math` isim alanındaki tüm elemanlara `using namespace math;` kullanılarak erişim sağlanmıştır. Böylece, `pi`, `square` ve `cube` isimleri doğrudan kullanılabilir hale gelir. Bu sayede, bu isimleri her seferinde `math::` ön ekiyle çağırmak zorunda kalmadan doğrudan kullanabiliriz.

Ancak, bu yaklaşım genellikle önerilmez çünkü isim alanını genel kapsamda getirir ve isim çakışmalarına yol açabilir. Bu nedenle, `using namespace` ifadesinin aşırı kullanımından kaçınılmalıdır. Bunun yerine, belirli isimleri veya belirli fonksiyonları belirli bir isim alanından getirmek için `using` declaration'lar kullanılmalıdır. Bu, kodun okunabilirliğini artırır ve isim çakışmalarını önler.

Working with multiple files (headers and modules)

C++ programlamada, "header files" (başlık dosyaları) ve "modüller" kodun organizasyonunu ve yeniden kullanılabilirliğini sağlamak için önemli yapısal unsurlardır. Her ikisi de kodun bölünmesini ve modülerleştirilmesini sağlar, ancak farklı mekanizmaları ve kullanımları vardır.

Header Files (Başlık Dosyaları):

Header files, genellikle `.h` veya `.hpp` uzantılı dosyalardır ve bir programın kullanabileceği fonksiyon prototiplerini, sınıf tanımlamalarını, sabitleri ve diğer deklarasyonları içerir. Header files, genellikle bir programın "public interface" (genel arayüz) kısmını temsil eder ve başka bir dosyadan erişilebilir olması amaçlanan bileşenlerin tanımlarını içerir.

Örnek olarak, `math.hpp` başlık dosyası:

```
#ifndef MATH_HPP
#define MATH_HPP

namespace math {
    const double pi = 3.14159;

    double square(double x);
    double cube(double x);
}

#endif
```

Yukarıdaki örnek, `math` isim alanındaki `square` ve `cube` fonksiyonlarının prototiplerini içeren bir başlık dosyasıdır.

Modüller:

C++20'den itibaren, C++ diline modüller de eklenmiştir. Modüller, bir dosyanın içeriğini başka bir dosyaya eklemenin gelişmiş bir yoludur ve header files'ların bir alternatifidir. Modüller, kodu daha organize eder ve bağımlılıkları daha iyi yönetir.

Örnek olarak, `math` modülü:

```
// math.cppm
export module math;

export const double pi = 3.14159;
```

```
export double square(double x) {
    return x * x;
}

export double cube(double x) {
    return x * x * x;
}
```

Yukarıdaki örnekte, `math` modülü `math.cppm` dosyasında tanımlanmıştır. Modüller, `export` anahtar kelimesiyle belirli bileşenlerin (değişkenler, fonksiyonlar) diğer dosyalardan kullanılabilir hale getirilmesini sağlar.

Örnek Kullanım:

Başlık dosyasının veya modülün kullanımı:

```
#include <iostream>
#include "math.hpp" // veya import math;

int main() {
    std::cout << "Pi: " << math::pi << std::endl;
    std::cout << "Square of 5: " << math::square(5) << std::endl;
    std::cout << "Cube of 3: " << math::cube(3) << std::endl;

    return 0;
}
```

Yukarıdaki örnekte, `math` başlık dosyası veya modülü kullanılarak `pi`, `square` ve `cube` bileşenlerine erişim sağlanmıştır. Bu şekilde, kodun organizasyonu sağlanır ve yeniden kullanılabilirlik artırılır.

Header dosyaları ve modüller arasında birkaç farklılık ve avantaj/dezavantaj bulunmaktadır:

Header Dosyalarının Avantajları:

1. **Geçmiş Uyumluluğu**: Header dosyaları, eski C++ standartlarıyla uyumluluk sağlar ve mevcut projelerde kullanılan yaygın bir yöntemdir.
2. **Derleme Süresi Performansı**: Header dosyaları, sadece ihtiyaç duyulan bileşenleri içermek yerine, tüm kodu içereceği için derleme süresini artırabilir. Ancak, bazı derleyiciler "header dosyası içindeki kodu yalnızca bir kez derleme" (header dosyası koruma) gibi optimizasyonlar sunarak bu sorunu hafifletebilir.
3. **Platform Bağımsızlığı**: Header dosyaları, C++'ın tüm platformlarda ve derleyicilerde desteklendiği yaygın bir yöntemdir.

Modüllerin Avantajları:

1. **Derleme Süresi Performansı**: Modüller, yalnızca gerektiğinde kullanılan kod bloklarını içerir ve sadece bir kez derlenir, bu da derleme süresini önemli ölçüde azaltabilir.
2. **Kod Tekrarı ve Hata Riskinin Azalması**: Modüller, kod tekrarını önler ve hata riskini azaltır çünkü bir dosyanın içeriği birçok farklı kaynaktan aynı anda alınmaz.

3. **İsim Alanı Kirliliğinin Azalması**: Modüller, isim çakışmalarını önler ve kodun daha temiz ve okunabilir olmasını sağlar.
4. **Optimizasyon Olanakları**: Modüller, derleyicinin daha iyi optimizasyonlar yapmasına izin verir çünkü bir modülün bağımlılıkları ve kullanılan diğer modüller derleyici tarafından daha iyi anlaşılabilir.
5. **Yeniden Yapılandırılabilirlik ve Geliştirilebilirlik**: Modüller, kodu daha modüler hale getirir ve daha kolay bakım ve geliştirme sağlar.

Dezavantajlar:

1. **Yetersiz Desteğin Olması**: Bazı eski derleyiciler ve platformlar, modülleri tam olarak desteklemez, bu da bazı projelerde kullanımını sınırlayabilir.
2. **Geçiş Süreci**: Mevcut projelerde header dosyalarından modüllere geçiş yapmak zaman alabilir ve bazı zorluklarla karşılaşılabilir.
3. **Karmaşıklık**: Modüller, başlangıçta karmaşık görünebilir ve bazı geliştiriciler için anlamak ve kullanmak daha zor olabilir.

Her iki yöntemin avantajları ve dezavantajları vardır ve projenin ihtiyaçlarına ve gereksinimlerine bağlı olarak tercih edilebilirler. Özellikle yeni projelerde modüllerin kullanılması, kodun derlenme süresini azaltmak, temizliği ve bakımı artırmak için iyi bir seçenek olabilir. Ancak, var olan büyük projelerde, geçiş süreci ve geriye dönük uyumluluk nedeniyle header dosyalarının kullanılması daha yaygın olabilir.

Modüllerin derlenme süresini azaltmak için kullanılan iki önemli optimizasyon tekniği precompiled headers (ön derlemeli başlık dosyaları) ve caching (ön belleğe alma) olarak adlandırılabilir. Bu teknikler, derleme süresini azaltmak ve projelerin daha hızlı derlenmesini sağlamak için kullanılır.

Precompiled Headers (Ön Derlemeli Başlık Dosyaları):

Precompiled headers, sıkça kullanılan başlık dosyalarının derleme süresini azaltmak için ön derlenmiş bir formda saklanmasını sağlar. Bu başlık dosyaları, projenin genellikle her dosyasında kullanılan ve neredeyse her derleme işleminde yeniden derlenmesi gereken başlık dosyalarıdır.

Örneğin, bir projede sıkça kullanılan standart kütüphane başlık dosyaları (`<iostream>`, `<vector>`, `<string>` vb.) ön derlenmiş bir forma dönüştürülerek saklanabilir. Böylece, bu başlık dosyaları her dosyanın derlenmesi sırasında yeniden derlenmek yerine, önceden derlenmiş bir formda kullanılabilir.

Caching (Ön Belleğe Alma):

Caching, derleme işlemlerinin sonuçlarını önbelleğe alarak tekrarlanan derleme işlemlerinde zaman kazanılmasını sağlar. Bu, derleme sürecinde oluşan ara ürünlerin (derleme çıktıları) önbelleğe alınarak, gelecekteki derlemelerde tekrar kullanılmasını sağlar.

Örnek olarak, bir dosyanın derleme çıktısı (obje dosyası veya derleme sırasında üretilen diğer ara dosyalar) önbelleğe alınabilir. Daha sonra, bu dosya değişmediği sürece, aynı derleme işlemi tekrarlandığında önbelleğe alınmış derleme çıktısı kullanılabilir ve derleme süresi önemli ölçüde azaltılabilir.

BMI (Binary Module Interface):

BMI, C++20 standardında tanıtılan bir kavramdır ve modüllerin ön derlemeli bir formda saklanmasını sağlar. BMI, derlenmiş modül dosyalarının ara ürünlerini depolayarak derleme süresini azaltır ve modüllerin derlenme sürecini hızlandırır.

BMI'nin temel amacı, modüllerin derlenmiş bir formda saklanmasını sağlamak ve yeniden derlenmesi gerektiğinde bu derlenmiş modül dosyalarını kullanarak derleme süresini azaltmaktır.

Bu teknikler, büyük projelerde önemli ölçüde derleme süresi kazanımı sağlar ve geliştiricilere daha hızlı bir geliştirme deneyimi sunar.

Creating Modules

C++20'den itibaren modüllerin kullanımı yaygınlaşmıştır. Modüller, `module` anahtar kelimesi ile tanımlanır ve modülün arayüzü ve implementasyonu ayrı dosyalarda tanımlanabilir. İşte bir örnek:

Module Interface (Modül Arayüzü):

```
// math.ixx veya math.cppm

export module math;

namespace math {
    double square(double x);
    double cube(double x);
}
```

Yukarıdaki örnekte, `math` modülünün arayüzünü belirleyen `math.hpp` dosyası bulunmaktadır. `square` ve `cube` fonksiyonlarının prototipleri burada tanımlanmıştır.

Module Implementation (Modül Implementasyonu):

```
// math.cpp

module math;

namespace math {
    double square(double x) {
        return x * x;
    }

    double cube(double x) {
        return x * x * x;
    }
}
```

Yukarıdaki örnekte, `math` modülünün implementasyonunu belirleyen `math.cpp` dosyası bulunmaktadır. `square` ve `cube` fonksiyonlarının gerçek kodları burada tanımlanmıştır.

Kullanım:

```
// main.cpp

import math;

#include <iostream>

int main() {
    std::cout << "Square of 5: " << math::square(5) << std::endl;
    std::cout << "Cube of 3: " << math::cube(3) << std::endl;

    return 0;
}
```

Yukarıdaki örnekte, `main.cpp` dosyası `math` modülünün arayüzünü içeri alır ve bu modülde tanımlı olan `square` ve `cube` fonksiyonlarını kullanır.

Bu şekilde, modül arayüzü ve implementasyonu ayrı dosyalarda tanımlanır ve kullanımı kolay hale getirilir. Ayrıca, modülün arayüzü ve implementasyonu bağımsız olarak derlenir, böylece modüllerin kullanımı daha hızlı ve etkili hale gelir.

Input/Output

C++ dilinde input/output işlemleri için standart input/output akışları (`std::cin`, `std::cout`, `std::cerr`, `std::clog`) ve C++20'den itibaren `std::print()` ve `std::println()` gibi C++20'de tanıtılan yeni işlevler kullanılabilir. İşte bunların kısa açıklamaları ve örnek kullanımları:

Standart Giriş/Çıkış Akışları:

- **std::cin**: Standart input akışıdır. Kullanıcıdan veri almak için kullanılır.
- **std::cout**: Standart output akışıdır. Veriyi konsola yazdırmak için kullanılır.
- **std::cerr**: Standart hata çıkış akışıdır. Hata mesajlarını konsola yazdırmak için kullanılır.
- **std::clog**: Standart log çıkış akışıdır. Log mesajlarını konsola yazdırmak için kullanılır.

Örnek:

```
#include <iostream>

int main() {
    int number;

    // Kullanıcıdan bir sayı al
    std::cout << "Bir sayı girin: ";
    std::cin >> number;

    // Konsola sayıyı yazdır
    std::cout << "Girdiğiniz sayı: " << number << std::endl;

    // Hata mesajı yazdır
    std::cerr << "Bir hata oluştu!" << std::endl;
```

```
    // Günlük mesajı yazdır
    std::clog << "Bu bir log mesajıdır." << std::endl;

    return 0;
}
```

C++20'de Tanıtılan Yeni İşlevler:

- **std::print()**: C++20 ile tanıtılan bir işlevdir. Veriyi konsola yazdırmak için kullanılır.
- **std::println()**: C++20 ile tanıtılan bir işlevdir. Veriyi konsola yazdırmak için kullanılır ve sonuna yeni satır ekler.

Örnek:

```
#include <iostream>

int main() {
    int number = 42;

    // Veriyi konsola yazdır
    std::print("Number: {}\\n", number); // C++20'den önceki versiyonlarda
bu satır hata verebilir

    // Veriyi konsola yazdır ve yeni satır ekle
    std::println("Hello, world!");

    return 0;
}
```

Bu örneklerde, input/output işlemleri için standart input/output akışları veya C++20'de tanıtılan `std::print()` ve `std::println()` işlevleri kullanılarak basit input/output işlemleri gerçekleştirilmiştir.

Initializing variables

C++'ta değişkenlerin başlatılması veya ataması için farklı yöntemler vardır. Bu yöntemler arasında functional notation, assignment notation ve uniform initialization (curly braces) gibi farklı yaklaşımlar bulunmaktadır. İşte bu yöntemlerin açıklamaları ve örnekleri:

Functional Notation (Fonksiyonel Gösterim):

Bu yöntemde, değişkenlerin değerleri parantezler içinde bir fonksiyon çağrısı gibi belirtilir. Bu, genellikle C++'ta işlevsel programlamaya daha yakın durumlar için kullanılır.

Örnek:

```
int x = int(5);
double y = double(3.14);
```

Bu örnekte, `x` ve `y` değişkenleri, `int` ve `double` türlerine dönüştürme işlevi aracılığıyla başlatılır.

Assignment Notation (Atama Gösterimi):

Bu yöntemde, değişkenlerin değerleri `=` operatörü ile atanır.

Örnek:

```
int x;
x = 5;

double y;
y = 3.14;
```

Bu örnekte, `x` ve `y` değişkenleri daha önce tanımlanmıştır ve sonra değerleri atanır.

Uniform Initialization (Düzgün Başlatma):

Bu yöntemde, değişkenlerin değerleri süslü parantezler `{}` ile belirtilir. Bu yöntem C++11'den itibaren kullanılmıştır ve genellikle daha tutarlı bir başlatma yöntemi olarak kabul edilir.

Örnek:

```
int x{5};
double y{3.14};
```

Bu örnekte, `x` ve `y` değişkenleri süslü parantezler kullanılarak başlatılmıştır. Bu yöntem, tür çıkarımı, liste başlatma ve diğer bazı C++ dil özelliklerine uyum sağlar.

Bu üç yöntem de değişkenlerin başlatılması veya ataması için kullanılabilir. Ancak, genellikle kod tutarlılığı ve okunabilirliği için uniform initialization yöntemi tercih edilir. Bu yöntem, C++'ın modern özellikleriyle daha iyi uyum sağlar ve hata olasılığını azaltır.

Type deduction using the `auto` keyword

C++'ta `auto` anahtar kelimesi, tür çıkarımı (type inference) yapmak için kullanılır. Bu, bir değişkenin türünü otomatik olarak belirlemek için kullanılır. `auto` anahtar kelimesi, genellikle kodun daha kısa ve okunabilir olmasını sağlar. İşte bir örnek:

```
#include <iostream>
#include <vector>

int main() {
    // Bir integer değişkeni tanımla ve başlat
    auto x = 5;

    // Bir double değişkeni tanımla ve başlat
    auto y = 3.14;

    // Bir string değişkeni tanımla ve başlat
    auto name = "John";
```

```
    // Bir vektör tanımla ve başlat
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    auto size = numbers.size();

    // Değişkenlerin türlerini yazdır
    std::cout << "x'nin türü: " << typeid(x).name() << std::endl;
    std::cout << "y'nin türü: " << typeid(y).name() << std::endl;
    std::cout << "name'in türü: " << typeid(name).name() << std::endl;
    std::cout << "size'nin türü: " << typeid(size).name() << std::endl;

    return 0;
}
```

Yukarıdaki örnekte, `auto` anahtar kelimesi kullanılarak değişkenler tanımlanmış ve başlatılmıştır. C++ derleyicisi, değişkenin başlatıldığı değerden türünü otomatik olarak belirler. Bu, kodun daha esnek olmasını ve tür değişikliklerinden etkilenmemesini sağlar. Örneğin, `x` değişkeni başlangıçta bir `int` olarak başlatılmış olsa da, daha sonra `double` veya başka bir türe dönüştürülebilir ve kod değiştirmeye gerek kalmaz. Bu, kodun daha temiz ve bakımı daha kolay hale gelmesini sağlar.

The Lifetime and the Scope of a variable

C++ programında, bir değişkenin yaşam döngüsü (lifetime) ve kapsamı (scope) önemli kavramlardır.

Lifetime (Yaşam Döngüsü):

Bir değişkenin yaşam döngüsü, bellekte ne kadar süreyle mevcut olduğunu belirtir. C++ dilinde, değişkenlerin farklı yaşam döngüsü süreçleri bulunmaktadır:

- **Automatic Storage Duration (Otomatik Depolama Süresi)**: Otomatik olarak oluşturulan yerel değişkenlerin yaşam döngüsü, tanımlandığı blok veya fonksiyonun çalışma süresine bağlıdır. Bu tür değişkenler, bir fonksiyon çağrıldığında oluşturulur ve fonksiyonun sona ermesiyle yok olur.
- **Static Storage Duration (Statik Depolama Süresi)**: Statik olarak tanımlanan değişkenlerin yaşam döngüsü, programın çalışma süresi boyunca devam eder. Bu tür değişkenler, program başladığında oluşturulur ve program sona erdiğinde yok edilirler.
- **Dynamic Storage Duration (Dinamik Depolama Süresi)**: Dinamik olarak bellekte yer ayrılan değişkenlerin yaşam döngüsü, programcı tarafından belirlenir. Bu tür değişkenler, dinamik bellek tahsisi operatörleri (`new`, `delete`) kullanılarak oluşturulur ve programcı tarafından belirtilen zamana kadar bellekte kalır.
- **Thread Storage Duration (İş Parçacığı Depolama Süresi)**: Bu, çoklu iş parçacığı (thread) ortamlarında kullanılan değişkenlerin yaşam döngüsüdür. Bir iş parçacığına özgü oluşturulan bu değişkenler, ilgili iş parçacığının çalışma süresine bağlıdır.

Scope (Kapsam):

Bir değişkenin kapsamı, o değişkenin erişilebilir olduğu kod bloğunu veya alanı belirtir. C++ dilinde, değişkenlerin farklı kapsam türleri vardır:

- **Global Scope (Global Kapsam)**: Programın herhangi bir yerinden erişilebilen değişkenlerin kapsamıdır. Bu tür değişkenler, programın herhangi bir noktasında kullanılabilirler.
- **Local Scope (Yerel Kapsam)**: Bir blok veya fonksiyon içinde tanımlanan değişkenlerin kapsamıdır. Bu tür değişkenler, tanımlandıkları blok veya fonksiyon içinde geçerlidirler.
- **Class Scope (Sınıf Kapsamı)**: Bir sınıf içinde tanımlanan üye değişkenlerin kapsamıdır. Bu tür değişkenler, sınıfın herhangi bir üye işlevi içinde kullanılabilirler.

Örnek:

```
#include <iostream>

// Global scope
int globalVariable = 10;

void foo() {
    // Local scope
    int localVariable = 20;
    std::cout << "Local variable: " << localVariable << std::endl;
}

int main() {
    // Static storage duration
    static int staticVariable = 30;

    foo();

    // Dynamic storage duration
    int* dynamicVariable = new int(40);
    std::cout << "Dynamic variable: " << *dynamicVariable << std::endl;
    delete dynamicVariable;

    std::cout << "Global variable: " << globalVariable << std::endl;
    std::cout << "Static variable: " << staticVariable << std::endl;

    return 0;
}
```

Bu örnekte, farklı yaşam döngüsü ve kapsam türlerine sahip değişkenler tanımlanmıştır. Her değişkenin yaşam döngüsü ve kapsamı, tanımlandığı yere bağlı olarak belirlenir. Bu kavramlar, değişkenlerin nasıl oluşturulduğu, kullanıldığı ve bellekten serbest bırakıldığı hakkında önemli bilgiler sağlar.

> Global variablelar default olarak static storage duration'a sahiptirler.

Scope Resolution Operator (::)

C++'ta, "::" olarak ifade edilen "scope resolution operator" (kapsam çözümleme operatörü), bir ismin hangi kapsamda olduğunu belirtmek için kullanılır. Bu operatör, global alanlar, sınıflar, isim alanları gibi kapsamları belirlemek için kullanılır. Özellikle isim çakışmalarını çözmek için önemlidir.

Örneğin, bir sınıf içinde bir üye fonksiyon tanımlanırken, bu fonksiyonun sınıfın üye değişkenlerine veya başka bir sınıfın üye fonksiyonlarına erişimini sağlamak için "::" operatörü kullanılabilir. Ayrıca, global isim alanlarına veya global değişkenlere erişim sağlamak için de kullanılabilir.

Örnek kullanım:

```
#include <iostream>

int x = 10; // Global değişken

namespace A {
    int x = 20; // A isim alanında bir değişken
    void printX() {
        std::cout << "x in namespace A: " << x << std::endl;
    }
}

namespace B {
    int x = 30; // B isim alanında bir değişken
    void printX() {
        std::cout << "x in namespace B: " << x << std::endl;
    }
}

int main() {
    std::cout << "Global x: " << ::x << std::endl; // Global x'e erişim

    A::printX(); // A isim alanındaki printX fonksiyonunu çağırma
    B::printX(); // B isim alanındaki printX fonksiyonunu çağırma

    return 0;
}
```

Bu örnekte "::" operatörü, farklı isim alanlarına erişim sağlamak için kullanılmıştır. "A::printX()" ifadesi, "A" isim alanındaki "printX" fonksiyonunu çağırırken, "B::printX()" ifadesi "B" isim alanındaki "printX" fonksiyonunu çağırır. Ayrıca, "::x" ifadesi global "x" değişkenine erişim sağlar. Bu şekilde, "::" operatörü isim çakışmalarını çözmek için kullanılabilir.

Using the const keyword in the declaration of pointers

1. Data (pointer tarafından point edilen) constant ama pointer kendisi değişebilir.

```
const char *ptr = "ABC";
// or
const char *ptr = {"ABC"};
```

```
// or
char const * ptr {"ABC"};
// -------------------------
*ptr = "Z"; // Compiler error! Because data is constant
ptr++;      //OK, because the address in the pointer may change
```

2. Pointer constant, data değişebilir

```
int data {10};
int * const cp {&data}; // Pointer is const, data may change
*cp = 15;               // OK, data is not constant
cp++;                   // Compiler Error! Because the pointer is constant
```

3. Pointer ve data constant

```
const double data {1.2};
const double * const ccp {&data}; // Pointer and data const
*ccp = "2.3"; // Compiler error!
ccp++;        // Compiler error!


// The same pointer definition may also written as follows
double const * const ccp {&data};
```

Inline functions

Inline fonksiyonlar, C++ dilinde fonksiyon çağrılarının yerine, derleme aşamasında çağrıldıkları yere kodlarının yerleştirilmesini sağlayan özel bir optimizasyon tekniğidir. `inline` anahtar kelimesi ile işaretlenen fonksiyonlar, genellikle kısa ve sıkça çağrılan fonksiyonlar için tercih edilir. İşte inline fonksiyonların standard fonksiyonlardan farkı, avantajları ve dezavantajları:

Farklar:

- **Çağrı Yerine Kod Yerleştirme**: Inline fonksiyonlar, çağrıldıkları yere doğrudan kodlarının yerleştirilmesini sağlar. Bu, fonksiyon çağrısı yapmanın maliyetini azaltır ve kodun hızlı çalışmasını sağlar.
- **Derleme Aşamasında Değerlendirme**: Inline fonksiyonlar, derleme aşamasında değerlendirilir ve çağrıldıkları yere yerleştirilir. Bu, çalışma zamanında işlemci tarafından işlenmeleri gerekmeyen ekstra bir çağrı maliyetinden kurtulmayı sağlar.

Avantajlar:

- **Performans Artışı**: Inline fonksiyonlar, fonksiyon çağrılarının maliyetini azaltarak kodun daha hızlı çalışmasını sağlar.
- **Küçük Boyutlu Kod**: Inline fonksiyonlar, çağrıldıkları yere kodlarının yerleştirilmesi nedeniyle, kod boyutunu artırmadan tekrarlanan çağrılar için daha etkili bir çözüm sunar.

Dezavantajlar:

- **Kod Boyutu Artışı**: Fonksiyonun kodları her çağrıldığında yerleştirildiği için, sıkça çağrılan inline fonksiyonlar kod boyutunu artırabilir. Bu, programın bellek kullanımını artırabilir.
- **Derleme Süresi Artışı**: Inline fonksiyonlar, her çağrıldığında kodlarının yerleştirilmesi gerektiği için, derleme süresini artırabilir. Özellikle karmaşık inline fonksiyonlar, derleme süresini önemli ölçüde uzatabilir.

Örnek:

```
#include <iostream>

// Inline fonksiyon tanımı
inline int add(int x, int y) {
    return x + y;
}

int main() {
    int result = add(5, 3); // Inline fonksiyon çağrısı
    std::cout << "Result: " << result << std::endl;

    return 0;
}
```

Yukarıdaki örnekte, `add` fonksiyonu `inline` anahtar kelimesi ile tanımlanmıştır. Bu nedenle, `add` fonksiyonu çağrıldığında, derleme aşamasında kodunun çağrıldığı yere yerleştirilir. Bu, `add` fonksiyonunun performansını artırır ve kodun hızlı çalışmasını sağlar.

Default function arguments

C++'ta, varsayılan fonksiyon argümanları (default function arguments), bir fonksiyonun çağrıldığında belirtilmeyen argümanlar için varsayılan değerler belirlemek için kullanılır. Bu, fonksiyonların esnekliğini artırır ve fonksiyonların daha geniş bir kullanım alanına sahip olmasını sağlar. Varsayılan argümanlar, fonksiyon prototiplerinde belirtilir ve fonksiyonun tanımında değer atanmazsa, bu varsayılan değerler kullanılır.

Örnek:

```
#include <iostream>

// Fonksiyon prototipi, varsayılan argümanlar belirtiliyor
void greet(std::string name = "Guest", int age = 0) {
    std::cout << "Hello, " << name << "! You are " << age << " years old."
<< std::endl;
}

int main() {
    // Argüman belirtilmediği için varsayılan değerler kullanılacak
    greet(); // "Hello, Guest! You are 0 years old."
```

```
    // Sadece bir argüman belirtilirse, diğeri varsayılan değeri alacak
    greet("Alice"); // "Hello, Alice! You are 0 years old."

    // Her iki argüman da belirtilirse, belirtilen değerler kullanılacak
    greet("Bob", 30); // "Hello, Bob! You are 30 years old."

    return 0;
}
```

Yukarıdaki örnekte, `greet` fonksiyonunun prototipi varsayılan argümanlarla tanımlanmıştır. Fonksiyon çağrılarında argümanlar belirtilmezse, fonksiyon varsayılan değerlerle çağrılır. Ancak, çağrıda argümanlar belirtilirse, belirtilen değerler kullanılır. Bu, fonksiyonların esnek bir şekilde çağrılmasını sağlar ve kodun daha okunabilir olmasını sağlar.

Overloading of function names

Fonksiyonların aşırı yüklenmesi (function overloading), aynı isme sahip ancak farklı parametre listelerine sahip birden fazla fonksiyon tanımlanmasını sağlayan bir C++ dil özelliğidir. Bu, farklı türlerde veya farklı sayıda parametre alabilen işlevlerin tanımlanmasına olanak tanır. Programcılar aynı işlev adını kullanarak farklı işlevler tanımlayabilir ve işlev çağrıları yapabilir.

Örnek:

```
#include <iostream>

// İki tamsayı parametre alan bir fonksiyon
int add(int a, int b) {
    return a + b;
}

// İki double parametre alan bir fonksiyon
double add(double a, double b) {
    return a + b;
}

// Üç tamsayı parametre alan bir fonksiyon
int add(int a, int b, int c) {
    return a + b + c;
}

int main() {
    std::cout << "1 + 2 = " << add(1, 2) << std::endl; // İlk fonksiyon
    std::cout << "1.5 + 2.5 = " << add(1.5, 2.5) << std::endl; // İkinci fo
nksiyon
    std::cout << "1 + 2 + 3 = " << add(1, 2, 3) << std::endl; // Üçüncü fon
ksiyon
```

```
    return 0;
}
```

Yukarıdaki örnekte, `add` fonksiyonu farklı parametre listelerine sahip üç farklı sürümde tanımlanmıştır. Birinci sürüm iki tamsayı parametre alırken, ikinci sürüm iki double parametre alır ve üçüncü sürüm üç tamsayı parametre alır. Program, hangi fonksiyonun çağrılacağını argümanların türüne göre belirler. Bu sayede, aynı işlev adı kullanılarak farklı işlevler tanımlanabilir ve çağrılabilir, bu da kodun daha esnek ve okunabilir olmasını sağlar.

Reference operator (&)

C++'ta "&" (ampersand) işareti, bir değişkenin bellek adresini almak için kullanılan referans operatörüdür. Referanslar, bir değişkenin bir başka isimle erişilebilir hale getirilmesini sağlar. Bu, özellikle fonksiyonlara argüman olarak değişkenlerin bellek adreslerini geçirirken veya bir fonksiyondan geri döndürülürken kullanışlıdır.

Örnek:

```
#include <iostream>

int main() {
    int x = 10;
    int& ref = x; // "ref", "x" değişkenine bir referans

    std::cout << "x: " << x << std::endl; // "x" değeri
    std::cout << "ref: " << ref << std::endl; // "ref" değeri, "x" ile aynı
bellek adresini gösterir

    ref = 20; // "ref" üzerinden "x" değişkeninin değerini değiştirme

    std::cout << "x: " << x << std::endl; // "x" değeri, güncellendi
    std::cout << "ref: " << ref << std::endl; // "ref" değeri, "x" ile aynı
bellek adresini gösterir

    return 0;
}
```

Yukarıdaki örnekte, `int& ref = x;` ifadesiyle `ref` adında bir referans oluşturulmuş ve bu referans `x` değişkenine bağlanmıştır. Bu sayede, `ref` üzerinden `x` değişkenine erişilebilir ve `x` değişkeninin değeri değiştirilebilir. Referanslar, bir değişkenin bir başka isimle erişilmesini sağlar, ancak referansın kendisi bir değişken değil, mevcut bir değişkenin adresini gösteren bir işarettir.

Call by reference

"Call by address" ve "call by reference" (referans ile çağrı), her ikisi de bir fonksiyona argüman olarak bir değişkenin adresini geçmenin bir yoludur. Ancak, bu iki yaklaşım arasında bazı önemli farklar vardır.

Call by Address (Adrese Göre Çağrı):

Bu yaklaşımda, fonksiyona bir değişkenin bellek adresi geçilir. Fonksiyon, bu adresi kullanarak doğrudan değişkenin değerini değiştirebilir. Adrese göre çağrıda, genellikle işaretçiler (pointers) kullanılır.

Örnek:

```
#include <iostream>

// Fonksiyon, bir işaretçi parametresi alır
void changeValue(int* ptr) {
    *ptr = 20; // İşaretçinin işaret ettiği değeri değiştir
}

int main() {
    int x = 10;

    std::cout << "Before change: " << x << std::endl;

    // Fonksiyona değişkenin adresini geç
    changeValue(&x);

    std::cout << "After change: " << x << std::endl;

    return 0;
}
```

Call by Reference (Referansa Göre Çağrı):

Bu yaklaşımda, fonksiyona bir değişkenin kendisi değil, değişkenin referansı (bir alias) geçilir. Fonksiyon, bu referansı kullanarak doğrudan değişkenin değerini değiştirebilir. Call by reference kullanılarak, kod daha temiz ve anlaşılır hale gelebilir.

Örnek:

```
#include <iostream>

// Fonksiyon, bir referans parametresi alır
void changeValue(int& ref) {
    ref = 20; // Referansın işaret ettiği değeri değiştir
}

int main() {
    int x = 10;

    std::cout << "Before change: " << x << std::endl;

    // Fonksiyona değişkenin referansını geç
    changeValue(x);
```

```
    std::cout << "After change: " << x << std::endl;

    return 0;
}
```

Her iki yaklaşım da bir fonksiyona bir değişkenin değerinin değiştirilmesi için bir yol sağlar. Ancak, call by reference kullanarak, kod daha okunabilir ve anlaşılır hale gelir ve işaretçilere (pointers) göre daha az hata eğilimindedir.

Örnek 2:

Büyük veri yapılarını (örneğin, büyük diziler veya nesneler) fonksiyonlara parametre olarak geçirirken, bu veri yapılarını stack belleğine kopyalamak yerine adres veya referans kullanmak önemli bir optimizasyon sağlayabilir. Çünkü büyük veri yapıları stack belleğinde yer kaplar ve bu, bellek kullanımını artırabilir ve programın performansını düşürebilir.

Adres veya referans kullanarak veri yapılarını fonksiyonlara geçirirsek, aslında veri yapılarının kendisi stack belleğine kopyalanmaz, sadece veri yapılarının bellek adresi geçilir. Bu, hem bellek kullanımını azaltır hem de işlemcinin veri yapılarını kopyalamak için harcadığı zamanı azaltır.

Örnek:

```
#include <iostream>
#include <vector>

// Büyük bir vektörü adres ile fonksiyona geçme
void processVector(const std::vector<int>& vec) {
    // Vektör elemanlarını işle
    for (int num : vec) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}

int main() {
    // Büyük bir vektör oluştur
    std::vector<int> bigVector(1000000, 1); // 1 milyon elemanı olan vektör

    // Fonksiyona büyük vektörün adresini geç
    processVector(bigVector);

    return 0;
}
```

Yukarıdaki örnekte, `processVector` fonksiyonu, büyük bir vektörün adresini alır ve bu vektörü işler. Bu şekilde, `bigVector` vektörünün tüm elemanlarını fonksiyona kopyalamak yerine, sadece vektörün bellek adresi geçilir. Bu, bellek kullanımını azaltır ve programın performansını artırır.

Return by reference

C++'ta "return by reference" (referans ile dönüş), bir fonksiyonun çağrıldığında oluşturulan bir geçici nesne yerine, bir değerin veya nesnenin doğrudan kendisinin döndürülmesini sağlar. Bu, fonksiyonun çağrıldığı yerde bir nesneye referans oluşturulmasına ve bu nesnenin değerinin doğrudan değiştirilmesine olanak tanır.

Return by reference genellikle büyük veri yapılarını (örneğin, büyük diziler veya nesneler) verimli bir şekilde döndürmek için kullanılır. Bu sayede, fonksiyonun çağrıldığı yerde bir kopyalama işlemi yapmadan, doğrudan orijinal veri yapısına erişim sağlanabilir.

Örnek:

```
#include <iostream>
#include <vector>

// Büyük bir vektörü referans ile döndürme
std::vector<int>& createVector() {
    // Büyük bir vektör oluştur
    static std::vector<int> bigVector = {1, 2, 3, 4, 5};

    // Oluşturulan vektörün referansını döndür
    return bigVector;
}

int main() {
    // Fonksiyondan vektör referansını al
    std::vector<int>& returnedVector = createVector();

    // Vektörü değiştir
    returnedVector.push_back(6);

    // Vektörü yazdır
    for (int num : createVector()) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Yukarıdaki örnekte, `createVector` fonksiyonu, büyük bir vektör oluşturur ve bu vektörün referansını döndürür. `main` fonksiyonunda, `createVector` fonksiyonundan döndürülen referans alınır ve bu referans üzerinden vektör değiştirilir. Böylece, fonksiyonun çağrıldığı yerde büyük bir vektör oluşturmak yerine, mevcut bir vektörün referansı döndürülür. Bu, bellek kullanımını azaltır ve performansı artırır.

Operator overloading

Operator overloading, C++ dilinde bir operatörün farklı türlerdeki veya farklı amaçlardaki kullanımlarını aynı isimdeki işlevlerle yeniden tanımlama işlemidir. Bu sayede, programcılar kendi

sınıf ve veri tiplerini oluştururken, o sınıf veya veri tipi üzerinde standart C++ operatörlerini (örneğin, +, -, *, / gibi) kullanabilirler.

Operator overloading, sınıflar arasında doğal ve tutarlı bir etkileşim sağlar. Örneğin, iki vektör nesnesinin toplanması, bir dize nesnesinin bir başka dize nesnesine eklenmesi gibi işlemler operatör overloading kullanılarak tanımlanabilir.

Örnek:

```
#include <iostream>

// Bir vektör sınıfı
class Vector {
private:
    int x, y;

public:
    Vector(int x, int y) : x(x), y(y) {}

    // + operatörü overloading'i
    Vector operator+(const Vector& other) const {
        return Vector(x + other.x, y + other.y);
    }

    // << operatörü overloading'i (stream operatörü)
    friend std::ostream& operator<<(std::ostream& os, const Vector& vec) {
        os << "(" << vec.x << ", " << vec.y << ")";
        return os;
    }
};

int main() {
    Vector v1(1, 2);
    Vector v2(3, 4);

    Vector result = v1 + v2; // + operatörü overloading'i
    std::cout << "Result: " << result << std::endl; // << operatörü overloa
ding'i

    return 0;
}
```

Yukarıdaki örnekte, Vector sınıfında + operatörü ve << operatörü overloading yapılmıştır. Böylece, iki vektör nesnesi toplandığında (+ operatörü), veya vektör nesnesi ekrana yazdırıldığında (<< operatörü), uygun işlemler yapılarak istenen sonuçlar elde edilir. Bu sayede, kullanıcılar kendi sınıflarında standart C++ operatörlerini kullanabilirler ve daha okunabilir ve tutarlı kodlar yazabilirler.

OOP03

Classes and Objects

"Class" ve "object" (veya "instance"), nesne yönelimli programlamada temel kavramlardır.

Class (Sınıf):

- Bir veri türü ve bu veri türüyle ilişkili veri ve işlevleri tanımlayan bir yapıdır.
- Bir şablon gibi davranır ve aynı sınıftan birden çok nesne (object) oluşturulabilir.
- Veri üyeleri (örneğin, değişkenler) ve fonksiyon üyeleri (metodlar) içerebilir.
- Bir sınıf, belirli bir türde nesnelerin (örneğin, köpekler, arabalar, öğrenciler gibi) genel özelliklerini ve davranışlarını tanımlar.

Örnek bir sınıf tanımı:

```
class Rectangle {
private:
    int width;
    int height;

public:
    // Constructor (kurucu)
    Rectangle(int w, int h) : width(w), height(h) {}

    // Alanı hesapla fonksiyonu
    int calculateArea() {
        return width * height;
    }
};
```

Object (Nesne):

- Sınıfın bir örneğidir ve sınıfta tanımlanan veri ve işlevlerin bir kopyasını içerir.
- Bir sınıfın özelliklerini ve davranışlarını gerçek dünyadan bir varlık olarak temsil eder.
- Bir sınıfın bir veya daha fazla örneğini oluşturarak kullanılır.

Örnek bir nesne oluşturma:

```
int main() {
    // Rectangle sınıfından iki nesne (object) oluşturma
    Rectangle rect1(5, 10); // width: 5, height: 10
    Rectangle rect2(3, 7);  // width: 3, height: 7

    // rect1 ve rect2 nesnelerinin alanlarını hesaplayıp ekrana yazdırma
    std::cout << "Area of rect1: " << rect1.calculateArea() << std::endl;
    std::cout << "Area of rect2: " << rect2.calculateArea() << std::endl;
```

```
    return 0;
}
```

Yukarıdaki örnekte, `Rectangle` sınıfından `rect1` ve `rect2` isimli iki nesne oluşturulmuştur. Her bir nesne, `Rectangle` sınıfının `width` ve `height` veri üyelerini içerir ve `calculateArea` fonksiyonunu kullanarak alanlarını hesaplar. Bu şekilde, sınıf tanımı (class) genel özellikleri ve davranışları tanımlarken, nesneler (objects) bu özellikleri ve davranışları temsil eder ve kullanır.

Defining methods as inline function

Bir fonksiyonu sınıf tanımı içinde tanımlarsanız, bu fonksiyon genellikle otomatik olarak bir "inline" fonksiyon olarak işaretlenir. Bunun nedeni, sınıf tanımı içinde tanımlanan fonksiyonların genellikle kısa ve sıkça çağrılan işlevler olmasıdır ve bu işlevlerin inline olarak ele alınması, çağrı maliyetini azaltabilir ve performansı artırabilir.

Örnek:

```
#include <iostream>

// Bir sınıf tanımı
class MyClass {
public:
    // Sınıf içinde bir fonksiyonun tanımı
    void inlineFunction() {
        std::cout << "This is an inline function." << std::endl;
    }
};

int main() {
    MyClass obj;
    obj.inlineFunction(); // Inline fonksiyonun çağrılması

    return 0;
}
```

Yukarıdaki örnekte, `inlineFunction` fonksiyonu `MyClass` sınıfının içinde tanımlanmıştır. Bu fonksiyon, kısa ve basit olduğu için inline olarak işaretlenebilir. Ancak, derleyici bu işareti dikkate almayabilir ve inline olarak işaretlemeyebilir. Bu nedenle, inline işaretinin sadece bir öneri olduğunu ve derleyicinin bu işareti dikkate alıp almayacağını belirlediğini unutmamak önemlidir.

Defining dynamic objects

Dinamik nesneler, program çalışma zamanında oluşturulan ve bellekte dinamik olarak tahsis edilen nesnelerdir. C++ dilinde dinamik nesnelerin oluşturulması genellikle `new` anahtar kelimesi ile yapılır ve bu nesnelerin bellekten silinmesi için `delete` anahtar kelimesi kullanılır.

Bir class ve pointer kullanarak dinamik bir nesne oluşturma örneği:

```
#include <iostream>

// Bir sınıf tanımı
class MyClass {
public:
    void printMessage() {
        std::cout << "Hello, this is a dynamic object!" << std::endl;
    }
};

int main() {
    // MyClass türünde bir işaretçi (pointer) oluşturma
    MyClass* ptr = new MyClass(); // ya da MyClass* ptr = new MyClass;
    // ya da
    MyClass *ptr1;
    ptr1 = new MyClass;
    // ya da
    MyClass *ptr2 {new MyClass};

    // Dinamik olarak oluşturulan nesnenin bir üyesini çağırma
    ptr->printMessage();

    // Dinamik olarak oluşturulan nesneyi bellekten silme
    delete ptr;

    return 0;
}
```

Yukarıdaki örnekte, `MyClass` türünden bir işaretçi (`ptr`) oluşturulmuş ve bu işaretçi `new` anahtar kelimesi ile dinamik olarak bir `MyClass` nesnesi ile ilişkilendirilmiştir. `ptr` üzerinden dinamik olarak oluşturulan nesnenin `printMessage` fonksiyonu çağrılarak bir mesaj yazdırılmıştır. Daha sonra, dinamik olarak oluşturulan nesne `delete` anahtar kelimesi kullanılarak bellekten silinmiştir.

Dinamik nesnelerin kullanımı, ihtiyaç duyulan bellek miktarı çalışma zamanında belirsiz olduğunda veya nesnenin yaşam döngüsünün çalışma zamanında belirlenmesi gerektiğinde faydalıdır. Bununla birlikte, dinamik bellek tahsisi ve serbest bırakma dikkatlice yapılmalı ve bellek sızıntılarından kaçınılmalıdır.

Defining Arrays of Objects

C++ dilinde, nesnelerin dizilerini oluşturabiliriz. Bu, bir sınıf türünden birden fazla nesneyi gruplamamıza ve daha kolay erişim sağlamamıza olanak tanır.

Örnek:

```
#include <iostream>

// Örnek bir sınıf tanımı: Point
```

```
class Point {
private:
    int x, y;

public:
    // Kurucu fonksiyon
    Point(int x_val, int y_val) : x(x_val), y(y_val) {}

    // Koordinatları yazdırma fonksiyonu
    void printCoordinates() {
        std::cout << "x: " << x << ", y: " << y << std::endl;
    }
};

int main() {
    // Point türünden bir nesne dizisi oluşturma
    Point points[3] = {
        Point(1, 2),
        Point(3, 4),
        Point(5, 6)
    };

    // Dizideki tüm noktaların koordinatlarını yazdırma
    for (int i = 0; i < 3; ++i) {
        points[i].printCoordinates();
    }

    return 0;
}
```

Yukarıdaki örnekte, `Point` adında bir sınıf tanımladık. Daha sonra `main` fonksiyonunda, `Point` sınıfının bir nesne dizisi olan `points` dizisini oluşturduk. Dizi, üç elemana sahiptir ve her bir elemanı bir `Point` nesnesidir. Daha sonra, döngü kullanarak dizideki her noktanın koordinatlarını yazdırdık.

Bu şekilde, sınıfların dizilerini kullanarak benzer nesneleri gruplayabilir ve daha organize bir şekilde yönetebiliriz.

Access Specifiers

C++'da erişim belirleyicileri (access specifiers), bir sınıfın üyelerinin (değişkenlerin veya fonksiyonların) diğer kod blokları tarafından erişimini kontrol etmek için kullanılır. Üç ana erişim belirleyicisi bulunur: `public`, `private` ve `protected`.

- **Public (Genel)**: `public` belirleyici ile tanımlanan üyeler, sınıf dışından erişilebilirler. Yani, bu üyeler herkes tarafından kullanılabilir ve değiştirilebilir.

Örnek:

```
class MyClass {
public:
    int publicVar; // Public değişken
    void publicFunction() { // Public fonksiyon
        // Fonksiyon kodu
    }
};
```

- **Private (Özel)**: `private` belirleyici ile tanımlanan üyeler, sadece tanımlandıkları sınıf içinden erişilebilirler. Yani, bu üyeler sadece sınıf içindeki fonksiyonlar tarafından kullanılabilir ve değiştirilebilir.

Örnek:

```
class MyClass {
private:
    int privateVar; // Private değişken
    void privateFunction() { // Private fonksiyon
        // Fonksiyon kodu
    }
};
```

- **Protected (Korumalı)**: `protected` belirleyici ile tanımlanan üyeler, hem tanımlandıkları sınıfın içinden hem de bu sınıftan türetilmiş alt sınıfların içinden erişilebilirler. Yani, bu üyeler, sadece türetilmiş sınıflar tarafından kullanılabilir ve değiştirilebilir.

Örnek:

```
class MyBaseClass {
protected:
    int protectedVar; // Protected değişken
    void protectedFunction() { // Protected fonksiyon
        // Fonksiyon kodu
    }
};

class MyDerivedClass : public MyBaseClass {
public:
    void accessProtectedMember() {
        protectedVar = 10; // MyBaseClass'ta tanımlanan protected değişkene
erişim
        protectedFunction(); // MyBaseClass'ta tanımlanan protected fonksiy
ona erişim
    }
};
```

Bu erişim belirleyicileri, sınıfların verilerini ve fonksiyonlarını kontrol altında tutarak, sınıfın kapsülleme (encapsulation) prensibine uygun şekilde davranmasını sağlar. Bu sayede, kodun

güvenliği artar ve kodun daha organize olmasına yardımcı olur.

> Class'ların default access modu private'tır.

struct keyword

`struct` anahtar kelimesi, C++ dilinde bir veri yapısı tanımlamak için kullanılan bir yapıdır. `struct` ile tanımlanan bir veri yapısı, `class` ile tanımlanan bir sınıfa oldukça benzer, ancak bazı temel farklılıklar vardır.

Struct Tanımı:

```
#include <iostream>

// Bir struct tanımı
struct Point {
    int x;
    int y;
};

int main() {
    // Struct türünden bir değişken tanımı ve kullanımı
    Point p1;
    p1.x = 10;
    p1.y = 20;

    std::cout << "x: " << p1.x << ", y: " << p1.y << std::endl;

    return 0;
}
```

Struct ve Class Arasındaki Farklar ve Benzerlikler:

Benzerlikler:

1. Hem `struct` hem de `class`, veri üyeleri ve fonksiyon üyelerini içerebilir.
2. Her ikisi de nesne yönelimli programlama (OOP) prensiplerini destekler.

Farklar:

1. **Varsayılan Erişim:**
   - `struct` tanımlarında varsayılan erişim belirleyicisi `public`'tir.
   - `class` tanımlarında varsayılan erişim belirleyicisi `private`'tır.
2. **Kalıtım:**
   - `struct` türü, kalıtımın varlığını belirtmez. Ancak, C++ dilinde `struct` tanımlarında da kalıtım kullanılabilir.
   - `class` türü, kalıtımı varsayılan olarak kabul eder.

3. **Sınıf ve Yapı Olarak Kullanım:**
    - `struct` genellikle basit veri yapılarını temsil etmek için kullanılır.
    - `class`, daha karmaşık yapılar ve işlevsellik gerektiren durumlar için kullanılır.

Genel olarak, `struct` ve `class` arasındaki temel fark, `class`'ın daha gelişmiş özelliklere ve işlevselliğe sahip olmasıdır. Ancak, C++ dilinde `struct` kullanarak da nesne yönelimli programlama prensiplerini uygulamak mümkündür.

Accessors and Mutators

Accessor (setter) ve mutatorlar (getter), bir sınıfın özel veri üyelerine erişmek veya bu üyeleri değiştirmek için kullanılan fonksiyonlardır. Bu, veri gizleme (encapsulation) prensibini uygulayarak sınıfın iç verilerinin dışarıdan doğrudan erişilmesini önler ve sınıfın iç yapısının güvenliğini sağlar. Accessorlar genellikle veri üyelerine okuma erişimi sağlarken, mutatorlar veri üyelerini güncellemek için kullanılır.

Örnek:

```
#include <iostream>

class Rectangle {
private:
    int width;
    int height;

public:
    // Constructor (kurucu)
    Rectangle(int w, int h) : width(w), height(h) {}

    // Getter (Accessor) fonksiyonlar
    int getWidth() const {
        return width;
    }

    int getHeight() const {
        return height;
    }

    // Setter (Mutator) fonksiyonlar
    void setWidth(int w) {
        width = w;
    }

    void setHeight(int h) {
        height = h;
    }
};
```

```
int main() {
    Rectangle rect(10, 20);

    // Accessorlar ile veriye erişim
    std::cout << "Width: " << rect.getWidth() << std::endl;
    std::cout << "Height: " << rect.getHeight() << std::endl;

    // Mutatorlar ile veriye güncelleme
    rect.setWidth(30);
    rect.setHeight(40);

    // Güncellenmiş veriyi Accessorlar ile erişim
    std::cout << "Updated Width: " << rect.getWidth() << std::endl;
    std::cout << "Updated Height: " << rect.getHeight() << std::endl;

    return 0;
}
```

Yukarıdaki örnekte, `Rectangle` sınıfında `getWidth` ve `getHeight` accessor fonksiyonları, `setWidth` ve `setHeight` mutator fonksiyonları tanımlanmıştır. Bu fonksiyonlar, sınıfın özel veri üyelerine erişim ve bu üyelerin güncellenmesi için kullanılır. Bu sayede, sınıfın iç verileri doğrudan erişilmez ve güvenlik sağlanır.

Friend functions and friend classes

`friend` anahtar kelimesi, C++ dilinde bir sınıf veya işlevin diğer sınıf veya işlevlere özel erişim izni vermesini sağlar. Bu, bir sınıfın veya işlevin private veya protected üyelerine diğer sınıfların veya işlevlerin doğrudan erişim sağlamasını mümkün kılar. `friend` anahtar kelimesi, sınıfın veya işlevin bildiriminde kullanılır ve bu nesnenin veya işlevin `friend` olarak belirtilen diğer nesnelerle ilişkili olmasını sağlar.

Friend Functions:

```
#include <iostream>

// MyClass sınıfının tanımı
class MyClass {
private:
    int data;

public:
    MyClass(int d) : data(d) {}

    // friend işlevin bildirimi
    friend void printData(const MyClass& obj);
};

// friend işlevin tanımı
```

```
void printData(const MyClass& obj) {
    std::cout << "Data: " << obj.data << std::endl; // MyClass'ın private v
erisine erişim
}

int main() {
    MyClass obj(10);
    printData(obj); // friend işlev çağrısı
    return 0;
}
```

Yukarıdaki örnekte, `printData` işlevi, `MyClass` sınıfının bir üyesi olmamakla birlikte, `MyClass` sınıfının `friend` işlevi olarak bildirilmiştir. Bu nedenle, `printData` işlevi, `MyClass` sınıfının private verilerine doğrudan erişebilir.

Friend Classes:

```
#include <iostream>

// MyClass sınıfının tanımı
class MyClass {
private:
    int data;

public:
    MyClass(int d) : data(d) {}

    // Arkadaş sınıfın bildirimi
    friend class FriendClass;
};

// Arkadaş sınıfın tanımı
class FriendClass {
public:
    void printData(const MyClass& obj) {
        std::cout << "Data: " << obj.data << std::endl; // MyClass'ın priva
te verisine erişim
    }
};

int main() {
    MyClass obj(10);
    FriendClass fc;
    fc.printData(obj); // Arkadaş sınıfın üye fonksiyonu çağrısı
    return 0;
}
```

Yukarıdaki örnekte, `FriendClass` sınıfı, `MyClass` sınıfını `friend` olarak tanımlamıştır. Bu nedenle, `FriendClass` sınıfı `MyClass` sınıfının private verilerine doğrudan erişebilir. Bu, `FriendClass` sınıfının `printData` üye fonksiyonunun içinde gösterilmiştir.

> Sınıflar arasındaki friend özelliği geçişli (transitive) değildir; A sınıfının B sınıfının friend'i ve B sınıfının C sınıfının friend'i olması, A sınıfının C sınıfının friend'i olduğu anlamına gelmez.

OOP04

Initializing Class Objects: Constructors

C++ dilinde, bir sınıfın constructorları, nesne oluşturulduğunda çağrılan özel üye fonksiyonlardır. Constructorlar, sınıfın başlatılmasını ve ilgili veri üyelerinin başlatılmasını sağlar.

1. **Default Constructor**:
   Tüm argümanlarını varsayılan olarak belirleyen veya hiçbir argüman gerektirmeyen bir constructor, yani hiçbir argüman olmadan çağrılabilen bir constructor'dır. Bu, sınıfın herhangi bir nesnesi oluşturulduğunda çağrılan ve hiçbir argüman almayan bir constructor'dır.

Örnek:

```
#include <iostream>

class MyClass {
public:
    // Default Constructor
    MyClass();
    // or
    MyClass() {}; // Default constructor with empty body
};

int main() {
    // MyClass tipinden bir nesne oluşturulduğunda default constructor çağr
ılır.
    MyClass obj;
    MyClass *ptr; // ptr bir obje olmadığı için default constructor çağırıl
maz.
    ptr = new MyClass; // Obje oluşturulduğu için default constructor çağır
ılır.
    return 0;
}
```

Yukarıdaki örnekte, `MyClass` adında bir sınıf tanımlanmıştır. Bu sınıf için constructor, `MyClass()` şeklinde tanımlanmıştır ve hiçbir argüman almaz.

- **Default Default Constructor**:
  Eğer sınıfın hiçbir özel veya varsayılan constructor tanımlanmamışsa ve bu constructor çağrılmamışsa, C++ derleyicisi otomatik olarak varsayılan bir nesne oluşturur. Bu, sınıfın herhangi bir nesnesi için çağrılan bir constructor'dır ve hiçbir argüman almaz. Bu, C++'ın varsayılan davranışıdır.

Örnek:

```
#include <iostream>

class MyClass {
public:

};

int main() {
    // MyClass tipinden bir nesne oluşturulmadan önce derleyici tarafından
    // varsayılan default constructor çağrılır.
    return 0;
}
```

Yukarıdaki örnekte, `MyClass` adında bir sınıf tanımlanmıştır ancak herhangi bir nesne oluşturulmadığı için herhangi bir constructor çağrılmamıştır. Bu durumda, derleyici otomatik olarak varsayılan default constructor'ı çağırır. Bu sınıfın hiçbir veri üyesi olmasa bile bu davranış geçerlidir.

2. **Constructors with parameters**:

   Bir sınıfın constructorları, nesne oluşturulduğunda çağrılan özel üye fonksiyonlardır. Constructorlar, sınıfın başlatılmasını ve ilgili veri üyelerinin başlatılmasını sağlar.

   Örnek:

```
#include <iostream>

class Rectangle {
private:
    int width;
    int height;

public:
    // Constructor
    Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    // Fonksiyon: Alanı hesapla
    int area() {
        return width * height;
```

```
    }
};

int main() {
    // Rectangle tipinden bir nesne oluşturulduğunda,
    // parametreleri alan constructor çağrılır.
    Rectangle r1(4, 5);
    std::cout << "Alan: " << r1.area() << std::endl;

    return 0;
}
```

Yukarıdaki örnekte, `Rectangle` adında bir sınıf tanımlanmıştır. `Rectangle` sınıfının bir constructor'ı var (`Rectangle(int w, int h)`). Bu constructor, bir dikdörtgenin genişliği ve yüksekliği gibi parametreleri alır ve bu değerleri sınıfın özel veri üyelerine atar.

Ancak, bu örnekte bir default constructor yok. Yani, eğer biz herhangi bir parametre vermeden bir `Rectangle` nesnesi oluştururursak, derleyici bir hata verecektir. Bu durumu düzeltmek için bir default constructor ekleyebiliriz:

```
#include <iostream>

class Rectangle {
private:
    int width;
    int height;

public:
    // Default Constructor
    Rectangle() {
        width = 0;
        height = 0;
    }

    // Constructor
    Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    // Fonksiyon: Alanı hesapla
    int area() {
        return width * height;
    }
};

int main() {
    // Parametre almayan constructor çağrılır.
    Rectangle r1;
```

```
    std::cout << "Alan: " << r1.area() << std::endl;

    // Parametreleri alan constructor çağrılır.
    Rectangle r2(4, 5);
    std::cout << "Alan: " << r2.area() << std::endl;

    return 0;
}
```

Bu kez, `Rectangle` sınıfına bir default constructor (`Rectangle()`) ekledik. Bu constructor, herhangi bir parametre almaz ve genişlik ve yükseklik değerlerini sıfıra ayarlar. Böylece, herhangi bir parametre belirtmeden bir `Rectangle` nesnesi oluşturabiliriz.

Multiple constructors

C++ dilinde bir sınıfın birden fazla constructor'a sahip olması mümkündür. Bu, aynı sınıf içinde farklı parametre listeleriyle birden fazla constructor tanımlanması anlamına gelir. Böylece, farklı durumlar için farklı constructor'ları kullanabiliriz.

Örnek:

```
#include <iostream>

class Rectangle {
private:
    int width;
    int height;

public:
    // Parametreleri alan constructor
    Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    // Default constructor
    Rectangle() {
        width = 0;
        height = 0;
    }

    // Bir parametre alan constructor
    Rectangle(int side) {
        width = side;
        height = side;
    }

    // Alanı hesaplayan fonksiyon
```

```
    int area() {
        return width * height;
    }
};

int main() {
    // Farklı constructor'ları kullanarak Rectangle nesneleri oluşturulur.
    Rectangle r1(4, 5);       // Genişlik: 4, Yükseklik: 5
    Rectangle r2;             // Default constructor kullanılır, Genişlik:
0, Yükseklik: 0
    Rectangle r3(6);          // Bir parametreli constructor kullanılır, Ka
re olur, Genişlik: 6, Yükseklik: 6

    // Alanları hesaplanır ve ekrana yazdırılır.
    std::cout << "Alan r1: " << r1.area() << std::endl;
    std::cout << "Alan r2: " << r2.area() << std::endl;
    std::cout << "Alan r3: " << r3.area() << std::endl;

    return 0;
}
```

Yukarıdaki örnekte, `Rectangle` sınıfı üç farklı constructor ile tanımlanmıştır:

1. `Rectangle(int w, int h)`: Genişlik ve yükseklik parametreleri alır.
2. `Rectangle()`: Parametre almaz, default constructor.
3. `Rectangle(int side)`: Tek bir parametre alır ve bu parametreyi hem genişlik hem de yükseklik olarak kabul eder.

Bu sayede farklı senaryolara göre uygun constructor kullanılarak nesneler oluşturulabilir.

Defining a default constructor using the `default` keyword

C++11'den itibaren, bir sınıfın default constructor'ını tanımlamak için `default` anahtar kelimesini kullanabiliriz. Bu, özellikle bir sınıfın bazı özel fonksiyonlarını varsayılan C++ davranışına geri döndürmek istediğimiz durumlarda kullanışlıdır. Örneğin, bir sınıfın özel bir kopya constructor'ı veya atama operatörü tanımladıktan sonra, sınıfın varsayılan kopya constructor'ını veya atama operatörünü kullanmak istiyorsak `default` anahtar kelimesini kullanabiliriz.

Örnek:

```
#include <iostream>

class MyClass {
private:
    int data;

public:
    // Özel bir constructor tanımladık
    MyClass(int d) : data(d) {}
```

```
    // Default constructor'u `default` anahtar kelimesi ile tanımladık
    MyClass() = default;

    // Fonksiyon: Veri değerini getir
    int getData() const {
        return data;
    }
};

int main() {
    // Default constructor çağrılır
    MyClass obj1;
    std::cout << "obj1 verisi: " << obj1.getData() << std::endl;

    // Özel constructor çağrılır
    MyClass obj2(42);
    std::cout << "obj2 verisi: " << obj2.getData() << std::endl;

    return 0;
}
```

Yukarıdaki örnekte, `MyClass` adında bir sınıf tanımladık. İlk olarak, `int` türünde bir `data` veri üyesi tanımladık. Daha sonra, özel bir constructor tanımladık (`MyClass(int d)`), bu constructor bir tamsayı alır ve `data` üyesine bu değeri atar. Ardından, `default` anahtar kelimesini kullanarak default constructor'ı tanımladık. Bu, sınıfın özel bir constructor'ı olmasına rağmen, hala default constructor'ı kullanabiliriz.

`default` anahtar kelimesi, sınıfın varsayılan davranışına geri dönmek istediğimizde veya özel bir constructor tanımladıktan sonra diğer özel fonksiyonları kullanmak istediğimizde oldukça kullanışlıdır.

Default arguments for constructor parameters

C++ dilinde, bir constructor'a varsayılan argümanlar atamak mümkündür. Bu, bir constructor çağrıldığında belirli parametrelerin sağlanmaması durumunda kullanılacak değerlerin önceden tanımlanmasını sağlar. Bu özellik, sınıf kullanımını esnek hale getirir.

Örnek:

```
#include <iostream>

class Rectangle {
private:
    int width;
    int height;

public:
    // Constructor with default arguments
```

```
    Rectangle(int w = 0, int h = 0) {
        width = w;
        height = h;
    }

    // Function to calculate area
    int area() {
        return width * height;
    }
};

int main() {
    // Creating objects using constructor with default arguments
    Rectangle r1;             // width = 0, height = 0
    Rectangle r2(4);          // width = 4, height = 0
    Rectangle r3(4, 5);       // width = 4, height = 5

    // Displaying areas
    std::cout << "Area of r1: " << r1.area() << std::endl;
    std::cout << "Area of r2: " << r2.area() << std::endl;
    std::cout << "Area of r3: " << r3.area() << std::endl;

    return 0;
}
```

Yukarıdaki örnekte, `Rectangle` sınıfının constructor'ı iki tamsayı parametresi alır: genişlik ve yükseklik. Ancak, bu parametreler için varsayılan değerler (`0`) belirttik. Böylece, eğer bir nesne oluştururken herhangi bir parametre belirtmezsek, derleyici varsayılan değerleri kullanacaktır.

Örneğin, `Rectangle r1;` çağrısında herhangi bir argüman belirtilmediği için varsayılan değerler kullanılır ve `width` ve `height` sıfıra atanır. Benzer şekilde, `Rectangle r2(4);` çağrısında `width` 4 olarak atanırken, `height` varsayılan değeri olan 0 olarak atanır. Son olarak, `Rectangle r3(4, 5);` çağrısında hem genişlik hem de yükseklik belirtilir ve bu değerler atanır. Bu şekilde, constructor parametrelerine varsayılan argümanlar atayarak sınıfın kullanımını daha esnek hale getirebiliriz.

Initializing arrays of objects

Bir nesne dizisi (array of objects) oluşturulduğunda, dizinin her öğesi (nesnesi) için varsa sınıfın default constructor'ı bir kez çağrılır.

```
Point pointArray[10]; // Default constructor is called 10 times
```

Argümanlarla bir constructor çağırmak için başlangıç değerlerinin bir listesi kullanılmalıdır.

```
Point (int = 0, int = 0) // can be called with zero, one or two arguments
```

```
// We do not provide the number of elements
Point pointArray[] = {10, 20, {30,40}}; // Array with 3 objects
```

```
// or to make program more readable
Point array[] = {Point {10}, Point {20}, Point {30,40}};
```

Point tipinde 3 tane obje oluşturuldu ve constructor 3 kez farklı argümanlarla çağırıldı.

Objects: Arguments:

array[0] → firstX = 10, firstY = 0

array[1] → firstX = 20, firstY = 0

array[2] → firstX = 30, firstY = 40

Member initializer list

C++ dilinde, sınıf üyelerini (veri üyeleri veya const üyeler gibi) başlatmak için kullanılan bir yapı olan "member initializer list" veya üye başlatıcı listesi, bir sınıfın constructor'ında kullanılır. Bu yapı, bir sınıfın üyelerine değer atamak için constructor'ın gövdesi yerine, constructor başlığının parantez içinde belirtilir.

Member initializer list kullanarak, bir sınıfın veri üyelerini başlatmak için daha etkili ve performanslı bir yol sağlanır. Özellikle, const veya referans üyeleri gibi durumlarda kullanımı zorunludur.

Örnek:

```
#include <iostream>

class MyClass {
private:
    int value;

public:
    // Constructor with member initializer list
    MyClass(int v) : value(v) {
        std::cout << "Constructor called with value: " << value << std::end
l;
    }

    // Getter function
    int getValue() const {
        return value;
    }
};

int main() {
    // Constructor with member initializer list kullanarak nesne oluşturma
    MyClass obj(42);

    // Nesnenin değerini ekrana yazdırma
    std::cout << "Value of obj: " << obj.getValue() << std::endl;
```

```
    return 0;
}
```

Yukarıdaki örnekte, `MyClass` adında bir sınıf tanımlandı. Bu sınıfın bir `value` adında bir veri üyesi var. Ardından, sınıfın constructor'ı tanımlandı.

Constructor başlığının hemen ardından, `value(v)` şeklinde bir member initializer list belirtildi. Bu, `value` adlı veri üyesine, constructor'a gelen `v` parametresinin değerini atar.

Bu şekilde, `value` veri üyesinin başlatılması için member initializer list kullanılmış oldu. Bu, daha etkili bir başlatma süreci sağlar ve özellikle const üyeleri veya referans üyeleri gibi özel durumlarda gereklidir.

Destructors

C++ dilinde bir destructor, bir nesnenin ömrü sona erdiğinde çağrılan özel bir üye fonksiyondur. Destructors, bir nesnenin bellekte ayrılan kaynakları serbest bırakmak veya temizlemek için kullanılır. Bu, dinamik olarak ayrılan belleği, dosya kapatma işlemlerini, veritabanı bağlantılarını vb. serbest bırakmak için kullanılabilir.

Destructors, sınıf isminden önce `~` simgesi ile tanımlanır. Bir sınıfın yalnızca bir destructor'ı olabilir ve parametre alamazlar veya bir değer döndüremezler.

Örnek:

```
#include <iostream>

class MyClass {
public:
    // Constructor
    MyClass() {
        std::cout << "Constructor called." << std::endl;
    }

    // Destructor
    ~MyClass() {
        std::cout << "Destructor called." << std::endl;
    }
};

int main() {
    // Bir nesne oluşturulduğunda constructor çağrılır
    MyClass obj;

    // Nesnenin ömrü sona erdiğinde destructor çağrılır
    return 0;
}
```

Yukarıdaki örnekte, `MyClass` adında bir sınıf tanımlanmıştır. Bu sınıfın bir constructor'ı ve bir destructor'ı vardır. `MyClass` türünden bir nesne oluşturulduğunda constructor çağrılır ve bu nesnenin

ömrü sona erdiğinde destructor çağrılır.

Destructors, bir nesnenin ömrünün sona erdiği durumlarda otomatik olarak çağrılır. Örneğin, bir nesne yerel bir kapsamdan çıktığında veya bir dinamik olarak oluşturulan nesne `delete` anahtar kelimesi ile serbest bırakıldığında destructor çağrılır. Bu sayede, bellek sızıntıları ve diğer kaynak sızıntıları gibi sorunlar önlenmiş olur.

Örnek 2:

```
#include <iostream>

class MyClass {
private:
    int* ptr;

public:
    // Constructor
    MyClass() {
        ptr = new int; // Dinamik bellek tahsisi
    }

    // Destructor
    ~MyClass() {
        delete ptr; // Dinamik belleği serbest bırak
    }
};

int main() {
    MyClass obj; // MyClass tipinden bir nesne oluşturulduğunda dinamik belle

    // Program sona erdiğinde, obj'nin ömrü sona erdiğinde destructor çağrılı
    return 0;
}
```

Yukarıdaki örnekte, `MyClass` adında bir sınıf tanımlanmıştır. Bu sınıf, bir tamsayı işaretçisi (`ptr`) içerir ve constructor'da dinamik olarak bellek tahsis edilir (`new` anahtar kelimesi kullanılarak). Destructor'da ise bu dinamik bellek tahsisini serbest bırakmak için `delete` anahtar kelimesi kullanılır. Bu sayede, bellek sızıntıları önlenmiş olur.

Constant Objects and const Member Functions

Constant Objects, nesne oluşturulduktan sonra içeriğinin değiştirilemeyeceği nesnelerdir. `const` üye fonksiyonlar ise sınıfın bir üyesi olan ve nesnenin durumunu değiştirmeyen fonksiyonlardır.

Örnek:

```
#include <iostream>

class Rectangle {
private:
```

```
    int width;
    int height;

public:
    Rectangle(int w, int h) : width(w), height(h) {}

    // const üye fonksiyon: Rectangle nesnesinin durumunu değiştirmez
    int area() const {
        return width * height;
    }

    // Bu üye fonksiyon, Rectangle nesnesinin durumunu değiştirir
    void resize(int new_width, int new_height) {
        width = new_width;
        height = new_height;
    }
};

int main() {
    // Constant object örneği
    const Rectangle rect(3, 4);

    // const üye fonksiyon çağrısı
    std::cout << "Alan: " << rect.area() << std::endl;

    // Bu satır hata verecektir, çünkü const nesnelerin durumunu değiştirem
ezsiniz.
    // rect.resize(5, 6);

    return 0;
}
```

Bu örnekte, `Rectangle` sınıfı bir dikdörtgeni temsil eder. `const` anahtar kelimesi, `area()` fonksiyonunun dikdörtgenin durumunu değiştirmeyeceğini belirtir. `resize()` fonksiyonu ise dikdörtgenin boyutunu değiştiren bir işlevdir.

`main()` fonksiyonunda `const Rectangle rect(3, 4)` ile bir sabit nesne oluşturulur. Bu nesne üzerinde `area()` fonksiyonu çağrılabilir çünkü bu fonksiyon, nesnenin durumunu değiştirmez. Ancak, `resize()` fonksiyonu hata verecektir çünkü `const` nesnelerin durumunu değiştirmeye izin vermez.

`mutable` anahtar kelimesi, bir sınıfın `const` olarak tanımlanmış nesneleri üzerinde bile değiştirilebilir olmasını sağlar. Bu durumda, sınıfın üye değişkenleri `mutable` olarak işaretlenirse, bu değişkenler `const` fonksiyonlar içinde bile değiştirilebilir.

Örnek:

```
#include <iostream>

class Counter {
private:
```

```
    mutable int count; // mutable anahtar kelimesi ile işaretlenmiş değişke
n

public:
    Counter(int c) : count(c) {}

    // const üye fonksiyon, ancak count değişkenini değiştirebilir
    void increment() const {
        ++count;
    }

    // const üye fonksiyon, count değişkenini değiştirebilir
    void reset() const {
        count = 0;
    }

    // const üye fonksiyon, count değerini döndürür
    int getCount() const {
        return count;
    }
};

int main() {
    const Counter c(5);

    std::cout << "Önceki sayım: " << c.getCount() << std::endl;

    c.increment(); // Bu çağrı, const üye fonksiyonu çağırır, ancak count d
eğerini değiştirebilir.
    std::cout << "Artırıldı: " << c.getCount() << std::endl;

    c.reset(); // Bu çağrı, const üye fonksiyonu çağırır, ancak count değer
ini sıfırlayabilir.
    std::cout << "Sıfırlandı: " << c.getCount() << std::endl;

    return 0;
}
```

Bu örnekte, `Counter` sınıfı bir sayacı temsil eder. `count` değişkeni, `mutable` anahtar kelimesiyle işaretlenmiştir, bu da `const` fonksiyonlar içinde bile değiştirilebileceği anlamına gelir.

`main()` fonksiyonunda, `const Counter c(5)` ile bir sabit nesne oluşturulur. `increment()` ve `reset()` fonksiyonları `const` olarak işaretlenmiştir, ancak `count` değişkenini değiştirebilirler çünkü `mutable` olarak işaretlenmiştir.

The Copy Constructor

Bir C++ sınıfının kopyalama kurucusu (copy constructor), bir nesnenin diğer bir nesneye kopyalanması gerektiğinde çağrılan özel bir üye fonksiyondur. Bu, nesne kopyalama işlemlerinin doğru şekilde gerçekleşmesini sağlar.

1. **Compiler Generated Copy Constructor (Derleyici tarafından oluşturulan kopyalama kurucusu):**

Eğer siz bir sınıfın kopyalama kurucusunu yazmazsanız, C++ derleyicisi otomatik olarak bir tane oluşturur. Bu otomatik olarak oluşturulan kopyalama kurucusu, her üye değişkenin kopyalanmasını gerçekleştirir. Ancak, bu varsayılan kopyalama kurucusu genellikle sadece üye değişkenlerin basit bir kopyasını alır. Eğer sınıfınızda dinamik bellek tahsisi veya başka karmaşık kaynak yönetimi varsa, bu otomatik olarak oluşturulan kopyalama kurucusu uygun davranmayabilir.

Örnek:

```
#include <iostream>

class MyClass {
private:
    int *ptr;

public:
    MyClass(int value) {
        ptr = new int(value);
    }

    // Derleyici tarafından oluşturulan kopyalama kurucusu
    // Bu durumda, doğru davranış sağlamaz
    /* MyClass(const MyClass &other) {
        ptr = new int(*other.ptr);
    } */

    ~MyClass() {
        delete ptr;
    }

    int getValue() const {
        return *ptr;
    }
};

int main() {
    MyClass obj1(5);
    MyClass obj2 = obj1; // Kopyalama işlemi

    std::cout << "obj1 değeri: " << obj1.getValue() << std::endl;
    std::cout << "obj2 değeri: " << obj2.getValue() << std::endl;
```

```
    return 0;
}
```

Bu örnekte, `MyClass` adında bir sınıf tanımlanmıştır ve `ptr` adında bir tam sayı işaretçisi üyesi vardır. Derleyici tarafından otomatik olarak oluşturulan kopyalama kurucusu `ptr` işaretçisinin kopyasını alır. Ancak, bu, derin kopyalama yapmadığı için iki nesnenin `ptr` işaretçisi aynı bellek alanını paylaşır, bu da problemlere yol açabilir.

1. **Programmer Written Copy Constructor (Programcı tarafından yazılan kopyalama kurucusu):**

Programcılar, sınıflarının kopyalama kurucusunu kendileri yazarak özel davranışlar ekleyebilirler. Bu, dinamik bellek tahsisi veya başka özel kaynak yönetimi durumlarında çok önemlidir.

Örnek:

```
#include <iostream>

class MyClass {
private:
    int *ptr;

public:
    MyClass(int value) {
        ptr = new int(value);
    }

    // Programcı tarafından yazılan kopyalama kurucusu
    MyClass(const MyClass &other) {
        ptr = new int(*other.ptr);
    }

    ~MyClass() {
        delete ptr;
    }

    int getValue() const {
        return *ptr;
    }
};

int main() {
    MyClass obj1(5);
    MyClass obj2 = obj1; // Kopyalama işlemi

    std::cout << "obj1 değeri: " << obj1.getValue() << std::endl;
    std::cout << "obj2 değeri: " << obj2.getValue() << std::endl;

    return 0;
}
```

Bu kez, `MyClass` sınıfı için özel bir kopyalama kurucusu tanımlandı. Bu kopyalama kurucusu, derin kopyalama yaparak `ptr` işaretçisinin aynı bellek alanını paylaşmasını önler. Bu, her nesne kendi bellek alanına sahip olur ve kaynakların güvenli bir şekilde yönetilmesini sağlar.

Deleting the Copy Constructor

"Deleting the copy constructor" ifadesi, bir sınıfın kopyalama kurucusunun (copy constructor) kullanılmasını engellemek için belirli koşullar altında işaretlenmesini ifade eder. Bu durum genellikle bir sınıfın kopyalanmasının istenmediği durumlarda kullanılır. Örneğin, bir sınıfın kopyalanmasını engellemek isteyebilirsiniz çünkü kopyalama işlemi mantıksal olarak uygun değildir veya sınıfın belirli kaynaklarını paylaşılamaz hale getirir.

Örnek:

```
#include <iostream>

class NonCopyableClass {
public:
    NonCopyableClass() {}

    // Kopyalama kurucusunu silme
    NonCopyableClass(const NonCopyableClass &) = delete;

    void showMessage() const {
        std::cout << "Bu sınıf kopyalanamaz." << std::endl;
    }
};

int main() {
    NonCopyableClass obj1;
    // NonCopyableClass obj2 = obj1; // Bu satır hata verecektir, çünkü kop
yalama kurucusu silinmiştir.

    obj1.showMessage();

    return 0;
}
```

Bu örnekte, `NonCopyableClass` adında bir sınıf tanımlanmıştır. Bu sınıfın kopyalama kurucusu, `= delete` ifadesi ile silinmiştir. Bu, sınıfın kopyalanmasını engeller.

`main()` fonksiyonunda, `NonCopyableClass` sınıfından bir nesne oluşturulur. Ancak, bu nesnenin başka bir nesneye kopyalanması istenmediği için kopyalama işlemi hata verir.

Ek olarak bir sınıfın kopyalama kurucusunu (copy constructor) özel (private) olarak yazarak, sınıfın dışındaki kodun bu kopyalama kurucusunu çağıramamasını sağlayabilirsiniz. Bu, sınıfın kopyalanmasını engellemenin bir yoludur, çünkü kopyalama kurucusu özel olarak işaretlendiği için yalnızca sınıfın içinden erişilebilir olur. Bu da sınıfın kopyalanmasını engeller.

Örnek:

```
#include <iostream>

class NonCopyableClass {
private:
    NonCopyableClass(const NonCopyableClass &) {} // Kopyalama kurucusunu ö
zel olarak tanımlama ve böylece silme

public:
    NonCopyableClass() {}

    void showMessage() const {
        std::cout << "Bu sınıf kopyalanamaz." << std::endl;
    }
};

int main() {
    NonCopyableClass obj1;
    // NonCopyableClass obj2 = obj1; // Bu satır hata verecektir, çünkü kop
yalama kurucusu özel olarak tanımlanmıştır.

    obj1.showMessage();

    return 0;
}
```

Bu örnekte, `NonCopyableClass` sınıfının kopyalama kurucusu özel olarak tanımlanmıştır ve dolayısıyla sınıfın dışından erişilemez hale gelmiştir. Bu nedenle, sınıfın dışındaki kodlar, bir nesneyi başka bir nesneye kopyalayamaz.

Bu yaklaşım, sınıfın kopyalanmasını engellemenin bir yoludur, ancak unutulmamalıdır ki bu sınıfın türetilmiş sınıflarının kopyalanmasını da engelleyecektir. Bu durumda, türetilmiş sınıfın da kopyalama işlemlerini ele alacak şekilde özel bir kopyalama kurucusu tanımlanması gerekebilir.

Passing Objects to Functions as Arguments

Nesneleri fonksiyonlara argüman olarak iletmek (passing objects to functions as arguments), C++ programlamada sıkça kullanılan bir tekniktir. Bu, fonksiyonların belirli bir nesne üzerinde işlem yapmasını sağlar. Nesneleri fonksiyonlara değer (value) olarak veya referans (reference) olarak geçmek, farklı davranışlar sergiler.

1. **Nesneleri Değer (Value) Olarak Geçmek:**

Nesneleri değer olarak geçmek, aslında nesnenin bir kopyasını oluşturur ve bu kopyayı fonksiyona iletir. Bu, orijinal nesnenin değişmez olduğu anlamına gelir. Fonksiyon içinde yapılan değişiklikler, sadece kopya nesne üzerinde etki eder.

Örnek:

```
#include <iostream>

class MyClass {
public:
    int value;

    MyClass(int v) : value(v) {}
};

void modifyValue(MyClass obj) {
    obj.value = 100; // Değişiklik, kopya nesne üzerinde yapılır
}

int main() {
    MyClass obj(42);

    modifyValue(obj);

    std::cout << "Orijinal değer: " << obj.value << std::endl;

    return 0;
}
```

Bu örnekte, `modifyValue()` fonksiyonuna `obj` adında bir `MyClass` nesnesi değer olarak geçirilir. Ancak, fonksiyon içinde yapılan değişiklikler, sadece bu nesnenin kopyası üzerinde etki eder. Dolayısıyla, `main()` fonksiyonunda, orijinal nesnenin değeri değişmemiş olarak kalır.

1. **Nesneleri Referans (Reference) Olarak Geçmek:**

Nesneleri referans olarak geçmek, nesnenin kendisinin fonksiyona iletilmesini sağlar. Bu, fonksiyon içinde yapılan değişikliklerin orijinal nesne üzerinde doğrudan etki etmesini sağlar.

Örnek:

```
#include <iostream>

class MyClass {
public:
    int value;

    MyClass(int v) : value(v) {}
};

void modifyValue(MyClass &obj) {
    obj.value = 100; // Değişiklik, orijinal nesne üzerinde yapılır
}

int main() {
    MyClass obj(42);
```

```
    modifyValue(obj);

    std::cout << "Değiştirilmiş değer: " << obj.value << std::endl;

    return 0;
}
```

Bu örnekte, `modifyValue()` fonksiyonuna `obj` adında bir `MyClass` nesnesi referans olarak geçirilir. Fonksiyon içinde yapılan değişiklikler, orijinal nesne üzerinde doğrudan etki eder. Dolayısıyla, `main()` fonksiyonunda, orijinal nesnenin değeri değişir ve `obj.value` 100 olarak yazdırılır.

`this` Pointer

`this` anahtar kelimesi, bir sınıfın üye fonksiyonları içinde o sınıfa ait olan mevcut nesneyi işaret eder. Bu, sınıfın kendi özelliklerine ve metotlarına içeriden erişmek için kullanılır. Özellikle aynı isimde farklı bir değişken veya parametre ile karışıklığı önlemek için kullanışlıdır.

Örnek:

```
#include <iostream>

class MyClass {
private:
    int value;

public:
    MyClass(int v) : value(v) {}

    void printValue() {
        std::cout << "Value: " << this->value << std::endl; // 'this' anaht
ar kelimesi ile mevcut nesneye erişim
    }

    void setValue(int v) {
        this->value = v; // 'this' anahtar kelimesi ile mevcut nesneye eriş
im
    }
};

int main() {
    MyClass obj1(42);
    obj1.printValue(); // Value: 42

    obj1.setValue(100);
    obj1.printValue(); // Value: 100
```

```
    return 0;
}
```

Bu örnekte, 'MyClass' adında bir sınıf tanımlanmıştır. Sınıfın üye fonksiyonları içinde 'this' anahtar kelimesi kullanılarak mevcut nesneye erişim sağlanmıştır. 'printValue()' fonksiyonunda 'this→value' ile mevcut nesnenin 'value' üye değişkenine erişilir ve değeri yazdırılır. 'setValue()' fonksiyonunda da 'this→value' ile mevcut nesnenin 'value' üye değişkenine yeni bir değer atanır.

Bu şekilde, 'this' anahtar kelimesi ile sınıf içindeki metotlarda mevcut nesneye erişim sağlanabilir ve bu, sınıfın içindeki farklı alanlarda aynı isimde değişkenlerin veya parametrelerin kullanılması durumunda kullanışlıdır.

Returning `this`

"This" anahtar kelimesi, bir sınıfın üye fonksiyonlarında o fonksiyonun çağrıldığı mevcut nesneyi işaret eder. Bu, nesnenin kendisini döndürmek için kullanılabilir. Bu, zincirleme metodları (chained methods) uygulamak veya bir fonksiyonun çağrıldığı nesneyi döndürmek için kullanılabilir.

Örnek:

```
#include <iostream>

class MyClass {
private:
    int value;

public:
    MyClass(int v) : value(v) {}

    MyClass& setValue(int v) {
        this->value = v;
        return *this; // Mevcut nesneyi döndürme
    }

    int getValue() const {
        return this->value;
    }
};

int main() {
    MyClass obj(42);
    std::cout << "Orijinal değer: " << obj.getValue() << std::endl;

    obj.setValue(100).setValue(200); // Zincirleme metodları kullanma
    std::cout << "Değiştirilmiş değer: " << obj.getValue() << std::endl;

    return 0;
}
```

Bu örnekte, "MyClass" adında bir sınıf tanımlanmıştır. "setValue()" adında bir üye fonksiyon, nesnenin "value" üye değişkenine bir değer atar ve ardından mevcut nesneyi döndürür. Böylece zincirleme metodları kullanılabilir. "getValue()" fonksiyonu ise "value" üye değişkeninin değerini döndürür.

"main()" fonksiyonunda, önce "obj" adında bir "MyClass" nesnesi oluşturulur ve başlangıç değeriyle initialize edilir. Daha sonra "setValue()" fonksiyonu zincirleme olarak iki kez çağrılır ve "getValue()" fonksiyonuyla değer kontrol edilir.

Bu şekilde, "this" anahtar kelimesi ile mevcut nesneyi döndürmek, zincirleme metodları uygulamak veya fonksiyonun çağrıldığı nesneyi döndürmek için kullanılabilir.

Bir üye fonksiyon içinde mevcut nesneyi döndürmek için iki farklı yaklaşım vardır: birincisi bir referans, diğeri ise bir işaretçi (pointer) kullanmaktır.

1. **Referans olarak this döndürme:**

```
#include <iostream>

class MyClass {
private:
    int value;

public:
    MyClass(int v) : value(v) {}

    MyClass& setValue(int v) {
        this->value = v;
        return *this; // Mevcut nesneyi referans olarak döndürme
    }

    int getValue() const {
        return this->value;
    }
};

int main() {
    MyClass obj(42);
    std::cout << "Orijinal değer: " << obj.getValue() << std::endl;

    obj.setValue(100).setValue(200); // Zincirleme metodları kullanma
    std::cout << "Değiştirilmiş değer: " << obj.getValue() << std::endl;

    return 0;
}
```

Bu örnekte, "setValue()" fonksiyonu "MyClass" tipinden bir referans döndürür. Fonksiyon içinde "return *this;" ifadesi kullanılarak, fonksiyonun çağrıldığı mevcut nesneyi referans olarak döndürür.

1. **İşaretçi (Pointer) olarak this döndürme:**

```
#include <iostream>

class MyClass {
private:
    int value;

public:
    MyClass(int v) : value(v) {}

    MyClass* setValue(int v) {
        this->value = v;
        return this; // Mevcut nesneyi işaretçi olarak döndürme
    }

    int getValue() const {
        return this->value;
    }
};

int main() {
    MyClass obj(42);
    std::cout << "Orijinal değer: " << obj.getValue() << std::endl;

    obj.setValue(100)->setValue(200); // Zincirleme metodları kullanma
    std::cout << "Değiştirilmiş değer: " << obj.getValue() << std::endl;

    return 0;
}
```

Bu örnekte, "setValue()" fonksiyonu "MyClass" tipinden bir işaretçi döndürür. Fonksiyon içinde "return this;" ifadesi kullanılarak, fonksiyonun çağrıldığı mevcut nesneyi işaretçi olarak döndürür.

Her iki örnek de aynı sonucu üretir ve zincirleme metodları uygulamak için kullanılabilir. Ancak, dönüş türü farklıdır: birincisinde referans, ikincisinde ise pointer kullanılmıştır.

Static Data Members:

Static data members, bir sınıfa ait özellikleri temsil eden ve sınıfın tüm örnekleri arasında paylaşılan değişkenlerdir. Örneğin, bir araba sınıfı düşünelim. Arabaların sayısını takip etmek için bir static data member kullanabiliriz.

```
#include <iostream>

class Car {
public:
    static int carCount; // Static data member

    Car() {
```

```
        carCount++; // Her yeni araba oluşturulduğunda carCount'u arttır
    }
};

int Car::carCount = 0; // Static data member'ın tanımı ve başlangıç değeri

int main() {
    Car car1;
    Car car2;

    std::cout << "Total cars: " << Car::carCount << std::endl; // Araba say
ısını yazdır
    return 0;
}
```

Static Constant Members:

Static constant members, bir sınıfa ait sabit özelliklerdir ve sınıfın herhangi bir örneği oluşturulmadan erişilebilirler. Örneğin, bir matematik sınıfı içinde pi sayısını bir static constant member olarak tanımlayabiliriz.

```
#include <iostream>

class Math {
public:
    static const double pi; // Static constant member
};

const double Math::pi = 3.14159; // Static constant member'ın tanımı ve değ
eri

int main() {
    std::cout << "Value of pi: " << Math::pi << std::endl; // pi sayısını y
azdır
    return 0;
}
```

Static Methods (Function Members):

Static methods, sınıfa ait fonksiyonlardır ve herhangi bir sınıf örneği oluşturulmadan doğrudan sınıf adıyla çağrılabilirler. Örneğin, bir utility sınıfında, iki sayının toplamını hesaplayan bir static method tanımlayabiliriz.

```
#include <iostream>

class Math {
public:
    static int add(int x, int y) { // Static method
```

```
        return x + y;
    }
};

int main() {
    std::cout << "5 + 3 = " << Math::add(5, 3) << std::endl; // İki sayının
toplamını yazdır
    return 0;
}
```

The Unified Modeling Language - UML

Unified Modeling Language (UML), yazılım mühendisliğinde kullanılan bir standart modelleme dilidir. UML, yazılım sistemlerinin analiz, tasarım, uygulama ve dokümantasyonu için grafiksel bir notasyon sunar. Bu grafiksel notasyon, yazılım sistemlerinin farklı yönlerini görselleştirmek için çeşitli diyagram türlerini içerir.

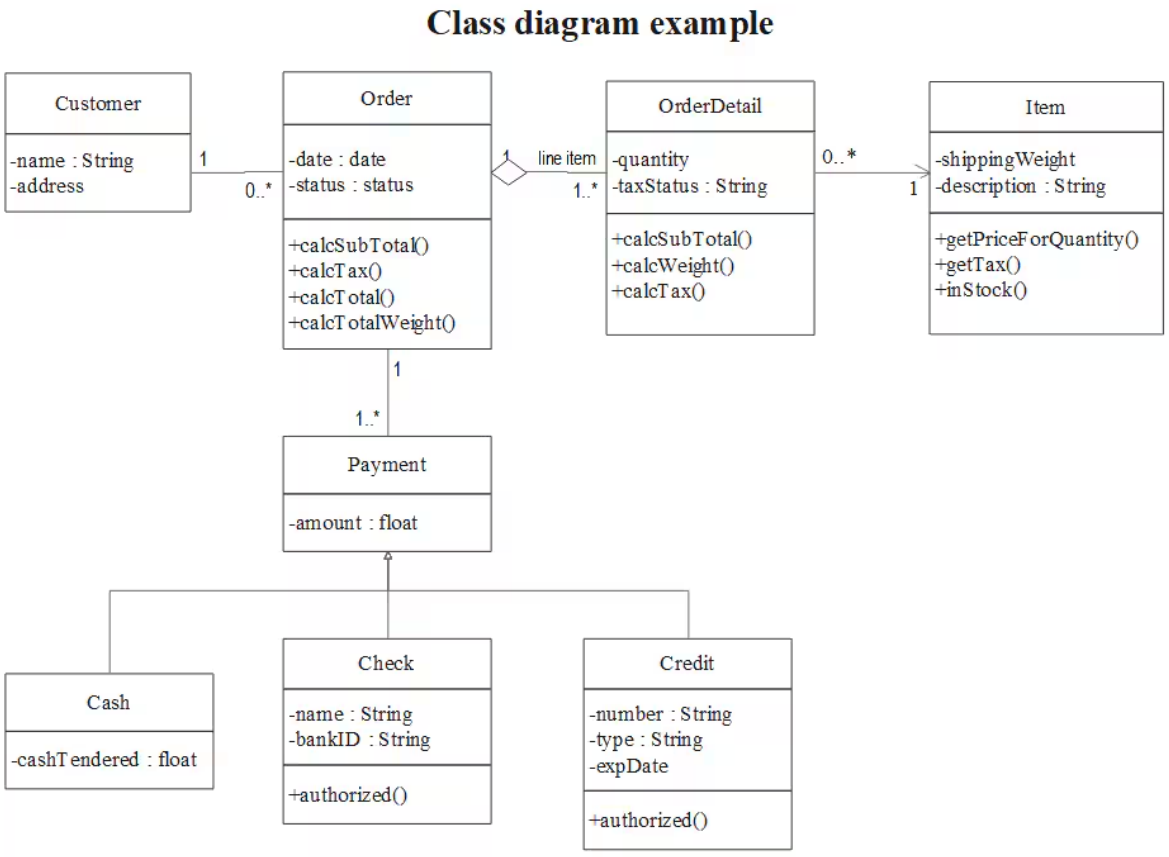
UML'nin temel amacı, karmaşık yazılım sistemlerini daha anlaşılabilir ve yönetilebilir hale getirmektir. Bir yazılım sisteminin tüm yönlerini (yapısal, davranışsal ve işlevsel) tanımlamak için kullanılabilir ve bu da yazılım geliştirme sürecinde paydaşlar arasında daha iyi bir iletişim ve anlayış sağlar.

> UML sistem dizaynı için bir metot değildir; yazılım sistemlerini analiz etmek için kullanılır.

UML'nin bazı yaygın kullanılan diyagram türleri şunlardır:

1. **Sınıf Diyagramları (Class Diagrams)**: Sınıflar arasındaki ilişkileri ve sınıfların özelliklerini ve yöntemlerini gösterir.
2. **Davranışsal Diyagramlar (Behavioral Diagrams)**: Sistem davranışını, süreçleri ve zamanlama ile ilgili detayları görselleştirir. Örnekler arasında durum, etkinlik ve zamanlama diyagramları bulunur.
3. **Yapısal Diyagramlar (Structural Diagrams)**: Sistemdeki yapısal öğeleri ve bu öğeler arasındaki ilişkileri gösterir. Örnekler arasında paket, bileşen ve dağıtım diyagramları bulunur.
4. **Etkileşim Diyagramları (Interaction Diagrams)**: Nesnelerin ve sınıfların birbirleriyle nasıl etkileşimde bulunduğunu gösterir. Örnekler arasında zaman çizelgesi ve iletişim diyagramları bulunur.
5. **Durum Diyagramları (State Diagrams)**: Bir nesnenin farklı durumlarını ve durumlar arasındaki geçişleri gösterir.

UML'nin bu çeşitli diyagram türleri, farklı yönleriyle yazılım sistemlerini tanımlamak ve görselleştirmek için kullanılır. Bu, yazılım geliştirme sürecinde analiz, tasarım, uygulama ve bakım aşamalarında kullanılarak, yazılım projelerinin daha etkin bir şekilde yönetilmesine ve iletişiminin sağlanmasına yardımcı olur.



OOP05

Operator Overloading

Operator Overloading, C++ programlama dilinde mevcut olan operatörlerin (örneğin, +, -, *, /) özel bir işlevsellikle yeniden tanımlanmasını sağlayan bir özelliktir. Bu, özel türlerin üzerinde doğrudan operatörlerin kullanılmasını mümkün kılar.

1. Operatör Overloading Nasıl Çalışır?

C++ dilinde, bir sınıfın üyeleri olarak tanımlanabilen belirli operatörler için özel işlevler tanımlayabiliriz. Bu özel işlevler, operatörün çalışmasını sağlamak için çağrılacaktır. Örneğin, iki nesneyi toplamak için `+` operatörünü kullanmak istiyorsak, bu operatörün çağrılmasını sağlayan bir `operator+` işlevi tanımlayabiliriz.

2. Operatör Overloading'in Avantajları

- Okunabilirlik ve Yazılabilirlik: Özel işlevler, belirli operatörlerin ne yapacağını açıkça ifade eder, bu da kodun okunabilirliğini artırır.
- Dilin Yeteneklerini Genişletme: C++'ın esnekliği sayesinde, kullanıcı tanımlı türlerin üzerinde bile operatörleri yeniden tanımlayabiliriz, bu da dilin yeteneklerini genişletir.

- Doğal Kullanım: Operatörlerin kullanılması, belirli türler üzerinde doğal ve sezgisel işlemler gerçekleştirilmesini sağlar.

3. Operatör Overloading Örnekleri

```
#include <iostream>

class Complex {
private:
    double real;
    double imaginary;

public:
    Complex(double r = 0.0, double i = 0.0) : real(r), imaginary(i) {}

    // + operatörünün aşırı yüklenmesi
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imaginary + other.imaginary);
    }

    // << operatörünün aşırı yüklenmesi (stream insertion operator)
    friend std::ostream& operator<<(std::ostream& os, const Complex& comple
x) {
        os << complex.real << " + " << complex.imaginary << "i";
        return os;
    }
};

int main() {
    Complex a(3.0, 4.0);
    Complex b(1.0, 2.0);

    Complex c = a + b; // + operatörü çağrılacak
    std::cout << "a + b = " << c << std::endl; // << operatörü çağrılacak

    return 0;
}
```

Bu örnek, `Complex` adında bir sınıf tanımlar ve bu sınıfta `+` ve `<<` operatörlerini aşırı yükler. Böylece, `Complex` türündeki nesneler üzerinde doğrudan `+` ve `<<` operatörlerini kullanabiliriz.

4. Dikkat Edilmesi Gerekenler

- Operatör Overloading, belirli kurallara uygun olmalıdır. Örneğin, operatörün imzası (parametre listesi ve dönüş türü) değiştirilemez.
- Operatörlerin semantiği, beklenen işleve uygun olmalıdır. Örneğin, `+` operatörü normalde iki sayıyı toplar, bu nedenle `Complex` türünde bir nesne için aşırı yüklenmişse, bu nesnelerin toplamını vermelidir.

Operator Overloading, C++'ın güçlü ve esnek bir özelliğidir, ancak doğru ve dikkatli bir şekilde kullanılmalıdır. Uygun şekilde kullanıldığında, kodun okunabilirliğini artırabilir ve belirli türler üzerinde doğal operasyonlar gerçekleştirmeyi mümkün kılar.

OOP06

Relationships Between Objects

Bir işyerinde, çalışanlar farklı departmanlarda görev yaparlar. Her departman, belirli bir işlevi yerine getirmek üzere bir araya gelmiş bir grup çalışandır. Örneğin, bir şirkette mühendislik, satış, insan kaynakları ve muhasebe gibi farklı departmanlar bulunabilir. Her departman, belirli bir rolü yerine getirmek için çalışanlardan oluşur.
Departmanlar ve çalışanlar arasındaki ilişki, OOP'de sınıflar arasındaki ilişkilere benzer.

- Departmanlar: Her departman bir sınıf olabilir. Örneğin, Department sınıfı, bir departmanın adını, numarasını ve çalışanlarını saklayabilir.
- Çalışanlar: Her çalışan da bir sınıf olabilir. Örneğin, Employee sınıfı, bir çalışanın adını, soyadını, pozisyonunu ve maaşını saklayabilir.

Şimdi, bu iki sınıf arasındaki ilişkilere bakalım:

- Bir Departmanın Çalışanları: Her departman, bir veya daha fazla çalışan içerebilir. Bu, bir departmanın içindeki çalışanları saklamak için Employee nesnelerinin bir koleksiyonunu içeren bir veri yapısı kullanarak sağlanabilir.
- Bir Çalışanın Bağlı Olduğu Departman: Her çalışan, bir departmana bağlıdır. Bu, her Employee nesnesinin bir Department nesnesine referans tutmasıyla sağlanabilir.

Bu şekilde, gerçek hayattaki çalışanlar ve departmanlar arasındaki ilişkiyi, OOP'deki sınıflar ve nesneler arasındaki ilişkilerle ilişkilendirebiliriz. Bu yaklaşım, yazılım sistemlerini gerçek dünya senaryolarına daha yakın bir şekilde modellememize ve tasarlamamıza yardımcı olur.

Types of Relationships in Object-Oriented Design (OOD)

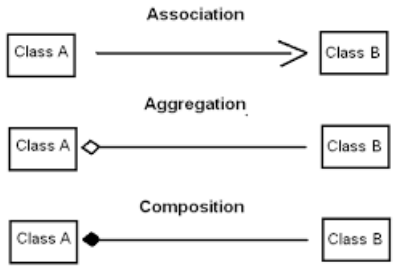
Relationship'lerin iki temel tipi vardır.

1. **Association (İlişki/has-a relationship)**:
   Association, bir sınıfın diğer bir sınıfın nesneleriyle ilişkili olduğunu ifade eder. Bu ilişki genellikle "bir sınıfın bir veya daha fazla nesnesi, diğer sınıfın bir veya daha fazla nesnesiyle ilişkilidir" şeklinde açıklanır. Örneğin, bir öğrenci sınıfı ve bir ders sınıfı arasındaki ilişki, bir öğrencinin bir veya birden fazla ders almasıdır.
   Örnek → A student has a class.
2. **Inheritance (Kalıtım/is-a relationship)**:
   Inheritance, bir sınıfın başka bir sınıftan özelliklerini ve davranışlarını miras aldığı ilişki türüdür. Bu, "bir sınıf bir diğer sınıfın bir türüdür" şeklinde ifade edilir. Örneğin, bir araç sınıfı ve bir otomobil sınıfı arasındaki ilişki, otomobilin araç sınıfından miras almasıdır.
   Örnek → A automobile is a vehicle.

Association (ilişki) kendi başına bir has-a (sahip) ilişkisi değildir. Ancak, Association'ın alt tipleri olan Aggregation ve Composition, has-a ilişkisinin farklı türlerini temsil eder.

1. **Aggregation (Toplantı)**:
   Aggregation, bir nesnenin diğer nesneleri içerebileceği ancak bu nesnelerin bağımsız olarak var olabileceği ilişki türüdür. Bu, "bir bütün parçalarından oluşur" şeklinde ifade edilir. Örneğin, bir üniversite sınıfı ve bir bölüm sınıfı arasındaki ilişki, bir üniversitenin bir veya birden fazla bölümü içermesidir.
2. **Composition (Birleşim)**:
   Composition, bir nesnenin diğer nesneleri içerdiği ve bu nesnelerin birlikte yaşadığı ve yok olduğu ilişki türüdür. Bu, "bir bütün parçalarıdır" şeklinde ifade edilir. Örneğin, bir bilgisayar sınıfı ve bir işlemci sınıfı arasındaki ilişki, bilgisayarın bir işlemci içermesi ve işlemci olmadan bilgisayarın var olamamasıdır.

Bu şekilde, Association gibi bir genel ilişki türünü daha spesifik ilişki türleri olan Aggregation ve Composition'a genelleyebiliriz. Her biri, sınıflar arasındaki ilişkileri ve yapıları daha iyi tanımlamak için kullanılır.



Association (”uses-a” relationship)

Association (uses-a relationship) genellikle bir nesnenin başka bir nesneyi içermesi veya kullanması durumunu ifade eder. Bu durumda, bir nesne başka bir nesneyi içerir veya ona bağımlıdır, ancak aralarında sıkı bir sahiplik ilişkisi yoktur. Bu ilişki, bir nesnenin diğerini içermesiyle tanımlanır, ancak içeren nesne, içerdiği nesnenin ömrünü etkilemez.

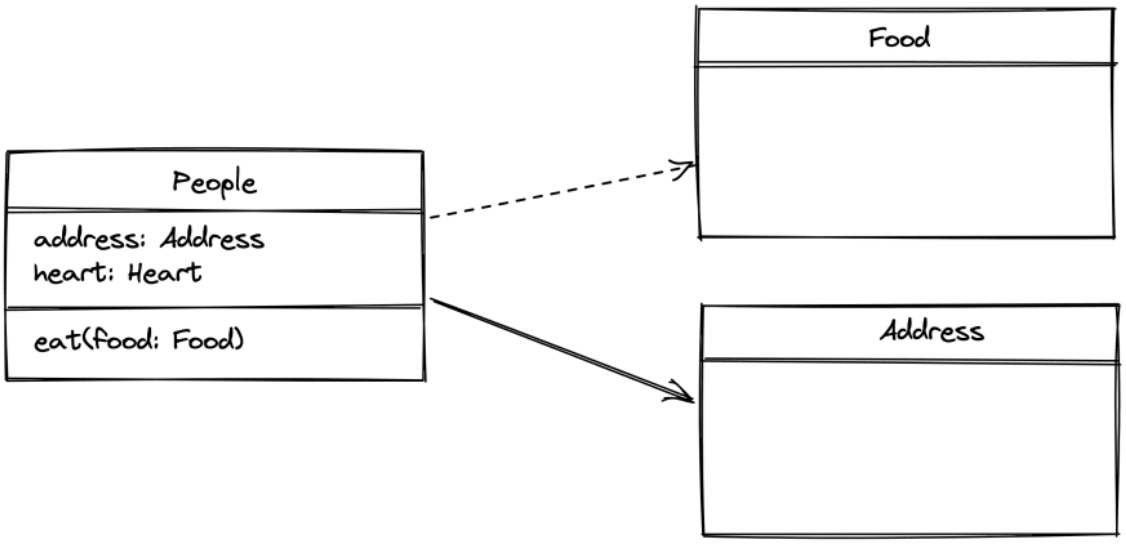
Örneğin, bir bilgisayar sınıfını düşünelim. Bu sınıf, birçok farklı parçadan oluşur, örneğin bir işlemci, bir bellek, bir anakart, bir ekran kartı vb. Bu durumda, bilgisayar sınıfı, diğer parçaları kullanır veya içerir, ancak bu parçaların yaşam döngüsü bilgisayar nesnesi tarafından yönetilmez.
Örnek:

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

// Parça sınıfları
class Processor {
public:
    void process() {
        std::cout << "Processing data..." << std::endl;
    }
};
```

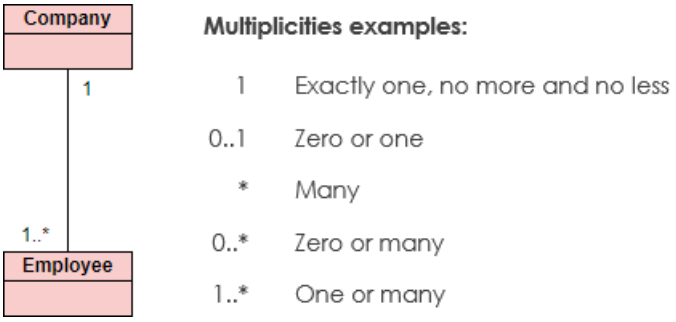
```
class Memory {
public:
    void store() {
        std::cout << "Storing data..." << std::endl;
    }
};

// Bilgisayar sınıfı, Processor ve Memory sınıflarını kullanır.
class Computer {
private:
    Processor processor;
    Memory memory;
public:
    void run() {
        processor.process();
        memory.store();
    }
};

int main() {
    Computer myComputer;
    myComputer.run();
    return 0;
}
```

Bu örnekte, `Computer` sınıfı, `Processor` ve `Memory` sınıflarını kullanır (uses-a relationship). `Computer` sınıfı, `Processor` ve `Memory` sınıflarını içerir, ancak bu sınıfların ömrünü kontrol etmez. `Computer` nesnesi oluşturulduğunda, onun bir parçası olarak `Processor` ve `Memory` nesneleri de otomatik olarak oluşturulur.

UML Notation:



Multiplicity:



Aggregation

Aggregation, nesne yönelimli programlamada bir sınıfın, başka bir sınıfın nesnesine sahip olduğu bir ilişki türüdür. Bu ilişki, "has-a" (sahip olma) ilişkisi olarak da bilinir. Aggregation'da, nesne yaşam döngüleri bağımsızdır; yani, bir nesne diğerinin bir parçası olabilir, ancak her biri kendi başına var olabilir.

Örnek:

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

// Bir Sınıf tanımlıyoruz
class Address {
public:
    std::string city;
    std::string street;
    int number;

    Address(const std::string& city, const std::string& street, int number)
        : city(city), street(street), number(number) {}

    void display() const {
        std::cout << street << " St., " << number << ", " << city << std::e
ndl;
    }
};

// Başka bir Sınıf tanımlıyoruz
class Person {
private:
    std::string name;
    int age;
    Address* address;  // Aggregation: Person nesnesi bir Address nesnesine
sahiptir

public:
    Person(const std::string& name, int age, Address* address)
```

```
        : name(name), age(age), address(address) {}

    void display() const {
        std::cout << "Name: " << name << std::endl;
        std::cout << "Age: " << age << std::endl;
        std::cout << "Address: ";
        address->display();
    }
};

int main() {
    // Address nesnesi oluşturuyoruz
    Address address("Istanbul", "Ataturk", 10);

    // Person nesnesi oluşturuyoruz ve Address nesnesini kullanıyoruz
    Person person("Ahmet", 30, &address);

    person.display();

    return 0;
}
```

Bu örnekte:

1. `Address` sınıfı, bir kişinin adres bilgilerini içerir.
2. `Person` sınıfı, bir kişinin adını, yaşını ve bir `Address` nesnesini içerir.
3. `Person` sınıfı, bir `Address` nesnesine sahip olmasına rağmen, `Address` nesnesi `Person` sınıfının bir parçası değildir. Bu, aggregation'ı gösterir.

`main` fonksiyonunda, bir `Address` nesnesi oluşturup, bu nesneyi bir `Person` nesnesi için kullanıyoruz. `Address` ve `Person` nesneleri bağımsız olarak var olabilir ve `Address` nesnesi `Person` nesnesinin bir parçası değildir, sadece ona referans olarak geçmektedir. Bu, aggregation'ın önemli bir özelliğidir.

Composition

Composition, nesne yönelimli programlamada bir sınıfın başka bir sınıfın nesnesine sahip olduğu ve bu sahipliğin yaşam döngüsüne bağlı olduğu bir ilişki türüdür. Yani, ana nesne yok edildiğinde, ona ait olan nesneler de yok edilir. Bu ilişki de "has-a" (sahip olma) ilişkisi olarak adlandırılır, ancak aggregation'dan farklı olarak, bileşen nesneler ana nesnenin bir parçasıdır ve onunla aynı yaşam döngüsünü paylaşır.

Örnek:

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

// Bir Sınıf tanımlıyoruz
class Engine {
public:
```

```
    std::string type;

    Engine(const std::string& type) : type(type) {}

    void start() const {
        std::cout << "Engine of type " << type << " is starting..." << st
d::endl;
    }
};

// Başka bir Sınıf tanımlıyoruz
class Car {
private:
    std::string brand;
    Engine engine;  // Composition: Car nesnesi bir Engine nesnesine sahipt
ir

public:
    Car(const std::string& brand, const std::string& engineType)
        : brand(brand), engine(engineType) {}  // engine nesnesi Car nesnes
inin bir parçasıdır

    void startCar() const {
        std::cout << "Starting car " << brand << std::endl;
        engine.start();
    }
};

int main() {
    // Car nesnesi oluşturuyoruz ve bu nesne kendi Engine nesnesine sahip
    Car car("Toyota", "V8");

    car.startCar();

    return 0;
}
```

Bu örnekte:

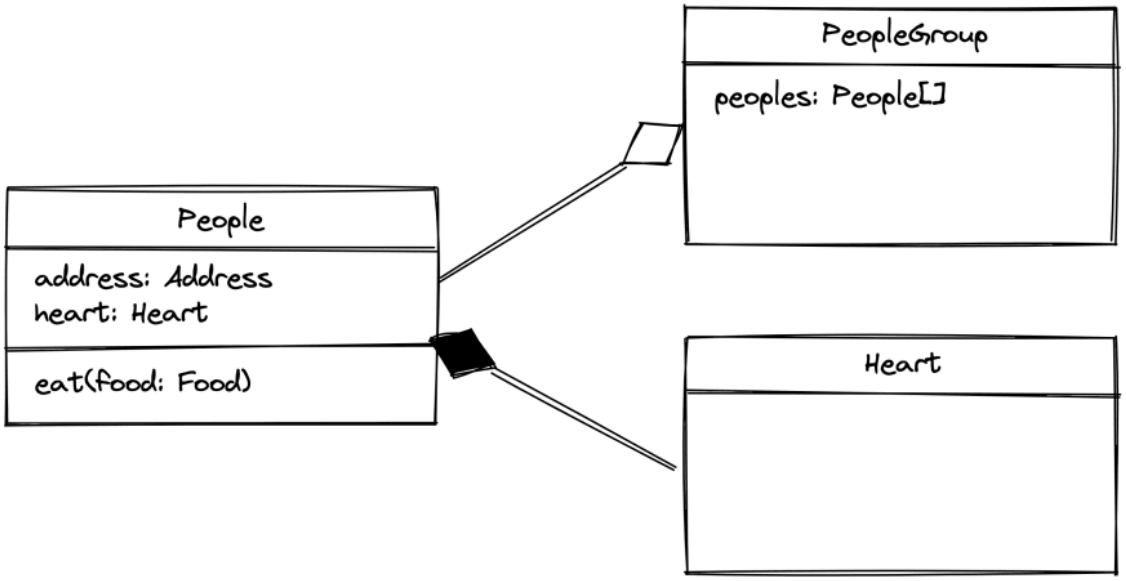
1. `Engine` sınıfı, bir aracın motor tipini içerir ve motorun çalıştırılması için bir metot sağlar.
2. `Car` sınıfı, bir aracın markasını ve bir `Engine` nesnesini içerir.
3. `Car` sınıfı, `Engine` nesnesine sahip olup, bu nesne `Car` nesnesinin bir parçasıdır ve onunla aynı yaşam döngüsünü paylaşır.

`main` fonksiyonunda, bir `Car` nesnesi oluşturuyoruz. `Car` nesnesi, `Engine` nesnesini içerir ve `Car` nesnesi yok edildiğinde, `Engine` nesnesi de yok edilir. Bu, composition'ın önemli bir özelliğidir.

Composition, bileşen nesnelerinin ana nesne ile sıkı bir şekilde bağlı olduğu ve yaşam döngülerinin birbirine bağlı olduğu bir ilişkiyi temsil eder. Bu ilişki, bileşen nesnelerinin ana nesnenin ayrılmaz bir

parçası olduğunu ve bağımsız olarak var olamayacaklarını gösterir.

UML Notation:



Default Constructors and Destructors in Composition

Bir sınıf, başka bir sınıfın nesnelerine sahip olduğunda, bu bileşen nesnelerin yaşam döngüsü ana sınıfın yaşam döngüsüne bağlıdır. Bu nedenle, bileşen nesneler için yapıcılar (constructors) ve yıkıcılar (destructors) otomatik olarak çağrılır.

Örnek:

```
#include <iostream>

// Bileşen Sınıf
class Engine {
public:
    Engine() {
        std::cout << "Engine created" << std::endl;
    }

    ~Engine() {
        std::cout << "Engine destroyed" << std::endl;
    }
};

// Ana Sınıf
class Car {
private:
    Engine engine;  // Composition: Car nesnesi bir Engine nesnesine sahipt
ir
```

```
public:
    Car() {
        std::cout << "Car created" << std::endl;
    }

    ~Car() {
        std::cout << "Car destroyed" << std::endl;
    }
};

int main() {
    Car myCar;  // Car nesnesi oluşturuluyor
    return 0;
}
```

Bu örnekte:

1. `Engine` sınıfı, bir default constructor ve destructor tanımlar.
2. `Car` sınıfı da bir default constructor ve destructor tanımlar.
3. `Car` sınıfı, `Engine` sınıfına ait bir nesneye sahiptir.

Program çalıştırıldığında, `Car` nesnesi oluşturulduğunda ve yok edildiğinde, aşağıdaki çıktı alınır:

```
Engine created
Car created
Car destroyed
Engine destroyed
```

Açıklama:

- `Car` nesnesi oluşturulduğunda (`Car myCar;`), önce `Engine` nesnesi oluşturulur çünkü `Engine` `Car` nesnesinin bir parçasıdır. Bu nedenle, `Engine` constructor'ı önce çağrılır ve "Engine created" mesajı yazdırılır.
- Daha sonra, `Car` constructor'ı çağrılır ve "Car created" mesajı yazdırılır.
- Program sonlandığında, `myCar` nesnesi kapsamdan çıktığında, önce `Car` destructor'ı çağrılır ve "Car destroyed" mesajı yazdırılır.
- Son olarak, `Engine` destructor'ı çağrılır ve "Engine destroyed" mesajı yazdırılır.

Constructors and Destructors in Composition with Parameters

Aşağıda, bileşen nesneler için parametreli constructor'ları kullanan ve ana sınıfın constructor'ında bu parametreleri sağlayan bir örnek bulunmaktadır:

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

// Bileşen Sınıf
```

```
class Engine {
public:
    std::string type;

    Engine(const std::string& engineType) : type(engineType) {
        std::cout << "Engine created with type: " << type << std::endl;
    }

    ~Engine() {
        std::cout << "Engine destroyed" << std::endl;
    }
};

// Ana Sınıf
class Car {
private:
    std::string brand;
    Engine engine;  // Composition: Car nesnesi bir Engine nesnesine sahipt
ir

public:
    Car(const std::string& carBrand, const std::string& engineType)
        : brand(carBrand), engine(engineType) {  // engine için parametre i
letilir
        std::cout << "Car created with brand: " << brand << std::endl;
    }

    ~Car() {
        std::cout << "Car destroyed" << std::endl;
    }
};

int main() {
    Car myCar("Toyota", "V8");  // Car nesnesi oluşturuluyor
    return 0;
}
```

Açıklama:

- `Engine` sınıfı, `std::string` türünde bir parametre alan bir constructor tanımlar ve bu parametreyi kullanarak motor tipini başlatır.
- `Car` sınıfı, `Engine` nesnesini içeren bir bileşen olarak tanımlar. `Car` sınıfının constructor'ı, hem `brand` hem de `engine` için parametreler alır ve bu parametreleri bileşen nesnesine iletir.
- `main` fonksiyonunda, `Car` nesnesi `myCar` oluşturulur ve uygun parametreler iletilir.

Program çalıştırıldığında, aşağıdaki çıktı alınır:

```
Engine created with type: V8
Car created with brand: Toyota
Car destroyed
Engine destroyed
```

Açıklama:

1. `Car myCar("Toyota", "V8");` ifadesi yürütüldüğünde, önce `Engine` constructor'ı çağrılır ve "Engine created with type: V8" mesajı yazdırılır.
2. Daha sonra, `Car` constructor'ı çağrılır ve "Car created with brand: Toyota" mesajı yazdırılır.
3. Program sonlandığında ve `myCar` kapsamdan çıktığında, önce `Car` destructor'ı çağrılır ve "Car destroyed" mesajı yazdırılır.
4. Son olarak, `Engine` destructor'ı çağrılır ve "Engine destroyed" mesajı yazdırılır.

Bu örnek, bileşen nesneler için parametreli constructor'ların ana sınıf constructor'ında nasıl kullanılacağını ve bileşen nesnelerin yaşam döngüsünün ana sınıfın yaşam döngüsüne nasıl bağlı olduğunu gösterir.

Dinamik Üye Nesneleri ve Destructor Kullanımı

Aşağıda, dinamik üye nesnelerinin kullanıldığı bir C++ örneği bulunmaktadır:

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

// Engine Sınıfı
class Engine {
public:
    std::string type;

    Engine(const std::string& engineType) : type(engineType) {
        std::cout << "Engine created with type: " << type << std::endl;
    }

    ~Engine() {
        std::cout << "Engine destroyed" << std::endl;
    }

    void start() const {
        std::cout << "Engine of type " << type << " is starting..." << st
d::endl;
    }
};

// Car Sınıfı
class Car {
private:
    std::string brand;
```

```
    Engine* engine;  // Dinamik olarak oluşturulan Engine nesnesi için poin
ter

public:
    // Constructor
    Car(const std::string& carBrand, const std::string& engineType) : brand
(carBrand) {
        engine = new Engine(engineType);  // Dinamik olarak Engine nesnesi
oluştur
        std::cout << "Car created with brand: " << brand << std::endl;
    }

    // Destructor
    ~Car() {
        delete engine;  // Dinamik olarak oluşturulan Engine nesnesini serb
est bırak
        std::cout << "Car destroyed" << std::endl;
    }

    void startCar() const {
        std::cout << "Starting car " << brand << std::endl;
        engine->start();
    }
};

int main() {
    Car myCar("Toyota", "V8");  // Car nesnesi oluşturuluyor
    myCar.startCar();  // Aracı başlat

    return 0;
}
```

Açıklama:

1. **Engine Sınıfı:**
   - `Engine` sınıfı, bir `std::string` türünde `type` üyesine sahiptir ve parametreli bir constructor ile bu üye başlatılır.
   - `Engine` sınıfı ayrıca bir destructor tanımlar ve bir `start` metodu içerir.
2. **Car Sınıfı:**
   - `Car` sınıfı, `brand` ve `engine` isimli iki üyeye sahiptir. `engine` üyesi, dinamik olarak oluşturulan bir `Engine` nesnesini işaret eden bir pointer'dır.
   - `Car` sınıfının constructor'ı, `brand` üyesini başlatır ve `engine` pointer'ına dinamik olarak oluşturulan bir `Engine` nesnesini atar (`new` operatörü kullanılarak).
   - `Car` sınıfının destructor'ı, dinamik olarak oluşturulan `Engine` nesnesini serbest bırakır (`delete` operatörü kullanılarak).

- `startCar` metodu, aracı başlatmak için `engine` nesnesinin `start` metodunu çağırır.

Program Çalıştırma:

Program çalıştırıldığında, aşağıdaki çıktılar alınır:

```
Engine created with type: V8
Car created with brand: Toyota
Starting car Toyota
Engine of type V8 is starting...
Car destroyed
Engine destroyed
```

Dinamik Bellek Yönetimi:

- `Car` nesnesi oluşturulduğunda (`Car myCar("Toyota", "V8");`), `Car` constructor'ı çağrılır ve `Engine` nesnesi dinamik olarak oluşturulur. Bu, "Engine created with type: V8" ve "Car created with brand: Toyota" mesajlarını üretir.
- `myCar.startCar();` çağrıldığında, `Car` nesnesinin `startCar` metodu çalışır ve bu, "Starting car Toyota" ve "Engine of type V8 is starting..." mesajlarını üretir.
- Program sonlandığında ve `myCar` kapsamdan çıktığında, `Car` destructor'ı çağrılır ve dinamik olarak oluşturulan `Engine` nesnesi serbest bırakılır. Bu, "Car destroyed" ve "Engine destroyed" mesajlarını üretir.

OOP07

Inheritance

Inheritance (kalıtım), nesne yönelimli programlamanın temel özelliklerinden biridir ve bir sınıfın (türetilmiş sınıf veya alt sınıf), başka bir sınıfın (taban sınıf veya üst sınıf) özelliklerini ve davranışlarını devralmasını sağlar. Kalıtım, kodun yeniden kullanılabilirliğini artırır ve sınıflar arasında hiyerarşik ilişkiler kurmaya olanak tanır.

Kalıtım Türleri

C++'ta kalıtım genellikle üç şekilde yapılır:

1. **Public Inheritance:** Taban sınıfın public ve protected üyeleri, türetilmiş sınıfta sırasıyla public ve protected olarak kalır.
2. **Protected Inheritance:** Taban sınıfın public ve protected üyeleri, türetilmiş sınıfta protected hale gelir.
3. **Private Inheritance:** Taban sınıfın public ve protected üyeleri, türetilmiş sınıfta private hale gelir.

Örnek: Public Inheritance

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

```
// Taban Sınıf
class Vehicle {
protected:
    std::string brand;

public:
    Vehicle(const std::string& b) : brand(b) {}

    void showBrand() const {
        std::cout << "Brand: " << brand << std::endl;
    }
};

// Türetilmiş Sınıf
class Car : public Vehicle {
private:
    std::string model;

public:
    Car(const std::string& b, const std::string& m) : Vehicle(b), model(m)
{}

    void showModel() const {
        std::cout << "Model: " << model << std::endl;
    }

    // Taban sınıfın fonksiyonunu override etme
    void showDetails() const {
        showBrand();
        showModel();
    }
};

int main() {
    Car myCar("Toyota", "Corolla");
    myCar.showDetails();  // Hem markayı hem de modeli gösterir

    return 0;
}
```

Açıklama:

1. **Vehicle Sınıfı:**
   - `Vehicle` sınıfı, bir `brand` üyesine sahiptir ve bu üye `protected` olarak tanımlanmıştır. Bu, `brand` üyesinin `Vehicle` sınıfından türetilen sınıflarda doğrudan erişilebilir olmasını sağlar.
   - `showBrand` adlı bir public fonksiyon, aracın markasını ekrana yazdırır.
2. **Car Sınıfı:**

- `Car` sınıfı, `Vehicle` sınıfından public olarak türetilmiştir.
- `Car` sınıfı, ek olarak `model` adlı bir private üyeye sahiptir ve bu üye aracın modelini temsil eder.
- `showModel` adlı bir public fonksiyon, aracın modelini ekrana yazdırır.
- `showDetails` adlı bir public fonksiyon, hem `showBrand` hem de `showModel` fonksiyonlarını çağırarak aracın marka ve modelini ekrana yazdırır.

3. **main Fonksiyonu:**
   - `Car` sınıfından bir `myCar` nesnesi oluşturulur ve constructor ile markası "Toyota" ve modeli "Corolla" olarak belirlenir.
   - `myCar.showDetails()` çağrısı ile aracın marka ve modeli ekrana yazdırılır.

Program Çıktısı:

```
Brand: Toyota
Model: Corolla
```

Örnek: Protected Inheritance

```
#include <iostream>

// Taban Sınıf
class Vehicle {
protected:
    int speed; // protected üye

public:
    Vehicle(int s) : speed(s) {}

    void showSpeed() const {
        std::cout << "Speed: " << speed << " km/h" << std::endl;
    }
};

// Türetilmiş Sınıf
class Car : protected Vehicle { // protected kalıtım
private:
    std::string brand;

public:
    Car(const std::string& b, int s) : Vehicle(s), brand(b) {}

    void showDetails() const {
        std::cout << "Brand: " << brand << std::endl;
        showSpeed(); // protected üyeye türetilmiş sınıf içerisinden erişim
    }
```

```
};

int main() {
    Car myCar("Toyota", 120);
    myCar.showDetails(); // Car sınıfının üyesi olan showDetails() çağrılır

    return 0;
}
```

Açıklama:

1. **Vehicle Sınıfı:**
   - `speed` adında bir `protected` üye tanımlanmıştır.
   - `showSpeed` adında bir `public` fonksiyon, aracın hızını ekrana yazdırır.
2. **Car Sınıfı:**
   - `Car` sınıfı, `Vehicle` sınıfından `protected` kalıtım alır. Bu, `speed` üyesinin `Car` sınıfı ve `Car` sınıfından türetilmiş sınıflar tarafından erişilebilir olduğu anlamına gelir.
   - `showDetails` adında bir fonksiyon, aracın markasını ve hızını ekrana yazdırır. `showSpeed` fonksiyonuna `Car` sınıfının üyesi olarak erişilir.
3. **main Fonksiyonu:**
   - `Car` sınıfından bir `myCar` nesnesi oluşturulur ve hızı 120 olarak belirlenir.
   - `myCar.showDetails()` çağrısı ile aracın markası ve hızı ekrana yazdırılır.

Program Çıktısı:

```
Brand: Toyota
Speed: 120 km/h
```

Bu örnekte, `Car` sınıfı `Vehicle` sınıfından `protected` kalıtım aldığı için `speed` üyesine erişim sağlanabiliyor ve bu üye `showSpeed` fonksiyonu aracılığıyla ekrana yazdırılıyor. Ancak `speed` üyesine `main` fonksiyonundan doğrudan erişim mümkün değildir.

Örnek: Private Inheritance

Taban sınıfın üyeleri türetilmiş sınıfa `private` olarak kalıtıldığında, bu üyelere sadece türetilmiş sınıfın kendi üyeleri erişebilir. Dışarıdan erişim engellenir, bu da türetilmiş sınıfın taban sınıfın bir örneğini içermesine rağmen, genel olarak bir "implementasyon detayı" olarak kabul edilir.
Örnek:

```
#include <iostream>

// Taban Sınıf
class Base {
private:
    int value;
```

```
public:
    Base(int v) : value(v) {}

    void showValue() const {
        std::cout << "Value: " << value << std::endl;
    }
};

// Türetilmiş Sınıf
class Derived : private Base { // private kalıtım
private:
    std::string name;

public:
    Derived(const std::string& n, int v) : Base(v), name(n) {}

    void showDetails() const {
        std::cout << "Name: " << name << std::endl;
        showValue(); // private üyeye türetilmiş sınıf içerisinden erişim
    }
};

int main() {
    Derived derivedObj("Example", 42);
    derivedObj.showDetails();

    return 0;
}
```

Açıklama:

1. **Base Sınıfı:**
   - `value` adında bir `private` üye tanımlanmıştır.
   - `showValue` adında bir `public` fonksiyon, değeri ekrana yazdırır.
2. **Derived Sınıfı:**
   - `Derived` sınıfı, `Base` sınıfından `private` kalıtım alır. Bu, `value` üyesinin `Derived` sınıfı ve `Derived` sınıfından türetilmiş sınıflar tarafından erişilebilir olduğu anlamına gelir.
   - `showDetails` adında bir fonksiyon, nesnenin adını ve değerini ekrana yazdırır. `showValue` fonksiyonuna `Derived` sınıfının üyesi olarak erişilir.
3. **main Fonksiyonu:**
   - `Derived` sınıfından bir `derivedObj` nesnesi oluşturulur ve değeri 42 olarak belirlenir.
   - `derivedObj.showDetails()` çağrısı ile nesnenin adı ve değeri ekrana yazdırılır.

Program Çıktısı:

```
Name: Example
Value: 42
```

Bu örnekte, `Derived` sınıfı `Base` sınıfından `private` kalıtım aldığı için `value` üyesine yalnızca `Derived` sınıfı tarafından erişim sağlanabilir. Dışarıdan erişim engellendiği için `main` fonksiyonundan doğrudan `value` üyesine erişim mümkün değildir.

Kalıtımın Avantajları:

1. **Kodun Yeniden Kullanımı:** Taban sınıftaki üyeler ve fonksiyonlar türetilmiş sınıflar tarafından yeniden kullanılabilir.
2. **Bakım Kolaylığı:** Ortak özellikler ve davranışlar tek bir yerde (taban sınıfta) tanımlanarak bakım ve güncellemeler kolaylaşır.
3. **Genişletilebilirlik:** Yeni sınıflar, mevcut sınıflardan türetilerek kolayca genişletilebilir.

Çok Biçimlilik (Polymorphism) ve Sanal Fonksiyonlar

Kalıtım aynı zamanda çok biçimlilik (polymorphism) ile birlikte kullanılarak dinamik davranışların gerçekleştirilmesini sağlar. Bunun için genellikle taban sınıf fonksiyonları `virtual` olarak tanımlanır.

Aşağıda, sanal fonksiyonların kullanıldığı bir örnek verilmiştir:

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

// Taban Sınıf
class Vehicle {
public:
    virtual void showDetails() const {
        std::cout << "This is a vehicle" << std::endl;
    }
};

// Türetilmiş Sınıf
class Car : public Vehicle {
private:
    std::string brand;
    std::string model;

public:
    Car(const std::string& b, const std::string& m) : brand(b), model(m) {}

    void showDetails() const override {
        std::cout << "Brand: " << brand << ", Model: " << model << std::end
l;
    }
};

// Türetilmiş Sınıf
```

```
class Bike : public Vehicle {
private:
    std::string brand;

public:
    Bike(const std::string& b) : brand(b) {}

    void showDetails() const override {
        std::cout << "Brand: " << brand << std::endl;
    }
};

void displayVehicleDetails(const Vehicle& vehicle) {
    vehicle.showDetails();
}

int main() {
    Car myCar("Toyota", "Corolla");
    Bike myBike("Yamaha");

    displayVehicleDetails(myCar);
    displayVehicleDetails(myBike);

    return 0;
}
```

Açıklama:

1. **Vehicle Sınıfı:**
   - `showDetails` fonksiyonu `virtual` olarak tanımlanmıştır, böylece türetilmiş sınıflarda override edilebilir.
2. **Car ve Bike Sınıfları:**
   - `Car` ve `Bike` sınıfları, `Vehicle` sınıfından türetilmiştir ve kendi `showDetails` fonksiyonlarını override ederler.
3. **displayVehicleDetails Fonksiyonu:**
   - `displayVehicleDetails` fonksiyonu, bir `Vehicle` referansı alır ve `showDetails` fonksiyonunu çağırır.
4. **main Fonksiyonu:**
   - `Car` ve `Bike` nesneleri oluşturulur ve `displayVehicleDetails` fonksiyonuna geçilir.
   - Dinamik olarak doğru `showDetails` fonksiyonları çağrılır.

Program Çıktısı:

```
Brand: Toyota, Model: Corolla
Brand: Yamaha
```

Bu örnek, kalıtım ve çok biçimliliği (polymorphism) birlikte kullanarak dinamik davranışların nasıl gerçekleştirilebileceğini göstermektedir. Kalıtım, kodun yeniden kullanılabilirliğini ve genişletilebilirliğini artırırken, sanal fonksiyonlar ve çok biçimlilik, farklı türetilmiş sınıfların uygun davranışlarını dinamik olarak çağırmayı sağlar.

Redefining (Overriding) the Members of the Base (Name Hiding)

Redefining veya overriding, türetilmiş sınıfın taban sınıftan (üst sınıf) aldığı bir üyenin (fonksiyon veya değişken) aynı isimdeki bir üye tarafından yeniden tanımlanmasıdır. C++'ta, bir fonksiyonu veya veri üyesini aynı isimle türetilmiş sınıfta yeniden tanımlamak, o üyenin taban sınıftaki tüm aşırı yüklemelerini (overloads) gizler.

Örnek:

```
#include <iostream>

// Taban Sınıf
class Base {
public:
    void display() const {
        std::cout << "Base display() called" << std::endl;
    }
};

// Türetilmiş Sınıf
class Derived : public Base {
public:
    void display() const {
        std::cout << "Derived display() called" << std::endl;
    }
};

int main() {
    Derived derivedObj;
    derivedObj.display(); // Türetilmiş sınıfın üye fonksiyonu çağrılır

    return 0;
}
```

Açıklama:

- `Base` sınıfında `display` adında bir fonksiyon tanımlanmıştır ve bu fonksiyonun içeriği "Base display() called" mesajını ekrana yazdırır.
- `Derived` sınıfı, `Base` sınıfından `public` kalıtım alır ve `display` adında aynı isimde bir fonksiyonu yeniden tanımlar (override). Yani, `Derived` sınıfında `display` fonksiyonu `Base` sınıfındaki `display` fonksiyonunu gizler.
- `main` fonksiyonunda, `Derived` sınıfından bir nesne oluşturulur ve `display` fonksiyonu çağrılır. Bu, `Derived` sınıfındaki `display` fonksiyonunu çağırır.

Program Çıktısı:

```
Derived display() called
```

Gördüğünüz gibi, türetilmiş sınıfın üye fonksiyonu çağrıldığında, türetilmiş sınıftaki (Derived) fonksiyon çağrılır. Taban sınıftaki aynı isimli fonksiyon görünmez hale gelir (name hiding). Bu durum, C++ dilinde sık rastlanan bir durumdur ve dikkatle yönetilmelidir. Eğer taban sınıftaki `display` fonksiyonunu da çağırmak istiyorsak, türetilmiş sınıftaki `display` fonksiyonu içinde açıkça taban sınıfındaki `display` fonksiyonu çağrılmalıdır.

```
#include <iostream>

// Taban Sınıf
class Base {
public:
    void display() const {
        std::cout << "Base display() called" << std::endl;
    }
};

// Türetilmiş Sınıf
class Derived : public Base {
public:
    void display() const {
        std::cout << "Derived display() called" << std::endl;
        Base::display(); // Taban sınıftaki display fonksiyonunu çağır
    }
};

int main() {
    Derived derivedObj;
    derivedObj.display(); // Türetilmiş sınıfın üye fonksiyonu çağrılır

    return 0;
}
```

Açıklama:

- `Base` sınıfında `display` adında bir fonksiyon tanımlanmıştır ve bu fonksiyonun içeriği "Base display() called" mesajını ekrana yazdırır.
- `Derived` sınıfı, `Base` sınıfından `public` kalıtım alır ve `display` adında aynı isimde bir fonksiyonu yeniden tanımlar (override). Ayrıca, `Derived` sınıfının `display` fonksiyonu içinde `Base::display()` çağrısı yapılarak taban sınıftaki `display` fonksiyonu açıkça çağrılır.
- `main` fonksiyonunda, `Derived` sınıfından bir nesne oluşturulur ve `display` fonksiyonu çağrılır. Bu, `Derived` sınıfındaki `display` fonksiyonunu çağırır ve içindeki `Base::display()` çağrısı ile taban sınıftaki `display` fonksiyonu da çağrılır.

Program Çıktısı:

```
Derived display() called
Base display() called
```

Bu şekilde, türetilmiş sınıftaki özel işlemlerin yanı sıra, taban sınıftaki işlemleri de çağırabilir ve gerektiğinde kullanabiliriz.

During overriding, the parameters of the Base methods can be changed:

Eğer bir sınıfın bir metodunu türetilmiş sınıfta yeniden tanımlamak istiyorsak, bunu `virtual` anahtar kelimesi olmadan yapabiliriz. Ancak, bu durumda, eğer taban sınıftaki metot `virtual` olarak tanımlanmamışsa, türetilmiş sınıftaki metot taban sınıfınkini gizler ve bu durum işlevsel çok biçimlilik (functional polymorphism) sağlamaz.

Aşağıdaki örnekte, `virtual` anahtar kelimesi olmadan türetilmiş sınıfta bir metot yeniden tanımlanmıştır:

```
#include <iostream>

// Taban Sınıf
class Base {
public:
    void show(int value) const {
        std::cout << "Base show(int) called: " << value << std::endl;
    }
};

// Türetilmiş Sınıf
class Derived : public Base {
public:
    // Base sınıftaki show fonksiyonunu gizler
    void show(double value) const {
        std::cout << "Derived show(double) called: " << value << std::endl;
    }
};

int main() {
    Derived derivedObj;
    derivedObj.show(3.14); // Türetilmiş sınıfın show(double) fonksiyonu ça
ğrılır

    return 0;
}
```

Açıklama:

- `Base` sınıfında `show` adında bir fonksiyon tanımlanmıştır ve `int` tipinde bir parametre alır.

- `Derived` sınıfı, `Base` sınıfından `public` kalıtım alır ve `show` fonksiyonunu `double` türünde bir parametreyle yeniden tanımlar (redefine eder).
- `main` fonksiyonunda, `Derived` sınıfından bir nesne oluşturulur ve `show` fonksiyonu çağrılır. Bu, `Derived` sınıfındaki `show` fonksiyonunu çağırır ve `double` türündeki parametre kullanılarak işlem gerçekleştirilir.

Program Çıktısı:

```
Derived show(double) called: 3.14
```

Bu durumda, taban sınıftaki metot türetilmiş sınıfta gizlenir ve işlevsel çok biçimlilik sağlanmaz. Çünkü taban sınıftaki metot `virtual` olarak tanımlanmadığı için, türetilmiş sınıftaki metot taban sınıfındakini override etmez, onu gizler. Bu nedenle, eğer işlevsel çok biçimlilik elde etmek istiyorsak, taban sınıftaki metotları `virtual` olarak tanımlamalıyız ve türetilmiş sınıfta `override` anahtar kelimesini kullanarak yeniden tanımlamalıyız.

Overloading and Name Hiding (Overriding)

Overloading ve overriding, C++ dilinde önemli kavramlardır. Her ikisi de çok biçimlilik (polymorphism) özelliklerini sağlar, ancak farklı şekillerde çalışırlar.

Overloading

Overloading, aynı isimdeki farklı fonksiyonların veya metotların tanımlanmasıdır. Bu farklı fonksiyonlar veya metotlar, parametre sayıları, türleri veya sıraları bakımından farklılık gösterebilirler. Derleyici, çağrılan fonksiyonun veya metotun hangi versiyonunu çağıracağını, kullanılan argümanlara ve bu argümanların türlerine göre belirler.

```
#include <iostream>

void print(int x) {
    std::cout << "int: " << x << std::endl;
}

void print(double x) {
    std::cout << "double: " << x << std::endl;
}

int main() {
    print(5);       // int: 5
    print(3.14);    // double: 3.14

    return 0;
}
```

Yukarıdaki örnekte, `print` fonksiyonu hem `int` hem de `double` parametreleri alabilen iki farklı versiyonu ile aşırı yüklenmiştir. Derleyici, hangi versiyonun çağrılacağını, kullanılan argümanların türlerine göre belirler.

Overriding

Overriding, bir alt sınıfta taban sınıfta tanımlanan bir metodu yeniden tanımlamaktır. Bu, türetilmiş sınıfın, taban sınıftaki metodu yeniden tanımlayarak, aynı imzaya sahip bir metodu gizlemesini sağlar. Böylece, türetilmiş sınıfta bu metodu çağırdığınızda, türetilmiş sınıftaki sürümü çağrılır.

```
#include <iostream>

// Taban Sınıf
class Base {
public:
    virtual void display() const {
        std::cout << "Base display() called" << std::endl;
    }
};

// Türetilmiş Sınıf
class Derived : public Base {
public:
    void display() const override {
        std::cout << "Derived display() called" << std::endl;
    }
};

int main() {
    Derived derivedObj;
    derivedObj.display(); // Derived display() called

    return 0;
}
```

Yukarıdaki örnekte, `Base` sınıfındaki `display` fonksiyonu `virtual` olarak tanımlanmıştır. `Derived` sınıfı bu fonksiyonu override eder. Bu nedenle, `Derived` sınıfından bir nesne oluşturulduğunda, `display` fonksiyonu çağrıldığında, türetilmiş sınıftaki sürümü çağrılır.

Overriding'de, name hiding gibi bir durum vardır. Eğer taban sınıftaki metot `virtual` olarak işaretlenmemişse, türetilmiş sınıftaki aynı adlı metot taban sınıftakinin üzerini kapatır ve çok biçimlilik özelliği sağlanmaz. Bu durumu önlemek için taban sınıftaki metotlar `virtual` olarak tanımlanmalı ve türetilmiş sınıftaki metotlar `override` anahtar kelimesiyle yeniden tanımlanmalıdır.

Constructors and Destructors in Inheritance

Bu örnekte, hem taban sınıfta hem de türetilmiş sınıfta kendi constructors ve destructors'larını tanımlayan bir senaryo gösterilmiştir.

```
#include <iostream>

// Taban Sınıf
class Base {
```

```
public:
    Base() {
        std::cout << "Base constructor called" << std::endl;
    }

    ~Base() {
        std::cout << "Base destructor called" << std::endl;
    }
};

// Türetilmiş Sınıf
class Derived : public Base {
public:
    Derived() {
        std::cout << "Derived constructor called" << std::endl;
    }

    ~Derived() {
        std::cout << "Derived destructor called" << std::endl;
    }
};

int main() {
    Derived derivedObj; // Nesne oluşturulduğunda constructorlar çağrılır
    // Nesnenin kapsamı bittiğinde destructorlar çağrılır
    return 0;
}
```

Açıklama:

- `Base` sınıfında, `Base` constructor ve destructor'ı tanımlanmıştır. Constructor nesne oluşturulduğunda, destructor ise nesnenin kapsamı bittiğinde çağrılır.
- `Derived` sınıfı, `Base` sınıfından kalıtım alır ve kendi constructor ve destructor'larını tanımlar.
- `main` fonksiyonunda, `Derived` sınıfından bir nesne (`derivedObj`) oluşturulur. Bu, hem `Base` hem de `Derived` constructor'larını çağırır. Nesnenin kapsamı sona erdiğinde, destructorlar çağrılır. Bu durum, `Derived` sınıfından kalıtım alınan `Base` sınıfının destructor'ını da içerir.

Program Çıktısı:

```
Base constructor called
Derived constructor called
Derived destructor called
Base destructor called
```

Bu çıktı, nesnenin oluşturulması ve sona erdirilmesi sırasında constructor ve destructor'ların çağrılma sırasını gösterir. İlk olarak, `Base` constructor çağrılır, ardından `Derived` constructor. Nesnenin kapsamı bittiğinde ise bu sıra tersine döner: önce `Derived` destructor, sonra `Base` destructor çağrılır. Bu, destructors'ların alt sınıftan üst sınıfa doğru çağrıldığını gösterir.

Constructors with parameters

Constructors parametre alabilir ve parametreli kurucular sınıfın örneklerini oluşturmak için belirli değerlerle çağrılabilir. Bu örnekte, hem taban sınıfta hem de türetilmiş sınıfta parametreli constructors kullanarak inheritance gösterilmiştir.

```
#include <iostream>

// Taban Sınıf
class Base {
private:
    int baseValue;

public:
    // Parametreli constructor
    Base(int value) : baseValue(value) {
        std::cout << "Base constructor called with value: " << baseValue <<
std::endl;
    }

    ~Base() {
        std::cout << "Base destructor called" << std::endl;
    }

    void showBaseValue() const {
        std::cout << "Base value: " << baseValue << std::endl;
    }
};

// Türetilmiş Sınıf
class Derived : public Base {
private:
    int derivedValue;

public:
    // Parametreli constructor
    Derived(int baseVal, int derivedVal) : Base(baseVal), derivedValue(deri
vedVal) {
        std::cout << "Derived constructor called with values: " << baseVal
<< ", " << derivedVal << std::endl;
    }

    ~Derived() {
        std::cout << "Derived destructor called" << std::endl;
    }

    void showDerivedValue() const {
```

```
        std::cout << "Derived value: " << derivedValue << std::endl;
    }
};

int main() {
    Derived derivedObj(10, 20); // Parametreli constructor çağrıldı
    derivedObj.showBaseValue(); // Base sınıfının metodu
    derivedObj.showDerivedValue(); // Derived sınıfının metodu

    return 0;
}
```

Açıklama:

- `Base` sınıfında, `int` tipinde bir parametre alan bir constructor tanımlanmıştır. Bu constructor, `baseValue` üyesini parametre değeriyle başlatır.
- `Derived` sınıfında, `Base` sınıfından kalıtım alınır ve `int` tipinde iki parametre alan bir constructor tanımlanır. Bu constructor, hem `Base` sınıfının constructor'ını hem de `derivedValue` üyesini başlatır.
- `main` fonksiyonunda, `Derived` sınıfından bir nesne (`derivedObj`) oluşturulurken, parametreli constructor çağrılır. Bu, `Base` sınıfındaki constructor'ı ve `Derived` sınıfındaki constructor'ı çağırır. Sonrasında her iki sınıfın da metotları çağrılarak ilgili değerler ekrana yazdırılır.

Program Çıktısı:

```
Base constructor called with value: 10
Derived constructor called with values: 10, 20
Base value: 10
Derived value: 20
Derived destructor called
Base destructor called
```

Bu çıktı, her iki sınıfın constructor'larının sırasıyla çağrılmasını, ardından destructor'ların çağrılmasını gösterir.

Constructos and Destructors in the inheritance with composition

```
#include <iostream>

// Bileşen Sınıf
class Component {
private:
    int componentValue;

public:
    // Parametreli constructor
```

```
    Component(int value) : componentValue(value) {
        std::cout << "Component constructor called with value: " << compone
ntValue << std::endl;
    }

    ~Component() {
        std::cout << "Component destructor called" << std::endl;
    }

    void showComponentValue() const {
        std::cout << "Component value: " << componentValue << std::endl;
    }
};

// Taban Sınıf
class Base {
private:
    Component component;

public:
    // Parametreli constructor
    Base(int baseValue, int componentValue) : component(componentValue) {
        std::cout << "Base constructor called with value: " << baseValue <<
std::endl;
    }

    ~Base() {
        std::cout << "Base destructor called" << std::endl;
    }

    void showBaseValue() const {
        std::cout << "Base value: " << std::endl;
        component.showComponentValue(); // Bileşen sınıfın metodu
    }
};

int main() {
    Base baseObj(10, 20); // Parametreli constructor çağrıldı
    baseObj.showBaseValue(); // Base sınıfının metodu

    return 0;
}
```

Açıklama:

- `Component` sınıfı, `int` tipinde bir parametre alan bir constructor ve bir destructor ile bir `showComponentValue` metodu içerir.

- `Base` sınıfı, `Component` sınıfını bileşen olarak içerir ve `int` tipinde iki parametre alan bir constructor ve bir destructor ile bir `showBaseValue` metodu içerir.
- `main` fonksiyonunda, `Base` sınıfından bir nesne (`baseObj`) oluşturulurken, parametreli constructor çağrılır. Bu, `Component` sınıfının constructor'ını da çağırarak bileşen nesnesini başlatır. Sonrasında `showBaseValue` metodu çağrılarak bileşen sınıfın metodu da çağrılır.

Program Çıktısı:

```
Component constructor called with value: 20
Base constructor called with value: 10
Base value:
Component value: 20
Base destructor called
Component destructor called
```

Bu çıktı, her iki sınıfın constructor'larının sırasıyla çağrılmasını, ardından destructor'ların sırasıyla çağrılmasını gösterir. Böylece, bileşen sınıfın ömrü, içeren sınıfın ömrüne bağlı olarak yönetilir.

Inheriting Constructors

Diyelim ki bir taban sınıfı var ve bu sınıfta sadece bir parametre alan bir constructor tanımlanmış olsun. Türetilmiş sınıfta, taban sınıfın bu constructor'ını kullanarak kendi constructor'ını oluşturmak isteniyor. İşte bu durumda "inheriting constructors" özelliği kullanılabilir.

```
#include <iostream>

// Taban Sınıf
class Base {
public:
    Base(int value) {
        std::cout << "Base constructor called with value: " << value << st
d::endl;
    }
};

// Türetilmiş Sınıf
class Derived : public Base {
public:
    using Base::Base; // Taban sınıfın constructor'ını miras al

    void display() const {
        std::cout << "Derived display() called" << std::endl;
    }
};

int main() {
    Derived derivedObj(10); // Parametreli constructor çağrıldı
```

```
    derivedObj.display(); // Derived sınıfının metodu

    return 0;
}
```

Açıklama:

- `Base` sınıfında, sadece bir parametre alan bir constructor tanımlanmıştır.
- `Derived` sınıfında, `using Base::Base;` ifadesi ile taban sınıfın constructor'ını miras alır. Bu, `Derived` sınıfının constructor'ı olmadan da, `Base` sınıfının constructor'ını kullanarak bir `Derived` nesnesi oluşturulabileceği anlamına gelir.
- `main` fonksiyonunda, `Derived` sınıfından bir nesne (`derivedObj`) oluşturulurken, parametreli constructor çağrılır. Bu, taban sınıfın constructor'ını çağırarak taban sınıfın bir örneğini oluşturur.
- `display` metodu çağrıldığında, bu metot `Derived` sınıfına aittir ve taban sınıfı ile doğrudan bir ilgisi yoktur.

Program Çıktısı:

```
Base constructor called with value: 10
Derived display() called
```

Bu çıktı, taban sınıfın constructor'ının çağrıldığını ve ardından türetilmiş sınıfın metotunun çağrıldığını gösterir.

The Copy Constructor under Inheritance

Kopyalama constructor'u (copy constructor), bir sınıfın bir başka nesnesinden bir kopya oluşturmak için kullanılır. Eğer bir sınıfın kopyalama constructor'u tanımlanmazsa, derleyici tarafından varsayılan bir kopyalama constructor'u oluşturulur. Kalıtım (inheritance) durumunda, türetilmiş sınıfın kopyalama constructor'u taban sınıfın kopyalama constructor'unu çağırmak zorundadır. Örnek:

```
#include <iostream>

// Taban Sınıf
class Base {
private:
    int baseValue;

public:
    // Parametreli constructor
    Base(int value) : baseValue(value) {
        std::cout << "Base constructor called with value: " << baseValue <<
std::endl;
    }

    // Kopyalama constructor'u
```

```
    Base(const Base& other) : baseValue(other.baseValue) {
        std::cout << "Base copy constructor called" << std::endl;
    }

    ~Base() {
        std::cout << "Base destructor called" << std::endl;
    }

    void showBaseValue() const {
        std::cout << "Base value: " << baseValue << std::endl;
    }
};

// Türetilmiş Sınıf
class Derived : public Base {
private:
    int derivedValue;

public:
    // Parametreli constructor
    Derived(int baseVal, int derivedVal) : Base(baseVal), derivedValue(deri
vedVal) {
        std::cout << "Derived constructor called with values: " << baseVal
<< ", " << derivedVal << std::endl;
    }

    // Kopyalama constructor'u
    Derived(const Derived& other) : Base(other), derivedValue(other.derived
Value) {
        std::cout << "Derived copy constructor called" << std::endl;
    }

    ~Derived() {
        std::cout << "Derived destructor called" << std::endl;
    }

    void showDerivedValue() const {
        std::cout << "Derived value: " << derivedValue << std::endl;
    }
};

int main() {
    Derived derivedObj1(10, 20); // Parametreli constructor çağrıldı
    Derived derivedObj2 = derivedObj1; // Kopyalama constructor çağrıldı
    derivedObj2.showBaseValue(); // Base sınıfının metodu
    derivedObj2.showDerivedValue(); // Derived sınıfının metodu
```

```
    return 0;
}
```

Açıklama:

- `Base` sınıfında, bir parametre alan bir constructor ve bir kopyalama constructor'u tanımlanmıştır.
- `Derived` sınıfında, `Base` sınıfından kalıtım alınmıştır. `Derived` sınıfında, bir parametre alan bir constructor ve bir kopyalama constructor'u tanımlanmıştır. `Derived` sınıfının kopyalama constructor'u, taban sınıfın kopyalama constructor'unu çağırmak zorundadır.
- `main` fonksiyonunda, `Derived` sınıfından bir nesne (`derivedObj1`) oluşturulurken, parametreli constructor çağrılır. Daha sonra, bu nesnenin bir kopyası (`derivedObj2`) oluşturulurken, kopyalama constructor çağrılır. Hem taban sınıfın hem de türetilmiş sınıfın kopyalama constructor'ları çağrılır.
- Nesnelerin kapsamı dışına çıktığında destructor'lar çağrılır.

Program Çıktısı:

```
Base constructor called with value: 10
Derived constructor called with values: 10, 20
Base copy constructor called
Derived copy constructor called
Base value: 10
Derived value: 20
Derived destructor called
Base destructor called
Derived destructor called
Base destructor called
```

Bu çıktı, her iki sınıfın constructor'larının ve kopyalama constructor'larının sırasıyla çağrılmasını ve nesnelerin kapsamı dışına çıktığında destructor'ların çağrılmasını gösterir.

Inheriting From the Library

"Inheriting from the library" konusu, standart kütüphaneden veya herhangi bir üçüncü parti kütüphaneden gelen sınıfları miras alarak, bu sınıfların yeteneklerini genişletmek veya özelleştirmek anlamına gelir. Bu konsepti daha iyi anlamak için, standart C++ kütüphanesinden gelen bir sınıf olan `std::vector` sınıfını miras alan bir sınıf oluşturabiliriz.

Aşağıdaki örnekte, `std::vector` sınıfını miras alan ve ona ek işlevsellik ekleyen bir `MyVector` sınıfı oluşturulmuştur:

```
#include <iostream>
#include <vector>

// std::vector sınıfından türetilmiş bir sınıf
class MyVector : public std::vector<int> {
public:
```

```
    // Standart constructor'ları miras almak için using ifadesi
    using std::vector<int>::vector;

    // Yeni bir fonksiyon ekliyoruz
    int sum() const {
        int total = 0;
        for (const int& val : *this) {
            total += val;
        }
        return total;
    }

    // Örnek bir display fonksiyonu
    void display() const {
        std::cout << "MyVector elements: ";
        for (const int& val : *this) {
            std::cout << val << " ";
        }
        std::cout << std::endl;
    }
};

int main() {
    // MyVector sınıfının nesnesini oluştur
    MyVector myVec = {1, 2, 3, 4, 5};

    // MyVector nesnesinin elemanlarını göster
    myVec.display();

    // MyVector nesnesinin elemanlarının toplamını hesapla
    std::cout << "Sum of elements: " << myVec.sum() << std::endl;

    return 0;
}
```

Açıklama:

- `MyVector` sınıfı, `std::vector<int>` sınıfından kalıtım alır. Bu, `MyVector` sınıfının `std::vector`'un tüm işlevselliğine sahip olduğu anlamına gelir.
- `using std::vector<int>::vector;` ifadesi, `std::vector` sınıfının constructor'larını `MyVector` sınıfına miras alır. Böylece, `std::vector`'un tüm constructor'ları `MyVector` tarafından kullanılabilir.
- `sum()` adlı bir fonksiyon eklenerek, vektördeki tüm elemanların toplamını hesaplayan yeni bir işlevsellik eklendi.
- `display()` adlı bir fonksiyon eklenerek, vektördeki tüm elemanları ekrana yazdıran bir işlevsellik eklendi.
- `main()` fonksiyonunda, `MyVector` sınıfından bir nesne oluşturuldu ve elemanları gösterildi, ardından elemanlarının toplamı hesaplandı.

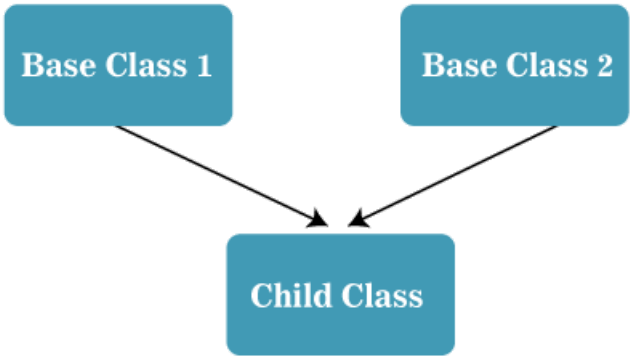
Program Çıktısı:

```
MyVector elements: 1 2 3 4 5
Sum of elements: 15
```

Bu örnek, standart kütüphaneden gelen bir sınıfı miras alarak nasıl genişletebileceğinizi ve özelleştirebileceğinizi göstermektedir.

Multiple Inheritance

Multiple Inheritance, bir sınıfın birden fazla sınıftan miras almasına olanak tanır. Yani, bir alt sınıf birden fazla üst sınıfın özelliklerini ve davranışlarını miras alabilir. Bu özellik, karmaşık hiyerarşiler oluşturmak için kullanılabilir, ancak dikkatli kullanılmazsa karmaşıklık ve isim çakışmaları gibi sorunlara yol açabilir.



Örnek:

Bir `Device` (Cihaz) sınıfı ve bir `Network` (Ağ) sınıfı olduğunu varsayalım. Bir `Smartphone` (Akıllı Telefon) sınıfı hem `Device` hem de `Network` sınıflarından miras alabilir.

Adım 1: Base Sınıfları Tanımlama

Öncelikle `Device` ve `Network` sınıflarını tanımlayalım.

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

class Device {
public:
    std::string brand;
    Device(std::string b) : brand(b) {}
    void showBrand() {
        std::cout << "Device Brand: " << brand << std::endl;
    }
};

class Network {
public:
```

```
    std::string networkType;
    Network(std::string nt) : networkType(nt) {}
    void showNetworkType() {
        std::cout << "Network Type: " << networkType << std::endl;
    }
};
```

Adım 2: Derived Sınıf Tanımlama

Şimdi `Smartphone` sınıfını hem `Device` hem de `Network` sınıflarından miras alacak şekilde tanımlayalım.

```
class Smartphone : public Device, public Network {
public:
    std::string model;
    Smartphone(std::string b, std::string nt, std::string m)
        : Device(b), Network(nt), model(m) {}
    void showDetails() {
        showBrand();
        showNetworkType();
        std::cout << "Smartphone Model: " << model << std::endl;
    }
};
```

Adım 3: Ana Fonksiyonu Yazma

`main` fonksiyonunu yazalım ve `Smartphone` sınıfını kullanarak bir nesne oluşturalım.

```
int main() {
    Smartphone myPhone("Apple", "5G", "iPhone 13");
    myPhone.showDetails();
    return 0;
}
```

Açıklama:

- `Device` ve `Network` sınıfları, sırasıyla cihaz markasını ve ağ türünü saklar.
- `Smartphone` sınıfı, hem `Device` hem de `Network` sınıflarından miras alır. Bu sınıf, hem cihaz markasını hem de ağ türünü ve ayrıca akıllı telefon modelini saklar.
- `Smartphone` sınıfının `showDetails` fonksiyonu, hem cihaz markasını hem de ağ türünü ve modelini ekrana yazdırır.
- `main` fonksiyonunda bir `Smartphone` nesnesi oluşturulur ve `showDetails` fonksiyonu çağrılır.

Repeated Base Classes (The Diamond Problem)

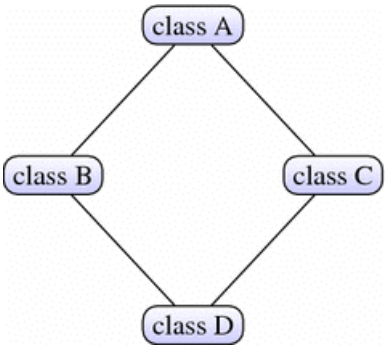
Diamond Problem, çoklu kalıtımın bir sorunudur ve adını, miras diyagramının şekil olarak bir elmas veya baklava dilimine benzemesinden alır.

Diamond Problem Nedir?

Diamond Problem, bir alt sınıfın iki farklı yol üzerinden aynı temel sınıftan miras alması durumunda ortaya çıkar. Bu durumda, hangi temel sınıfın üyesinin kullanılacağı belirsiz hale gelir ve isim çakışmaları gibi sorunlar ortaya çıkabilir.

Örnek:

Aşağıda bir Diamond Problem örneği bulacaksınız. `A` sınıfı, `B` ve `C` sınıflarına temel sınıf olur ve `D` sınıfı hem `B` hem de `C` sınıflarından miras alır. Böylece, `D` sınıfı iki farklı yolla `A` sınıfından miras alır.



Adım 1: Temel ve Ara Sınıfları Tanımlama

Öncelikle `A`, `B`, ve `C` sınıflarını tanımlayalım.

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

class A {
public:
    void show() {
        std::cout << "Class A" << std::endl;
    }
};

class B : public A {
public:
    void show() {
        std::cout << "Class B" << std::endl;
    }
};

class C : public A {
public:
    void show() {
        std::cout << "Class C" << std::endl;
    }
};
```

Adım 2: Diamond Problem Oluşturma

Şimdi `D` sınıfını hem `B` hem de `C` sınıflarından miras alacak şekilde tanımlayalım.

```
class D : public B, public C {
public:
    void show() {
        std::cout << "Class D" << std::endl;
    }
};
```

Adım 3: Ana Fonksiyonu Yazma

`main` fonksiyonunu yazalım ve `D` sınıfını kullanarak bir nesne oluşturalım.

```
int main() {
    D obj;
    obj.show();  // Ambiguous: Which show() should be called?
    return 0;
}
```

Diamond Problem'in Çözümü

Diamond Problem'i çözmek için `virtual` keyword'ünü kullanarak sanal kalıtım uygulanabilir. Bu sayede `A` sınıfından olan miras tek bir kopya olarak alt sınıflarda bulunur.

Adım 4: Sanal Kalıtım ile Diamond Problem'in Çözümü

`A`, `B` ve `C` sınıflarını sanal kalıtım ile güncelleyelim.

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

class A {
public:
    void show() {
        std::cout << "Class A" << std::endl;
    }
};

class B : virtual public A {
public:
    void show() {
        std::cout << "Class B" << std::endl;
    }
};

class C : virtual public A {
public:
    void show() {
        std::cout << "Class C" << std::endl;
```

```
    }
};

class D : public B, public C {
public:
    void show() {
        std::cout << "Class D" << std::endl;
    }
};

int main() {
    D obj;
    obj.A::show();  // Disambiguation: Specify which show() to call
    return 0;
}
```

Açıklama:

- `class B : virtual public A` ve `class C : virtual public A` şeklinde tanımlama yaparak `A` sınıfından sanal kalıtım aldık. Bu, `D` sınıfının tek bir `A` sınıfı kopyasına sahip olmasını sağlar.
- `D` sınıfı içinde `obj.A::show();` şeklinde belirterek hangi `show` fonksiyonunun çağrılacağını belirttik.

Bu şekilde, Diamond Problem sanal kalıtım kullanılarak çözülebilir ve isim çakışmaları engellenir.

Pointers to objects and inheritance

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

// Hayvan sınıfı
class Hayvan {
public:
    virtual void sesCikar() {
        cout << "Hayvanın sesi..." << endl;
    }
};

// Kedi sınıfı, Hayvan sınıfından miras alıyor
class Kedi : public Hayvan {
public:
    void sesCikar() override {
        cout << "Miyav!" << endl;
    }
};

// Köpek sınıfı, Hayvan sınıfından miras alıyor
class Köpek : public Hayvan {
```

```
public:
    void sesCikar() override {
        cout << "Hav hav!" << endl;
    }
};

int main() {
    // Hayvan sınıfı için pointer oluşturuluyor ve Kedi ve Köpek nesnelerin
e işaret edilebilir
    Hayvan *ptr;

    Kedi kedi;
    Köpek köpek;

    // Kedi nesnesine işaret eden pointer
    ptr = &kedi;
    ptr->sesCikar(); // Kedinin "Miyav!" sesi çıkması beklenir

    // Köpek nesnesine işaret eden pointer
    ptr = &köpek;
    ptr->sesCikar(); // Köpeğin "Hav hav!" sesi çıkması beklenir

    return 0;
}
```

Bu örnekte, `Hayvan` adında bir üst sınıf ve `Kedi` ile `Köpek` adında iki alt sınıf tanımladık. `Kedi` ve `Köpek`, `Hayvan` sınıfından miras alıyor. `Hayvan` sınıfında `sesCikar()` adında bir sanal fonksiyon tanımladık ve alt sınıflarda bu fonksiyonu ezdik (override ettik). `main()` fonksiyonunda `Hayvan` sınıfının bir işaretçisi oluşturduk ve önce `Kedi` nesnesine, sonra da `Köpek` nesnesine işaret etmesini sağladık. Bu şekilde, işaretçi nesne türünden bağımsız olarak davranır ve çağrılan fonksiyon, işaret ettiği nesnenin türüne göre belirlenir. Bu, dinamik bağlama (dynamic binding) olarak bilinir.

Accessing members of the Derived class via a pointer to the Base class

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

// Base class (Üst sınıf)
class Base {
public:
    int baseVar;

    void baseFunc() {
        cout << "Base class function" << endl;
    }
};
```

```
// Derived class (Alt sınıf), Base sınıfından miras alıyor
class Derived : public Base {
public:
    int derivedVar;

    void derivedFunc() {
        cout << "Derived class function" << endl;
    }
};

int main() {
    // Base sınıfı işaretçisi oluşturuluyor ve Derived nesnesine işaret edi
lebilir
    Base *basePtr;

    Derived derivedObj;
    basePtr = &derivedObj;

    // Base sınıfındaki üye değişken ve fonksiyonlara erişim
    basePtr->baseVar = 10;
    basePtr->baseFunc();

    // Derived sınıfındaki üye değişken ve fonksiyonlara dolaylı erişim
    // Ancak, basePtr, Derived türünden bir nesneye işaret etmesine rağmen,
    // Derived sınıfının özel üye değişken ve fonksiyonlarına doğrudan eriş
ilemez
    // basePtr->derivedVar = 20;  // Hata! Derived sınıfının üye değişkenle
rine erişim yok
    // basePtr->derivedFunc();     // Hata! Derived sınıfının üye fonksiyon
larına erişim yok

    return 0;
}
```

Bu örnekte, `Base` adında bir üst sınıf ve `Derived` adında bir alt sınıf tanımladık. `Derived`, `Base` sınıfından miras alıyor. `main()` fonksiyonunda, `Base` sınıfının bir işaretçisi (`basePtr`) oluşturduk ve onu bir `Derived` nesnesine işaret etmesini sağladık.

İşaretçi aracılığıyla `baseVar` gibi üye değişkenlere ve `baseFunc()` gibi üye fonksiyonlara doğrudan erişim sağlanabilir. Ancak, `basePtr`, `Base` türünden bir işaretçi olduğu için, `Derived` sınıfının özel üye değişkenlerine (`derivedVar`) ve özel üye fonksiyonlarına (`derivedFunc()`) doğrudan erişim sağlanamaz. Bu erişim, yalnızca `Derived` sınıfının kendi işaretçisiyle mümkündür.

References to objects and inheritance

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

```
// Base class (Üst sınıf)
class Base {
public:
    void baseFunc() {
        cout << "Base class function" << endl;
    }
};

// Derived class (Alt sınıf), Base sınıfından miras alıyor
class Derived : public Base {
public:
    void derivedFunc() {
        cout << "Derived class function" << endl;
    }
};

// Fonksiyon, Base sınıfı referansı alıyor
void callBaseFunc(Base &obj) {
    obj.baseFunc();
}

int main() {
    Base baseObj;
    Derived derivedObj;

    // Base sınıfı nesnesi referansı, Base sınıfının fonksiyonunu çağıran f
onksiyona gönderiliyor
    callBaseFunc(baseObj);

    // Derived sınıfı nesnesi referansı, Base sınıfının fonksiyonunu çağıra
n fonksiyona gönderiliyor
    // Burada Derived sınıfının fonksiyonu çağrılamaz, çünkü referans Base
türünden
    callBaseFunc(derivedObj);

    return 0;
}
```

Bu örnekte, `Base` ve `Derived` sınıfları tanımlanmıştır. `callBaseFunc()` adında bir fonksiyon, bir `Base` sınıfı referansı alır ve bu referans üzerinden `baseFunc()` fonksiyonunu çağırır.

`main()` fonksiyonunda, hem `Base` hem de `Derived` sınıflarından nesneler oluşturulur. `callBaseFunc()` fonksiyonu, hem `Base` hem de `Derived` nesne referanslarını kabul edebilir. Bu, C++'ın dinamik bağlama (dynamic binding) özelliğinden yararlanarak, hangi fonksiyonun çağrılacağını çalışma zamanında belirler. Bu örnekte, `Derived` sınıfının `derivedFunc()` fonksiyonu `callBaseFunc()` fonksiyonuna gönderilse bile, yalnızca `Base` sınıfının `baseFunc()` fonksiyonu çağrılır.

Pointers to objects under private inheritance

Örnek 1:

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

class Base{
public:
   void methodBase();
};

class Derived : private Base {
    // Burada Base sınıfından özel olarak miras alındı
    // Yani Base sınıfının public üyeleri Derived sınıfı içinde private ola
rak erişilebilir hale geldi.
};

int main(){
   Derived dObj;

   // Hata: Derived sınıfı Base sınıfından private olarak miras aldığı için
Base sınıfının public üyelerine dışarıdan erişim mümkün değil.
   dObj.methodBase();

   // Hata: Base sınıfının bir işaretçisine, Derived sınıfının nesnesi olan
dObj'nin adresini atamaya çalıştık.
   // Ancak Derived sınıfı, Base sınıfından özel olarak miras aldığı için,
Base sınıfının public üyelerine erişim mümkün değil.
   Base* bPtr = &dObj;

   return 0;
}
```

Açıklama:

1. `dObj.methodBase();` satırı:
   - `Derived` sınıfı, `Base` sınıfından özel olarak miras aldığı için, `methodBase()` fonksiyonuna `Derived` sınıfı dışından erişim mümkün değildir. Dolayısıyla bu satırda bir hata oluşur.
2. `Base* bPtr = &dObj;` satırı:
   - `Derived` sınıfı, `Base` sınıfından özel olarak miras aldığı için, `Derived` sınıfının nesnesine işaret eden bir işaretçiyi (`&dObj`) `Base` sınıfının işaretçisine (`Base* bPtr`) atayamayız. Çünkü `Base` sınıfının public üyelerine dışarıdan erişim mümkün değildir. Bu nedenle bu satırda da bir hata oluşur.

Örnek 2:

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

// Base class (Üst sınıf)
class Base {
private:
    int baseVar;

public:
    Base(int var) : baseVar(var) {}

    void baseFunc() {
        cout << "Base class function, baseVar: " << baseVar << endl;
    }
};

// Derived class (Alt sınıf), Base sınıfından özel miras alıyor
class Derived : private Base {
private:
    int derivedVar;

public:
    Derived(int baseVar, int var) : Base(baseVar), derivedVar(var) {}

    void derivedFunc() {
        cout << "Derived class function, derivedVar: " << derivedVar << end
l;
    }

    // Base sınıfının fonksiyonlarına erişimi sağlayan bir fonksiyon
    void callBaseFunc() {
        baseFunc();
    }
};

int main() {
    // Derived sınıfından bir nesne oluşturuluyor
    Derived derivedObj(10, 20);

    // Base sınıfının fonksiyonlarına Derived sınıfı aracılığıyla erişiliyo
r
    derivedObj.callBaseFunc();

    return 0;
}
```

Bu örnekte, `Base` ve `Derived` adında iki sınıf tanımladık. `Derived` sınıfı, `Base` sınıfından özel olarak miras alıyor (private inheritance). `Base` sınıfında `baseVar` adında bir özel üye değişken ve `baseFunc()` adında bir üye fonksiyon bulunmaktadır.

`Derived` sınıfında ise `derivedVar` adında bir özel üye değişken, `Derived` sınıfına özgü bir üye fonksiyon olan `derivedFunc()` ve `Base` sınıfının fonksiyonlarına erişim sağlayan `callBaseFunc()` fonksiyonu bulunmaktadır.

`main()` fonksiyonunda, `Derived` sınıfından bir nesne oluşturulur ve bu nesnenin `callBaseFunc()` fonksiyonu aracılığıyla `Base` sınıfının fonksiyonlarına erişim sağlanır. Bu, `Derived` sınıfının `Base` sınıfını özel olarak miras almasına rağmen, `Derived` sınıfının bir üyesi olarak `callBaseFunc()` fonksiyonu tarafından gerçekleştirilir.

OOP08

Polymorphism

Polimorfizm, bir nesne veya yöntemin birden fazla şekli veya işlevi olabilme özelliğidir. Nesne yönelimli programlamada, polimorfizm genellikle miras almayla ve sanal fonksiyonlarla birlikte kullanılır. Polimorfizm, kodun daha esnek, yeniden kullanılabilir ve genellikle daha anlaşılabilir olmasını sağlar.

1. **Compile-Time Polymorphism**:
    - Overloading ve templates gibi tekniklerle gerçekleştirilir.
    - Overloading, aynı isimde farklı parametre listelerine sahip birden fazla fonksiyon tanımlanmasıdır.
    - Templates, genel veri tiplerine veya sınıflara bağlı olarak çalışabilen işlevler veya sınıflar sağlar.
2. **Run-Time Polymorphism**:
    - Inheritance ve virtual (sanal) fonksiyonlar aracılığıyla gerçekleştirilir.
    - Virtual fonksiyonlar, türetilmiş sınıflar tarafından ana sınıfların işlevlerini yeniden tanımlamak için kullanılır.
    - Bu, bir işlevin hangi sınıfın örneği tarafından çağrıldığını çalışma zamanında belirlemeyi sağlar.

Gerçek Hayat Örneği: Şekiller (Shapes) Sistemi

Düşünelim ki bir çizim uygulaması yazıyorsunuz ve kullanıcıya farklı şekiller çizme olanağı sunuyorsunuz. Bu sistemde kareler, daireler, üçgenler gibi farklı şekillerin çizimleri bulunmaktadır. Her bir şekil için çizme işlevi aynıdır ancak şekillerin farklı özellikleri ve davranışları olabilir.

Özellikler:

- Her şeklin alanı farklı şekilde hesaplanabilir.
- Her şeklin rengi olabilir.

Davranışlar:

- Her şekil çizilebilir.

- Her şeklin bilgileri yazdırılabilir.

C++ Programı

```
#include <iostream>
#include <cmath>
using namespace std;

// Base class (Üst sınıf)
class Shape {
protected:
    string color;

public:
    Shape(string clr) : color(clr) {}

    virtual void draw() {
        cout << "Drawing a shape with color: " << color << endl;
    }

    virtual double area() = 0; // Saf sanal fonksiyon (pure virtual functio
n)
};

// Derived class (Alt sınıf), Shape sınıfından türetilmiştir
class Circle : public Shape {
private:
    double radius;

public:
    Circle(string clr, double r) : Shape(clr), radius(r) {}

    void draw() override {
        cout << "Drawing a circle with color: " << color << ", radius: " <<
radius << endl;
    }

    double area() override {
        return M_PI * radius * radius;
    }
};

// Derived class (Alt sınıf), Shape sınıfından türetilmiştir
class Square : public Shape {
private:
    double side;

public:
```

```
    Square(string clr, double s) : Shape(clr), side(s) {}

    void draw() override {
        cout << "Drawing a square with color: " << color << ", side: " << s
ide << endl;
    }

    double area() override {
        return side * side;
    }
};

int main() {
    Shape* shapes[2];
    shapes[0] = new Circle("blue", 5.0);
    shapes[1] = new Square("red", 4.0);

    for (int i = 0; i < 2; ++i) {
        shapes[i]->draw();
        cout << "Area: " << shapes[i]->area() << endl;
    }

    return 0;
}
```

Bu programda, `Shape` adında bir üst sınıf ve `Circle` ile `Square` adında iki alt sınıf tanımlanmıştır. Her iki alt sınıf da `Shape` sınıfından türetilmiştir. Her bir şeklin alanını hesaplamak için pure (saf) virtual (sanal) bir fonksiyon olan `area()` tanımlanmıştır. `main()` fonksiyonunda, `Shape` sınıfının işaretçileri olan `shapes` dizisine `Circle` ve `Square` nesneleri atanmış ve ardından her şeklin `draw()` ve `area()` fonksiyonları çağrılmıştır. Bu, polimorfizmin çalışma zamanında hangi fonksiyonun çağrılacağını belirlemesini gösterir.

Calling Redefined, nonvirtual member functions using pointers

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

// Base class (Üst sınıf)
class Base {
public:
    void display() {
        cout << "Base display()" << endl;
    }
};

// Derived class (Alt sınıf), Base sınıfından türetilmiştir
class Derived : public Base {
```

```
public:
    void display() {
        cout << "Derived display()" << endl;
    }
};

int main() {
    Derived derivedObj;
    Base* basePtr = &derivedObj; // Base sınıfının işaretçisi, Derived sını
fının nesnesine işaret ediyor

    // Derived sınıfının üye fonksiyonu, Derived sınıfının nesnesi üzerinde
n çağrılıyor
    derivedObj.display();

    // Base sınıfının işaretçisi aracılığıyla Base sınıfının üye fonksiyonu
çağrılıyor
    basePtr->display(); // Sonuç: "Base display()"

    return 0;
}
```

Bu örnekte, `Base` adında bir üst sınıf ve `Derived` adında bir alt sınıf tanımlanmıştır. `Derived` sınıfı, `Base` sınıfından türetilmiştir. Hem `Base` hem de `Derived` sınıflarında aynı isimde bir `display()` fonksiyonu tanımlanmıştır, ancak `Derived` sınıfında bu fonksiyon yeniden tanımlanmıştır.

`main()` fonksiyonunda, `Derived` sınıfından bir nesne oluşturulur. Bu nesne üzerinden `display()` fonksiyonu çağrıldığında, `Derived` sınıfının yeniden tanımlanan `display()` fonksiyonu çalışır.

Ardından, `Base` sınıfının işaretçisi `basePtr`, `Derived` sınıfının nesnesine işaret eder. Bu işaretçi aracılığıyla `display()` fonksiyonu çağrıldığında ise, `Base` sınıfının orijinal `display()` fonksiyonu çalışır. Bu, polimorfizmin kullanılmadığı bir durumdur. Ana sınıfın işlevinin çağrılması, işaretçinin türüne bakılmaksızın tanımlanan işlevle uyumlu olan sınıfın işlevini çağırır. Bu polimorfizm değildir. Karar verme durumu compile-time'da gerçekleşir.

Calling Redefined, virtual member functions using pointers (Polymorphism)

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

// Base class (Üst sınıf)
class Base {
public:
    // Sanal fonksiyon
    virtual void display() {
        cout << "Base display()" << endl;
    }
};
```

```
// Derived class (Alt sınıf), Base sınıfından türetilmiştir
class Derived : public Base {
public:
    // Sanal fonksiyonun yeniden tanımlanması
    void display() override {
        cout << "Derived display()" << endl;
    }
};

int main() {
    Derived derivedObj;
    Base* basePtr = &derivedObj; // Base sınıfının işaretçisi, Derived sını
fının nesnesine işaret ediyor

    // Derived sınıfının üye fonksiyonu, Derived sınıfının nesnesi üzerinde
n çağrılıyor
    derivedObj.display(); // Sonuç: "Derived display()"

    // Base sınıfının işaretçisi aracılığıyla, sanal fonksiyon çağrılıyor
    basePtr->display(); // Sonuç: "Derived display()"

    return 0;
}
```

Bu örnekte, `Base` adında bir üst sınıf ve `Derived` adında bir alt sınıf tanımlanmıştır. `Base` sınıfında `display()` adında bir sanal fonksiyon (virtual function) tanımlanmıştır. `Derived` sınıfında ise bu fonksiyon yeniden tanımlanmıştır.

Açıklama:

1. **Sanal Fonksiyon Tanımlama:**
   - `Base` sınıfında `virtual void display()` olarak tanımlanan fonksiyon, sanal fonksiyon olarak belirlenmiştir. Bu, bu fonksiyonun alt sınıflar tarafından yeniden tanımlanabileceği anlamına gelir.
2. **Fonksiyonun Yeniden Tanımlanması:**
   - `Derived` sınıfında `display()` fonksiyonu yeniden tanımlanmış ve `override` anahtar kelimesi kullanılarak bu fonksiyonun sanal bir fonksiyonun yeniden tanımlanması olduğu belirtilmiştir.
3. **Polimorfizmin Uygulanması:**
   - `main()` fonksiyonunda `Derived` sınıfından bir nesne (`derivedObj`) oluşturulmuştur.
   - `Base` sınıfının işaretçisi (`basePtr`), `Derived` sınıfının nesnesine (`derivedObj`) işaret etmektedir.
   - `basePtr` işaretçisi üzerinden `display()` fonksiyonu çağrıldığında, işaretçinin türü `Base` olmasına rağmen `Derived` sınıfının `display()` fonksiyonu çağrılır. Bu, sanal fonksiyonlar ve polimorfizmin bir sonucudur.

Sonuç olarak, işaretçi veya referans `Base` türünde olsa bile, `display()` fonksiyonu `Derived` sınıfı tarafından yeniden tanımlandığı için `Derived` sınıfının fonksiyonu çağrılır. Bu, C++'da run-time

polimorfizminin bir örneğidir.

Using a reference to base class to pass arguments

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

// Base class (Üst sınıf)
class Base {
public:
    virtual void display() {
        cout << "Base display()" << endl;
    }
};

// Derived class (Alt sınıf), Base sınıfından türetilmiştir
class Derived : public Base {
public:
    void display() override {
        cout << "Derived display()" << endl;
    }
};

// Bir Base referansı alan ve display() fonksiyonunu çağıran fonksiyon
void showDisplay(Base& obj) {
    obj.display();
}

int main() {
    Base baseObj;
    Derived derivedObj;

    // Base sınıfı referansı aracılığıyla Base sınıfının nesnesi geçiliyor
    showDisplay(baseObj); // Sonuç: "Base display()"

    // Base sınıfı referansı aracılığıyla Derived sınıfının nesnesi geçiliy
or
    showDisplay(derivedObj); // Sonuç: "Derived display()"

    return 0;
}
```

Açıklama:

1. **Sınıfların Tanımlanması:**
   - `Base` adında bir üst sınıf ve `Derived` adında bu sınıftan türetilmiş bir alt sınıf tanımlanmıştır.

- `Base` sınıfında `virtual void display()` adlı sanal bir fonksiyon vardır.
- `Derived` sınıfı, `Base` sınıfının `display()` fonksiyonunu geçersiz kılar (`override`).

2. **Temel Sınıf Referansını Kullanan Fonksiyon:**
   - `showDisplay(Base& obj)` fonksiyonu, `Base` sınıfından bir referans (`Base&`) alır ve `display()` fonksiyonunu çağırır.
   - Bu fonksiyon, hem `Base` hem de `Derived` sınıfının nesnelerini alabilir.
3. **Fonksiyon Çağrısı:**
   - `main()` fonksiyonunda, `Base` ve `Derived` sınıflarından nesneler oluşturulmuştur.
   - `showDisplay(baseObj)` çağrısı, `Base` sınıfının nesnesini alır ve `Base` sınıfının `display()` fonksiyonunu çağırır.
   - `showDisplay(derivedObj)` çağrısı, `Derived` sınıfının nesnesini alır ve `Derived` sınıfının `display()` fonksiyonunu çağırır. Bu, polimorfizm ve sanal fonksiyonlar sayesinde mümkündür.

Polimorfizmin Çalışma Prensibi:

- `showDisplay` fonksiyonu bir `Base` referansı alır, bu nedenle bu referansın hangi sınıfın örneğine işaret ettiğini anlamak zorundadır. Bu, sanal fonksiyonlar sayesinde yapılır.
- Eğer `display()` fonksiyonu `Base` sınıfında sanal olarak tanımlanmasaydı, `showDisplay` fonksiyonu her zaman `Base` sınıfının `display()` fonksiyonunu çağırırdı.
- Ancak sanal fonksiyonlar ve run-time polimorfizmi sayesinde, `showDisplay` fonksiyonu `Derived` sınıfının `display()` fonksiyonunu çağırabilir.

Benefits of Polymorphism

1. **Kodun Esnekliği ve Yeniden Kullanılabilirliği**:
   - Aynı arayüzü (interface) paylaşan farklı sınıflar oluşturabilirsiniz. Bu sınıfların her biri kendi özel uygulamalarını sağlar, ancak aynı temel işlevselliği sunar. Bu, kodun tekrar kullanılabilirliğini ve esnekliğini artırır.
2. **Bakım ve Genişletilebilirlik**:
   - Yeni türler veya sınıflar eklemek kolaydır. Mevcut kodu değiştirmeye gerek kalmadan yeni sınıflar ekleyebilir ve bunları mevcut arayüzlere entegre edebilirsiniz. Bu, sistemi daha modüler hale getirir ve bakımını kolaylaştırır.
3. **Kodun Basitliği ve Anlaşılabilirliği**:
   - Polimorfizm, karmaşık koşul (if-else veya switch-case) ifadelerinin sayısını azaltır. Aynı temel sınıftan türetilen sınıflar, aynı arayüzü uyguladıkları için, kod daha okunabilir ve anlaşılır hale gelir.
4. **Dinamik Bağlama (Dynamic Binding)**:
   - Polimorfizm sayesinde hangi fonksiyonun çağrılacağı çalışma zamanında belirlenir. Bu, programın daha esnek ve dinamik olmasını sağlar. Aynı işaretçi veya referans, farklı sınıflardan gelen nesnelere işaret edebilir ve uygun fonksiyonu çağırabilir.
5. **Kodun Modülerliği**:

- Polimorfizm, büyük projeleri daha küçük, yönetilebilir parçalara ayırmayı kolaylaştırır. Her sınıf, kendi sorumluluklarına sahip olur ve bu sınıfların etkileşimi belirli arayüzler aracılığıyla sağlanır.

Early (static) binding vs late (dynamic) binding

Early (static) binding

Tanım:
Early (static) binding, fonksiyon çağrılarının derleme zamanında çözüldüğü bağlama türüdür. Compiler, hangi fonksiyonun çağrılacağını belirler ve bu bilgi run-time'da değişmez.

Özellikler:

- Derleme zamanında fonksiyon çağrıları çözülür.
- Performans avantajı sağlar çünkü çağrılacak fonksiyon run-time'da belirlenmez.
- Polimorfizm sağlanamaz.

Örnek:

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

class Base {
public:
    void display() {
        cout << "Base display()" << endl;
    }
};

class Derived : public Base {
public:
    void display() {
        cout << "Derived display()" << endl;
    }
};

int main() {
    Base baseObj;
    Derived derivedObj;

    Base* basePtr = &baseObj;
    Base* derivedPtr = &derivedObj;

    basePtr->display(); // Base display()
    derivedPtr->display(); // Base display() - Çünkü statik bağlama
```

```
    return 0;
}
```

Yukarıdaki örnekte, `basePtr` ve `derivedPtr` işaretçileri kullanılarak `display()` fonksiyonları çağrıldığında, derleyici hangi fonksiyonun çağrılacağını derleme zamanında belirler. İşaretçi türü `Base` olduğundan, `Base` sınıfının `display()` fonksiyonu çağrılır.

Late (dynamic) binding

Tanım:
Late (dynamic) binding, fonksiyon çağrılarının çalışma zamanında çözüldüğü bağlama türüdür. Bu, polimorfizm ve sanal fonksiyonlar ile gerçekleştirilir.

Özellikler:

- Çalışma zamanında fonksiyon çağrıları çözülür.
- Esneklik ve polimorfizm sağlar.
- Performans maliyeti olabilir çünkü çağrılacak fonksiyon run-time'da belirlenir.

Örnek:

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

class Base {
public:
    virtual void display() {
        cout << "Base display()" << endl;
    }
};

class Derived : public Base {
public:
    void display() override {
        cout << "Derived display()" << endl;
    }
};

int main() {
    Base baseObj;
    Derived derivedObj;

    Base* basePtr = &baseObj;
    Base* derivedPtr = &derivedObj;

    basePtr->display(); // Base display()
    derivedPtr->display(); // Derived display() - Çünkü dinamik bağlama
```

```
    return 0;
}
```

Yukarıdaki örnekte, `Base` sınıfında `display()` fonksiyonu `virtual` olarak tanımlanmıştır. Bu, fonksiyon çağrılarının çalışma zamanında çözüleceği anlamına gelir. `derivedPtr` işaretçisi kullanılarak `display()` fonksiyonu çağrıldığında, `Derived` sınıfının `display()` fonksiyonu çağrılır.

Static ve Dynamic Binding Karşılaştırması

1. **Binding Zamanı:**
   - **Static Bağlama:** Compile time.
   - **Dynamic Bağlama:** Run time.
2. **Esneklik:**
   - **Static Bağlama:** Daha az esneklik, polimorfizm desteklenmez.
   - **Dynamic Bağlama:** Daha fazla esneklik, polimorfizm desteklenir.
3. **Performans:**
   - **Static Bağlama:** Daha hızlı çünkü derleme zamanında belirlenir.
   - **Dynamic Bağlama:** Daha yavaş olabilir çünkü run-time'da belirlenir.
4. **Kullanım Durumları:**
   - **Static Bağlama:** Polimorfizmin gerekmediği durumlarda.
   - **Dynamic Bağlama:** Polimorfizmin ve esnekliğin önemli olduğu durumlarda.

How late binding (polymorphism) works

1. Calling Nonvirtual Methods

Geç bağlamanın temel özelliği, sanal (virtual) yöntemlerin çalışma zamanında belirlenmesidir. Sanal olmayan (nonvirtual) yöntemler derleme zamanında belirlenir ve bu da erken bağlamadır (early binding).

Örnek:

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

class Base {
public:
    void nonVirtualMethod() {
        cout << "Base nonVirtualMethod()" << endl;
    }
};

class Derived : public Base {
public:
    void nonVirtualMethod() {
```

```
        cout << "Derived nonVirtualMethod()" << endl;
    }
};

int main() {
    Base baseObj;
    Derived derivedObj;

    Base* basePtr = &derivedObj; // Base pointer pointing to a Derived obje
ct

    // Calling nonvirtual method
    basePtr->nonVirtualMethod(); // Base nonVirtualMethod()

    return 0;
}
```

Yukarıdaki kodda, `Base` sınıfındaki `nonVirtualMethod()` fonksiyonu sanal (virtual) olarak tanımlanmamıştır. Bu nedenle, `Base` sınıfının işaretçisi (`basePtr`) kullanılarak `nonVirtualMethod()` çağrıldığında, derleme zamanında belirlenen `Base` sınıfının fonksiyonu çağrılır.

2. Calling Virtual Methods

Sanal (virtual) yöntemler, çalışma zamanında belirlenir. Bu, polimorfizmin temelini oluşturur ve bu sayede bir alt sınıfın (derived class) yeniden tanımladığı (override) fonksiyon çağrılabilir.

Örnek:

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

class Base {
public:
    virtual void virtualMethod() {
        cout << "Base virtualMethod()" << endl;
    }
};

class Derived : public Base {
public:
    void virtualMethod() override {
        cout << "Derived virtualMethod()" << endl;
    }
};

int main() {
    Base baseObj;
    Derived derivedObj;

    Base* basePtr = &derivedObj; // Base pointer pointing to a Derived obje
```

```
ct

    // Calling virtual method
    basePtr->virtualMethod(); // Derived virtualMethod()

    return 0;
}
```

Yukarıdaki kodda, `Base` sınıfındaki `virtualMethod()` fonksiyonu sanal (virtual) olarak tanımlanmıştır. Bu nedenle, `Base` sınıfının işaretçisi (`basePtr`) kullanılarak `virtualMethod()` çağrıldığında, çalışma zamanında belirlenen `Derived` sınıfının fonksiyonu çağrılır.

3. Calling Virtual Methods and the Virtual Table (vtable)

C++'da, sanal yöntemlerin çalışma zamanında doğru şekilde çağrılmasını sağlamak için sanal tablo (vtable) kullanılır. Her sınıfın kendi sanal tablosu vardır ve bu tablo, sınıfın sanal yöntemlerini ve bunların bellek adreslerini içerir. İşaretçi (pointer) veya referans (reference) bir sanal yöntemi çağırdığında, bu tablo kullanılarak doğru yöntem bulunur ve çağrılır.

Örnek:

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

class Base {
public:
    virtual void virtualMethod() {
        cout << "Base virtualMethod()" << endl;
    }

    virtual void anotherVirtualMethod() {
        cout << "Base anotherVirtualMethod()" << endl;
    }
};

class Derived : public Base {
public:
    void virtualMethod() override {
        cout << "Derived virtualMethod()" << endl;
    }

    void anotherVirtualMethod() override {
        cout << "Derived anotherVirtualMethod()" << endl;
    }
};

int main() {
    Base baseObj;
    Derived derivedObj;
```

```
    Base* basePtr1 = &baseObj;
    Base* basePtr2 = &derivedObj;

    // Calling virtual methods
    basePtr1->virtualMethod();          // Base virtualMethod()
    basePtr1->anotherVirtualMethod();   // Base anotherVirtualMethod()

    basePtr2->virtualMethod();          // Derived virtualMethod()
    basePtr2->anotherVirtualMethod();   // Derived anotherVirtualMethod()

    return 0;
}
```

Bu örnekte, `Base` ve `Derived` sınıflarının her ikisinde de iki sanal yöntem (`virtualMethod` ve `anotherVirtualMethod`) bulunmaktadır. `main()` fonksiyonunda, `basePtr1` ve `basePtr2` işaretçileri kullanılarak sanal yöntemler çağrılır.

- `basePtr1` işaretçisi, `Base` sınıfının nesnesine işaret eder, bu nedenle `Base` sınıfının yöntemleri çağrılır.
- `basePtr2` işaretçisi, `Derived` sınıfının nesnesine işaret eder, bu nedenle `Derived` sınıfının yöntemleri çağrılır.

Sanal Tablo (vtable) ve Sanal İşaretçi (vptr)

Sanal tablo (vtable), bir sınıfın sanal yöntemlerinin bellek adreslerini içeren bir tablodur. Her sınıfın bir sanal tablosu vardır. Sanal işaretçi (vptr), her nesnenin bu tabloya işaret eden bir işaretçisidir.

1. **vtable (sanal tablo)**:
   - Her sınıf için derleyici tarafından oluşturulur.
   - Sanal fonksiyonların adreslerini içerir.
2. **vptr (sanal işaretçi)**:
   - Her nesne için oluşturulur.
   - Nesnenin sınıfına ait sanal tabloya işaret eder.

Örnek Açıklaması:

- `Base` sınıfı için `vtable` şu şekilde olabilir:

```
vtable for Base:
+-----------------------+
| Base::virtualMethod() |
| Base::anotherVirtualMethod() |
+-----------------------+
```

- `Derived` sınıfı için `vtable` şu şekilde olabilir:

```
vtable for Derived:
+-----------------------+
| Derived::virtualMethod() |
```

```
| Derived::anotherVirtualMethod() |
+------------------------+
```

Nesne `derivedObj` oluşturulduğunda, `vptr` işaretçisi `Derived` sınıfının `vtable`ına işaret eder. Bu sayede, `basePtr2->virtualMethod()` çağrıldığında, `Derived::virtualMethod()` yöntemi çağrılır.

Polymorphism does not work with objects

Polimorfizm, C++'da temel olarak işaretçiler ve referanslar ile çalışır. Bu, nesne yönelimli programlamada çalışma zamanı çok biçimliliği (runtime polymorphism) sağlamak için kullanılır. Ancak, doğrudan nesnelerle çalışırken polimorfizmin beklediğiniz gibi çalışmamasının nedeni, C++'da fonksiyonların statik bağlama (early binding) ile çözülmesidir. Bu durum, doğrudan nesnelerle çalışırken fonksiyonların derleme zamanında belirlenmesine yol açar ve bu da polimorfizmi engeller.

Polimorfizm ve İşaretçiler/Referanslar

Polimorfizmin işaretçiler ve referanslarla nasıl çalıştığını anlamak için aşağıdaki örneği göz önünde bulunduralım:

Örnek:

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

class Base {
public:
    virtual void display() {
        cout << "Base display()" << endl;
    }
};

class Derived : public Base {
public:
    void display() override {
        cout << "Derived display()" << endl;
    }
};

void show(Base& obj) {
    obj.display();
}

int main() {
    Base baseObj;
    Derived derivedObj;

    show(baseObj);       // Base display()
    show(derivedObj);    // Derived display()
```

```
    return 0;
}
```

Yukarıdaki kodda, `show` fonksiyonu bir `Base` referansı alır. `show` fonksiyonuna `Base` nesnesi veya `Derived` nesnesi geçildiğinde, doğru `display` fonksiyonu çağrılır. Bu, çalışma zamanı polimorfizmini gösterir ve sanal fonksiyon tablosu (vtable) sayesinde mümkündür.

Polimorfizm Nesnelerle Çalışmaz

Ancak, doğrudan nesneler kullanıldığında, polimorfizm çalışmaz. Bunun nedeni, nesnelerle çalışırken derleme zamanında hangi fonksiyonun çağrılacağının belirlenmesidir. Bu durumu göstermek için aşağıdaki örneği inceleyelim:

Örnek:

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

class Base {
public:
    virtual void display() {
        cout << "Base display()" << endl;
    }
};

class Derived : public Base {
public:
    void display() override {
        cout << "Derived display()" << endl;
    }
};

int main() {
    Base baseObj;
    Derived derivedObj;

    // Doğrudan nesnelerle çağrı
    baseObj.display();    // Base display()
    derivedObj.display(); // Derived display()

    // Nesne kopyalama
    Base anotherBaseObj = derivedObj;
    anotherBaseObj.display(); // Base display()

    return 0;
}
```

Yukarıdaki kodda, `anotherBaseObj` nesnesi `derivedObj` nesnesinin bir kopyası olarak oluşturulmuştur. Ancak, `anotherBaseObj.display()` çağrıldığında `Base` sınıfının `display` fonksiyonu çağrılır. Bunun nedeni,

`anotherBaseObj` nesnesinin türünün `Base` olarak belirlenmiş olması ve polimorfizmin işaretçiler veya referanslarla çalışırken geçerli olmasıdır.

Neden Nesnelerle Polimorfizm Çalışmaz?

1. **Statik Bağlama (Early Binding)**:
   - Doğrudan nesnelerle çalışırken fonksiyon çağrıları derleme zamanında belirlenir.
   - Bu nedenle, `Base` sınıfı türünde bir nesne `Derived` sınıfı türünde bir nesnenin kopyası olarak oluşturulsa bile, `Base` sınıfının fonksiyonları çağrılır.
2. **Nesne Slicing**:
   - Türev sınıf nesnesi bir temel sınıf nesnesine kopyalandığında, türev sınıfın kendine özgü üyeleri kesilir (slicing).
   - Sonuç olarak, kopyalanan nesne temel sınıfın bir örneği haline gelir ve temel sınıfın fonksiyonları çağrılır.

Polimorfizm C++'da işaretçiler ve referanslar ile çalışır çünkü bu şekilde sanal fonksiyon tablosu (vtable) kullanılarak fonksiyon çağrıları çalışma zamanında belirlenebilir. Ancak, doğrudan nesneler kullanıldığında, fonksiyon çağrıları derleme zamanında çözülür ve bu nedenle polimorfizm çalışmaz. İşaretçiler ve referanslar kullanarak, türev sınıf fonksiyonlarının doğru şekilde çağrılmasını sağlamak için sanal fonksiyonların çalışma zamanında bağlanmasını sağlayabilirsiniz.

The rules about virtual functions

1. Sanal Fonksiyonlar Temel Sınıfta Tanımlanmalıdır

Bir fonksiyonun türetilmiş sınıflarda geçersiz kılınmasını (override) sağlamak için, bu fonksiyonun temel sınıfta `virtual` anahtar kelimesi ile tanımlanması gerekir.

Örnek:

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

class Base {
public:
    virtual void display() {
        cout << "Base display()" << endl;
    }
};

class Derived : public Base {
public:
    void display() override {
        cout << "Derived display()" << endl;
    }
};
```

```
int main() {
    Base* basePtr = new Derived();
    basePtr->display(); // Derived display()
    delete basePtr;
    return 0;
}
```

2. Türetilmiş Sınıflarda Sanal Fonksiyonlar Otomatik Olarak Sanaldır

Temel sınıfta bir fonksiyon `virtual` olarak tanımlandığında, türetilmiş sınıflarda bu fonksiyon `virtual` anahtar kelimesi kullanılmasa bile sanal olmaya devam eder.

Örnek:

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

class Base {
public:
    virtual void display() {
        cout << "Base display()" << endl;
    }
};

class Derived : public Base {
public:
    void display() { // 'override' kullanılmamış olsa bile bu sanaldır
        cout << "Derived display()" << endl;
    }
};

int main() {
    Base* basePtr = new Derived();
    basePtr->display(); // Derived display()
    delete basePtr;
    return 0;
}
```

3. Sanal Fonksiyonlar Türetilmiş Sınıflarda `override` Anahtar Kelimesi ile Belirtilmeli

`override` anahtar kelimesi, türetilmiş sınıflarda bir fonksiyonun temel sınıftaki bir sanal fonksiyonu geçersiz kıldığını açıkça belirtir. Bu, hata ayıklamayı ve kodun anlaşılmasını kolaylaştırır.

Örnek:

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

```
class Base {
public:
    virtual void display() {
        cout << "Base display()" << endl;
    }
};

class Derived : public Base {
public:
    void display() override {
        cout << "Derived display()" << endl;
    }
};

int main() {
    Base* basePtr = new Derived();
    basePtr->display(); // Derived display()
    delete basePtr;
    return 0;
}
```

4. Sanal Yıkıcılar (Virtual Destructors)

Bir sınıf sanal fonksiyonlar içeriyorsa, bu sınıfın yıkıcısı (destructor) da sanal olmalıdır. Bu, türetilmiş sınıfların yıkıcılarının doğru şekilde çağrılmasını sağlar.

Örnek:

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

class Base {
public:
    virtual ~Base() {
        cout << "Base Destructor" << endl;
    }
    virtual void display() {
        cout << "Base display()" << endl;
    }
};

class Derived : public Base {
public:
    ~Derived() {
        cout << "Derived Destructor" << endl;
    }
    void display() override {
        cout << "Derived display()" << endl;
    }
```

```
};

int main() {
    Base* basePtr = new Derived();
    basePtr->display(); // Derived display()
    delete basePtr; // Correctly calls ~Derived() and then ~Base()
    return 0;
}
```

5. Pure Virtual Fonksiyonlar ve Soyut Sınıflar

Bir sınıfta bir veya daha fazla saf sanal fonksiyon (pure virtual function) varsa, bu sınıf soyut sınıf (abstract class) olarak adlandırılır ve doğrudan örneklenemez.

Örnek:

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

class Base {
public:
    virtual void display() = 0; // Pure virtual function
};

class Derived : public Base {
public:
    void display() override {
        cout << "Derived display()" << endl;
    }
};

int main() {
    // Base baseObj; // Error: cannot instantiate an abstract class
    Derived derivedObj;
    derivedObj.display(); // Derived display()

    Base* basePtr = &derivedObj;
    basePtr->display(); // Derived display()
    return 0;
}
```

6. Sanal Fonksiyonlar Tabanlı Çok Biçimlilik (Polymorphism)

Sanal fonksiyonlar, aynı temel sınıfı paylaşan farklı sınıfların nesneleri için farklı davranışlar sergilemesini sağlar. Bu, işaretçiler veya referanslar kullanılarak gerçekleştirilir.

Örnek:

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

```
class Shape {
public:
    virtual void draw() {
        cout << "Drawing Shape" << endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing Circle" << endl;
    }
};

class Square : public Shape {
public:
    void draw() override {
        cout << "Drawing Square" << endl;
    }
};

void drawShape(Shape& shape) {
    shape.draw();
}

int main() {
    Circle circle;
    Square square;

    drawShape(circle); // Drawing Circle
    drawShape(square); // Drawing Square

    return 0;
}
```

7. Sanal Fonksiyonların Performans Maliyeti

Sanal fonksiyon çağrıları, doğrudan fonksiyon çağrılarından biraz daha yavaştır çünkü çalışma zamanında sanal tablo (vtable) kullanılarak çözülür. Ancak, modern derleyiciler bu maliyeti minimize etmek için çeşitli optimizasyonlar yapar.

8. Türetilmiş Sınıfta Sanal Fonksiyonun Dönüş Türü (Return Type) Temel Sınıftaki ile Aynı Olmalıdır

C++'da, bir türetilmiş sınıftaki sanal fonksiyonun dönüş türü, temel sınıftaki sanal fonksiyonun dönüş türü ile aynı olmalıdır. Ancak, C++'da "covariant return types" adı verilen bir özellik, türetilmiş sınıfın sanal fonksiyonunun dönüş türünün, temel sınıfın dönüş türünün bir türevi (derived type) olmasına izin verir.

Kural

- **Aynı Dönüş Türü**: Genel kural olarak, bir türetilmiş sınıftaki sanal fonksiyonun dönüş türü, temel sınıftaki sanal fonksiyonun dönüş türü ile tam olarak aynı olmalıdır.
- **Covariant Dönüş Türleri**: İstisna olarak, dönüş türü bir işaretçi veya referans türü ise, türetilmiş sınıfın sanal fonksiyonunun dönüş türü, temel sınıfın dönüş türünün bir alt türü olabilir.

Örnek:

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

class Base {
public:
    virtual Base* getObject() {
        cout << "Base::getObject()" << endl;
        return this;
    }
};

class Derived : public Base {
public:
    Derived* getObject() override { // Covariant return type
        cout << "Derived::getObject()" << endl;
        return this;
    }
};

int main() {
    Base baseObj;
    Derived derivedObj;

    Base* basePtr = &derivedObj;
    basePtr->getObject(); // Derived::getObject()

    return 0;
}
```

Yukarıdaki örnekte, `Base` sınıfında `getObject` fonksiyonu `Base*` türünde bir işaretçi dönerken, `Derived` sınıfında `getObject` fonksiyonu `Derived*` türünde bir işaretçi döner. Bu, covariant dönüş türlerinin bir örneğidir.

Detaylı Açıklama:

- **Aynı Dönüş Türü**:
  Eğer dönüş türü bir temel tür (primitive type) veya kullanıcı tanımlı bir türse, türetilmiş sınıftaki sanal fonksiyonun dönüş türü temel sınıftakiyle aynı olmalıdır. Aksi takdirde, derleyici bir hata verir.

  Örnek:

```
class Base {
public:
    virtual int getValue() {
        return 42;
    }
};

class Derived : public Base {
public:
    // Hatalı: int yerine double döndürüyor
    double getValue() override {
        return 3.14;
    }
};
```

Yukarıdaki örnekte `Derived::getValue()` fonksiyonu `double` döndürüyor, oysa `Base::getValue()` fonksiyonu `int` döndürüyor. Bu derleme hatası verir.

- **Covariant Dönüş Türleri**:
  Eğer dönüş türü bir işaretçi veya referans ise, türetilmiş sınıftaki sanal fonksiyonun dönüş türü, temel sınıftaki dönüş türünün bir alt türü olabilir. Bu, türetilmiş sınıflarda polimorfizmi desteklerken esneklik sağlar.

  Örnek:

```
class Base {
public:
    virtual Base* clone() const {
        return new Base(*this);
    }
};

class Derived : public Base {
public:
    Derived* clone() const override {
        return new Derived(*this);
    }
};
```

Yukarıdaki örnekte, `Base::clone()` fonksiyonu `Base*` türünde bir işaretçi dönerken, `Derived::clone()` fonksiyonu `Derived*` türünde bir işaretçi döner. Bu, covariant dönüş türlerinin kullanımını gösterir ve doğru çalışır.

Özet:

- **Aynı Dönüş Türü**: Temel ve türetilmiş sınıflardaki sanal fonksiyonlar aynı dönüş türüne sahip olmalıdır.
- **Covariant Dönüş Türleri**: Eğer dönüş türü bir işaretçi veya referans türüyse, türetilmiş sınıftaki sanal fonksiyon, temel sınıfın dönüş türünün bir türevini dönebilir.

Bu kural, türetilmiş sınıfların fonksiyonları için tutarlılığı sağlar ve polimorfizmin doğru şekilde çalışmasına yardımcı olur. Covariant dönüş türleri ise esneklik sağlayarak, daha spesifik türlerin döndürülmesine olanak tanır.

`override` specifier

C++11 ile tanıtılan `override` belirteci, türetilmiş sınıflarda sanal fonksiyonların geçersiz kılındığını (override) açıkça belirtmek için kullanılır. Bu belirteç, kodun daha okunabilir ve bakımı kolay olmasını sağlar, aynı zamanda derleyiciye belirli hataları yakalama fırsatı verir.

`override` Belirtecinin Kullanımı ve Faydaları

1. Geçersiz Kılma (Overriding) Doğrulaması

`override` belirteci, bir fonksiyonun temel sınıftaki sanal bir fonksiyonu geçersiz kıldığını belirtir. Eğer temel sınıfta böyle bir fonksiyon yoksa veya fonksiyon imzası uyuşmuyorsa, derleyici bir hata verecektir.

Örnek:

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

class Base {
public:
    virtual void display() const {
        cout << "Base display" << endl;
    }
};

class Derived : public Base {
public:
    void display() const override { // Correctly overrides Base::display
        cout << "Derived display" << endl;
    }
};

int main() {
    Base* b = new Derived();
    b->display(); // Derived display
    delete b;
    return 0;
}
```

Yukarıdaki örnekte, `Derived` sınıfındaki `display` fonksiyonu, `Base` sınıfındaki `display` fonksiyonunu geçersiz kılıyor ve `override` belirteci ile bu açıkça belirtiliyor.

2. Hata Yakalama

`override` belirteci, yazım hatalarını ve yanlış geçersiz kılmaları yakalamada yardımcı olur. Örneğin, temel sınıftaki sanal fonksiyonun imzası değiştirilirse veya türetilmiş sınıfta yanlış yazılırsa, derleyici bir hata üretir.

Örnek:

```
class Base {
public:
    virtual void display() const {
        cout << "Base display" << endl;
    }
};

class Derived : public Base {
public:
    // void display() override { // Error: Does not match Base::display() s
ignature
    void display() const override { // Correctly matches Base::display() si
gnature
        cout << "Derived display" << endl;
    }
};
```

Yukarıdaki örnekte, `Derived` sınıfındaki `display` fonksiyonunun imzası `Base` sınıfındaki `display` fonksiyonu ile uyuşmazsa, derleyici bir hata verir. Bu, yanlışlıkla yapılan hataları önler.

3. `override` ve `final` Belirtecinin Birlikte Kullanımı

`override` belirteci, `final` belirteci ile birlikte kullanılabilir. `final`, bir sanal fonksiyonun daha fazla geçersiz kılınamayacağını belirtir.

Örnek:

```
class Base {
public:
    virtual void display() const {
        cout << "Base display" << endl;
    }
};

class Derived : public Base {
public:
    void display() const override final { // This function cannot be overri
dden further
        cout << "Derived display" << endl;
    }
};

class MoreDerived : public Derived {
public:
```

```
    // void display() const override { // Error: display in Derived is fina
l
    //     cout << "MoreDerived display" << endl;
    // }
};
```

Yukarıdaki örnekte, `Derived` sınıfındaki `display` fonksiyonu `final` olarak belirtilmiştir, bu da `MoreDerived` sınıfının bu fonksiyonu geçersiz kılamayacağı anlamına gelir.

Özet

- **Geçersiz Kılma Doğrulaması**: `override` belirteci, türetilmiş sınıftaki bir fonksiyonun temel sınıftaki bir sanal fonksiyonu geçersiz kıldığını belirtir.
- **Hata Yakalama**: Yanlış yazılmış veya yanlış imza ile tanımlanmış fonksiyonları yakalar.
- **Kod Okunabilirliği**: Kodun okunabilirliğini artırır ve niyetin açıkça belirtilmesini sağlar.
- `final` **ile Birlikte Kullanım**: `final` belirteci ile birlikte kullanılarak, fonksiyonların daha fazla geçersiz kılınması engellenebilir.

`override` belirteci, C++'da sanal fonksiyonların doğru şekilde kullanılmasını sağlar ve kodun bakımını kolaylaştırır. Bu, özellikle büyük kod tabanlarında hata ayıklamayı ve kodun anlaşılmasını önemli ölçüde kolaylaştırır.

`final` specifier

C++11 ile tanıtılan `final` belirteci, sınıflar ve sanal fonksiyonlar için kullanılabilen bir belirteçtir. Bu belirteç, bir sınıfın veya sanal fonksiyonun daha fazla türetilmesini veya geçersiz kılınmasını engeller. `final` belirteci, kodun belirli kısımlarının değiştirilmesini istemediğinizde veya performans optimizasyonları yapmak istediğinizde faydalıdır.

`final` Belirtecinin Kullanımı ve Faydaları

1. Sınıflar İçin `final` Belirtecinin Kullanımı

Bir sınıf `final` olarak belirtilirse, bu sınıftan başka sınıfların türetilmesi engellenir.

Örnek:

```
class Base {
public:
    virtual void display() const {
        cout << "Base display" << endl;
    }
};

class Derived final : public Base { // 'final' sınıf
public:
    void display() const override {
        cout << "Derived display" << endl;
    }
```

```
};

// Error: Cannot derive from 'final' class 'Derived'
// class MoreDerived : public Derived {
// public:
//     void display() const override {
//         cout << "MoreDerived display" << endl;
//     }
// };

int main() {
    Derived d;
    d.display(); // Derived display
    return 0;
}
```

Yukarıdaki örnekte, `Derived` sınıfı `final` olarak belirtilmiştir, bu nedenle `MoreDerived` sınıfı `Derived` sınıfından türetilemez. Derleyici bu durumu hata olarak bildirir.

2. Sanal Fonksiyonlar İçin `final` Belirtecinin Kullanımı

Bir sanal fonksiyon `final` olarak belirtilirse, bu fonksiyonun türetilmiş sınıflarda geçersiz kılınması engellenir.

Örnek:

```
class Base {
public:
    virtual void display() const {
        cout << "Base display" << endl;
    }
};

class Derived : public Base {
public:
    void display() const override final { // 'final' fonksiyon
        cout << "Derived display" << endl;
    }
};

// Error: Cannot override 'final' function 'display' in 'Derived'
// class MoreDerived : public Derived {
// public:
//     void display() const override {
//         cout << "MoreDerived display" << endl;
//     }
// };

int main() {
    Derived d;
```

```
    d.display(); // Derived display
    return 0;
}
```

Yukarıdaki örnekte, `Derived` sınıfındaki `display` fonksiyonu `final` olarak belirtilmiştir, bu nedenle `MoreDerived` sınıfı bu fonksiyonu geçersiz kılamaz. Derleyici bu durumu hata olarak bildirir.

Faydaları

1. **Hata Önleme**: `final` belirteci, istemeden yapılan türetmeleri veya geçersiz kılmaları engelleyerek potansiyel hataları önler.
2. **Kod Anlaşılabilirliği**: `final` belirteci, hangi sınıfların veya fonksiyonların değiştirilemeyeceğini açıkça belirtir, bu da kodun anlaşılabilirliğini artırır.
3. **Performans Optimizasyonları**: Derleyiciler, `final` belirteci kullanılarak belirtilmiş sınıflar ve fonksiyonlar üzerinde daha agresif optimizasyonlar yapabilir. Örneğin, sanal fonksiyon çağrılarının statik bağlama (early binding) ile çözülmesi mümkün olabilir, bu da performansı artırır.

Özet

- **Sınıflar İçin `final`**: Bir sınıfı `final` olarak belirtmek, bu sınıftan türetmeyi engeller.
- **Sanal Fonksiyonlar İçin `final`**: Bir sanal fonksiyonu `final` olarak belirtmek, bu fonksiyonun daha fazla geçersiz kılınmasını engeller.
- **Faydalar**: Hata önleme, kodun anlaşılabilirliğini artırma ve performans optimizasyonlarına olanak sağlar.

`final` belirteci, kodun güvenliğini ve kararlılığını artırmak için önemli bir araçtır. Özellikle büyük projelerde, hangi kısımların değiştirilebileceğini ve hangi kısımların sabit kalması gerektiğini net bir şekilde belirlemek, yazılımın bakımını ve genişletilebilirliğini kolaylaştırır.

Overloading, Name Hiding, Overriding, Polymorphism

1. Overloading

Overloading, aynı isme sahip birden fazla fonksiyonun tanımlanmasını ifade eder. Bu fonksiyonlar farklı parametre listeleri ile çağrılabilir. Derleyici, hangi işlevin çağrılacağını derleme zamanında (compile-time) belirler.

2. Name Hiding (Compile-time Overriding)

Name hiding, bir türetilmiş sınıfın bir üye fonksiyonunun temel sınıfın aynı isimli bir fonksiyonunu gizlemesi (override etmesi) durumunu ifade eder. Ancak, bu durumun polimorfik davranışla ilgisi yoktur çünkü gizlenen fonksiyonlar derleme zamanında çözümlenir.

3. Polymorphism (Run-time Overriding)

Polymorphism, aynı isme sahip farklı işlevlerin çalışma zamanında (run-time) davranışlarının değişebilmesidir. Polimorfizm, sanal fonksiyonlar kullanılarak veya işaretçiler ve referanslar aracılığıyla gerçekleştirilir. Bu, türetilmiş sınıfların temel sınıfın sanal fonksiyonlarını geçersiz kılmasına (override) ve çalışma zamanında doğru fonksiyonun çağrılmasını sağlar.

Abstract Classes (Soyut Sınıflar)

Soyut sınıflar, içinde saf sanal fonksiyonlar (pure virtual functions) bulunan ve dolayısıyla doğrudan örneklenemeyen sınıflardır. Bu sınıflar genellikle arayüzlerin (interfaces) tanımlanmasında kullanılır.

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

// Soyut sınıf
class Shape {
public:
    // Saf sanal fonksiyonlar
    virtual void draw() const = 0;
    virtual double area() const = 0;
};

// Türetilmiş sınıflar
class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    // Saf sanal fonksiyonları uygula
    void draw() const override {
        cout << "Drawing Circle" << endl;
    }

    double area() const override {
        return 3.14 * radius * radius;
    }
};

class Rectangle : public Shape {
private:
    double width, height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}

    // Saf sanal fonksiyonları uygula
    void draw() const override {
        cout << "Drawing Rectangle" << endl;
    }

    double area() const override {
```

```
        return width * height;
    }
};

int main() {
    // Shape sınıfı doğrudan örneklenemez
    // Shape shape; // Error: cannot instantiate abstract class

    Circle circle(5.0);
    Rectangle rectangle(4.0, 6.0);

    // Shape tipinde işaretçiler kullanarak polimorfizm kullanabiliriz
    Shape* shape1 = &circle;
    Shape* shape2 = &rectangle;

    shape1->draw(); // Drawing Circle
    cout << "Area of Circle: " << shape1->area() << endl; // Area of Circl
e: 78.5

    shape2->draw(); // Drawing Rectangle
    cout << "Area of Rectangle: " << shape2->area() << endl; // Area of Rec
tangle: 24

    return 0;
}
```

Yukarıdaki örnekte, `Shape` sınıfı soyut bir sınıftır ve saf sanal fonksiyonlar içerir: `draw()` ve `area()`. `Circle` ve `Rectangle` sınıfları bu saf sanal fonksiyonları uygular ve kendi şekillerine göre bu fonksiyonları doldurur. Ana programda `Shape` sınıfından türetilen işaretçiler kullanılarak bu sınıfların polimorfik davranışı sergilenir. Bu şekilde, soyut sınıflar ve saf sanal fonksiyonlar kullanılarak, farklı sınıfların ortak bir arayüzle birleştirilmesi ve genelleştirilmesi sağlanır.

Desing to an interface, not an implementation

Bu prensip, bir sistemi tasarlarken, kodun belirli bir arayüz veya kontrat üzerine odaklanmasını ve spesifik bir uygulamaya değil, genel bir arayüze yönelik olmasını önerir.

Prensibin Anlamı ve Faydaları:

1. **Esneklik ve Değişkenlik**: Bir arayüze odaklanmak, uygulamanın iç detaylarına bağımlı olmadan, kodun daha esnek ve değişken olmasını sağlar. Bu, sistemdeki değişikliklerin daha kolay uygulanabileceği anlamına gelir.
2. **Bağımsızlık ve Entegrasyon**: Bir arayüze dayalı tasarım, farklı bileşenlerin birbirine bağımlılığını azaltır ve farklı modüllerin daha kolay bir şekilde entegre edilmesini sağlar. Böylece, bileşenler arasındaki ilişkileri ve etkileşimleri sınırlar.
3. **Yeniden Kullanılabilirlik**: Bir arayüz üzerinde tasarlanmış bir sistem, bu arayüzü uygulayan farklı sınıfları kolayca değiştirme ve yeniden kullanma esnekliği sağlar. Bu, kodun daha verimli bir şekilde yazılmasını ve bakımının yapılmasını sağlar.

4. **Test Edilebilirlik**: Arayüze dayalı tasarım, birim testleri yazmayı ve kodun test edilmesini kolaylaştırır. Çünkü, kodun herhangi bir uygulamaya değil, belirli bir arayüze odaklanması, testlerin daha az bağımlılık içermesini sağlar.
5. **Daha İyi Tasarım ve Mimari**: Arayüze dayalı tasarım prensibi, daha iyi bir yazılım tasarımı ve mimarisi oluşturulmasına yardımcı olur. Bu prensip, kodun daha modüler, anlaşılır ve sürdürülebilir olmasını sağlar.

Örnek:

Örneğin, bir dosya sistemi yönetim sistemi tasarlarken, işlevselliğin bir arayüz üzerine odaklanması önemlidir. Bu arayüz, dosya oluşturma, dosya silme, dosya okuma gibi temel işlevleri tanımlar. Farklı işletim sistemleri için farklı dosya sistemleri uygulamaları bu arayüzü uygular ve bu sayede aynı dosya yönetim sistemi kodu, farklı platformlarda çalışabilir.

```
// Dosya sistem arayüzü
class FileSystem {
public:
    virtual void createFile(const string& filename) = 0;
    virtual void deleteFile(const string& filename) = 0;
    virtual void readFile(const string& filename) = 0;
};

// Windows dosya sistem uygulaması
class WindowsFileSystem : public FileSystem {
public:
    void createFile(const string& filename) override {
        // Windows'ta dosya oluşturma işlemi
    }

    void deleteFile(const string& filename) override {
        // Windows'ta dosya silme işlemi
    }

    void readFile(const string& filename) override {
        // Windows'ta dosya okuma işlemi
    }
};

// Linux dosya sistem uygulaması
class LinuxFileSystem : public FileSystem {
public:
    void createFile(const string& filename) override {
        // Linux'ta dosya oluşturma işlemi
    }

    void deleteFile(const string& filename) override {
        // Linux'ta dosya silme işlemi
    }
```

```
    void readFile(const string& filename) override {
        // Linux'ta dosya okuma işlemi
    }
};
```

Yukarıdaki örnekte, `FileSystem` arayüzü dosya sistemi işlevlerini tanımlar ve farklı işletim sistemlerinde bu işlevleri uygulamak için farklı sınıflar oluşturulur. Bu sayede, aynı arayüz üzerinden farklı platformlarda çalışabilen esnek bir dosya sistem yönetimi sağlanır.

The Open-Closed Principle

The Open-Closed Principle (OCP), yazılım tasarımının SOLID prensipleri arasında yer alan önemli bir kavramdır. Bu prensip, Bertrand Meyer tarafından önerilmiştir ve "Software Engineering" adlı kitabında tanıtılmıştır. OCP, bir sınıfın davranışını değiştirmek için sınıfın kodunu değiştirmek yerine, yeni davranışları eklemek veya mevcut davranışları genişletmek için açık, ancak değişikliğe kapalı olması gerektiğini belirtir.

Prensibin Anlamı ve Faydaları:

1. **Açık Olma (Open for Extension)**: Sistem, yeni davranışları eklemek veya mevcut davranışları değiştirmek için açık olmalıdır. Yeni gereksinimler eklendiğinde, kodu değiştirmeden sistemi genişletebilmeliyiz.
2. **Değişikliğe Kapalı Olma (Closed for Modification)**: Varolan kod, yeni davranışlar eklemek veya mevcut davranışları değiştirmek için kapalı olmalıdır. Bu, kodun stabil kalmasını ve mevcut işlevselliğin zarar görmemesini sağlar.
3. **Yeniden Kullanılabilirlik**: OCP, yazılım bileşenlerinin daha modüler ve yeniden kullanılabilir olmasını sağlar. Bir bileşeni değiştirmeden veya kodu değiştirmeden farklı bağlamlarda kullanabilme yeteneği, kodun yeniden kullanılabilirliğini artırır.
4. **Kod Bakımı**: OCP, kodun bakımını kolaylaştırır. Kodun genişletilebilir olması, gelecekteki değişikliklerin daha kolay uygulanabilmesini sağlar.

Örnek:

Örneğin, bir ödeme sistemini ele alalım. Başlangıçta sadece kredi kartı ödemelerini destekleyen bir sistemimiz var. Ancak ileride yeni ödeme yöntemleri (örneğin, Dijital, Banka Transferi) eklememiz gerekebilir. OCP'yi uygulamak için, ödeme işlemini bir arayüzle soyutlayabilir ve farklı ödeme yöntemlerini bu arayüzü uygulayarak ekleyebiliriz. Böylece, ödeme sistemi açık (yeniden kullanılabilir) ve değişikliğe kapalı (mevcut kod değişmeden yeni ödeme yöntemleri ekleniyor).

```
// Ödeme arayüzü
class Payment {
public:
    virtual void pay(double amount) = 0;
};

// Kredi kartı ödeme sınıfı
class CreditCardPayment : public Payment {
public:
```

```
    void pay(double amount) override {
        // Kredi kartı ödeme işlemi
    }
};

// Dijital ödeme sınıfı
class DijitalPayment : public Payment {
public:
    void pay(double amount) override {
        // Dijital ödeme işlemi
    }
};

// Banka transferi ödeme sınıfı
class BankTransferPayment : public Payment {
public:
    void pay(double amount) override {
        // Banka transferi ödeme işlemi
    }
};
```

Yukarıdaki örnekte, `Payment` arayüzü ödeme işlemlerini soyutlar. Her yeni ödeme yöntemi için yeni bir sınıf oluşturabiliriz ve bu sınıfı `Payment` arayüzünü uygulayarak ekleyebiliriz. Bu şekilde, sistemimiz yeni ödeme yöntemlerine açıktır (genişletilebilir), ancak mevcut kod değişmeden kalır (değişikliğe kapalı). Bu da OCP'nin uygulandığını gösterir.

Virtual Constructors

C++ dilinde, bir nesne oluşturulurken, sırayla üst sınıfın ve ardından türetilmiş sınıfların yapıcı fonksiyonları çağrılır. Dolayısıyla, bir nesne oluşturulurken, sanal fonksiyonların varlığından önce yapıcı fonksiyonların tamamlanması gerekir.

Sanal fonksiyonlar, türetilmiş sınıfların üyeleri tarafından geçersiz kılınabilir. Ancak, üst sınıfın yapıcı fonksiyonları çağrıldığında, türetilmiş sınıfın üyeleri henüz oluşturulmamıştır ve dolayısıyla sanal fonksiyonların geçersiz kılınması mümkün değildir.

Bu nedenle, C++ dilinde bir yapıcı fonksiyonun sanal olarak işaretlenmesi veya sanal fonksiyon olarak tanımlanması anlamsızdır ve hatta hatalı olabilir. Dolayısıyla, C++ dilinde yapısal olarak bir sınıfın yapıcı fonksiyonları sanal olamaz. Yapıcı fonksiyonlar, sınıfın kendisini oluşturmakla görevlidir ve bu süreçte sanal fonksiyonların geçersiz kılınması mümkün değildir.

Virtual Destructors

Virtual Destructors, bir sınıf hiyerarşisinde temel sınıfta sanal bir yıkıcı tanımlamak için kullanılır. Bu, bir nesne türünün bilinmeyen bir türdeki bir işaretçi veya referansla silinmesi gerektiğinde, dinamik bağlamayı (dynamic binding) doğru şekilde çözmek için gereklidir.

```
class Base {
public:
    Base() { std::cout << "Base constructor" << std::endl; }
    virtual ~Base() { std::cout << "Base destructor" << std::endl; }
};

class Derived : public Base {
public:
    Derived() { std::cout << "Derived constructor" << std::endl; }
    ~Derived() { std::cout << "Derived destructor" << std::endl; }
};

int main() {
    Base* ptr = new Derived();
    delete ptr; // Bu satırda doğru yıkıcı çağrılması için Base'in yıkıcısı
nın sanal olması gerekir
    return 0;
}
```

Yukarıdaki örnekte, `Base` sınıfının yıkıcısı (`~Base()`) sanal olarak işaretlenmiştir. Bu, `Base` sınıfından türetilmiş olan `Derived` sınıfının nesneleri `Base` sınıfı işaretçisi veya referansı ile silindiğinde, `Derived` sınıfının yıkıcısının çağrılmasını sağlar. Eğer `~Base()` sanal olmasaydı, yıkıcı hiyerarşideki en üst sınıfın yıkıcısı olan `~Base()` çağrılırdı ve `Derived` sınıfının yıkıcısı çağrılmazdı, bu da bellek sızıntısına neden olurdu.

Bu nedenle, C++ dilinde, bir sınıf hiyerarşisinde en az bir sanal fonksiyon bulunuyorsa, genellikle o sınıfın yıkıcısının da sanal olması tavsiye edilir. Bu, doğru yıkıcının çağrılmasını sağlar ve dinamik bellek tahsisinin güvenli bir şekilde yönetilmesine yardımcı olur.

OOP09

Generic Programming: Templates

Generic programming, C++ dilinde kodun veri türlerinden bağımsız olarak yazılmasına olanak sağlar. Bu, özellikle tekrar eden kodları önlemek ve kodun yeniden kullanılabilirliğini artırmak için kullanılır. C++ dilinde generic programming'in ana mekanizması "templates" (şablonlar) olarak bilinir.

Templates, kodun tek bir kez yazılmasını ve birçok farklı veri türüyle çalışabilmesini sağlar. C++ dilinde iki ana tür template vardır: function templates (fonksiyon şablonları) ve class templates (sınıf şablonları).

Function Templates

Fonksiyon şablonları, belirli bir veri türünden bağımsız olarak çalışan fonksiyonlar tanımlamanıza olanak tanır. Şablonlar, aynı fonksiyonun farklı veri türleriyle kullanılabilmesini sağlar.

Örnek:

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

// Function template
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    // int tipinde
    int result1 = add(5, 3);
    cout << "5 + 3 = " << result1 << endl; // 5 + 3 = 8

    // double tipinde
    double result2 = add(2.5, 1.2);
    cout << "2.5 + 1.2 = " << result2 << endl; // 2.5 + 1.2 = 3.7

    // string tipinde
    string result3 = add(string("Hello, "), string("World!"));
    cout << "\\"Hello, \\" + \\"World!\\" = " << result3 << endl; // "Hell
o, " + "World!" = Hello, World!

    return 0;
}
```

Yukarıdaki örnekte, `add` fonksiyon şablonu herhangi bir veri türü (int, double, string) ile çalışabilir. Şablonun tür parametresi `T` olarak tanımlanır ve fonksiyon çağrıldığında, `T` yerine uygun veri türü geçer.

Class Templates

Sınıf şablonları, belirli bir veri türünden bağımsız olarak çalışan sınıflar tanımlamanıza olanak tanır. Bu, özellikle container sınıfları ve veri yapıları için kullanışlıdır.

Örnek:

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

// Class template
template <typename T>
class Box {
private:
    T value;
public:
    Box(T v) : value(v) {}

    void setValue(T v) {
```

```
        value = v;
    }

    T getValue() const {
        return value;
    }
};

int main() {
    // int tipinde Box
    Box<int> intBox(123);
    cout << "intBox value: " << intBox.getValue() << endl; // intBox value:
123

    // double tipinde Box
    Box<double> doubleBox(456.78);
    cout << "doubleBox value: " << doubleBox.getValue() << endl; // doubleB
ox value: 456.78

    // string tipinde Box
    Box<string> stringBox("Template");
    cout << "stringBox value: " << stringBox.getValue() << endl; // stringB
ox value: Template

    return 0;
}
```

Yukarıdaki örnekte, `Box` sınıfı herhangi bir veri türüyle çalışabilir. `Box` sınıfı bir şablon sınıfıdır ve tür parametresi `T` olarak tanımlanır. Bu sınıf, int, double, string gibi farklı türler için ayrı ayrı tanımlanabilir.

> *Programın kullandığı RAM miktarı template kullanılsa da fonksiyonlar ayrı ayrı yazılsa da aynıdır. Compiler belirli bir veri türü için her template örneğini yalnızca bir kez oluşturur.*

Objects as Template Arguments

C++ dilinde şablonlar, sınıfları, fonksiyonları ve hatta nesneleri şablon argümanları olarak kabul edebilir. Bu, belirli bir nesneyi bir şablon argümanı olarak kullanarak şablonun davranışını özelleştirebilmenizi sağlar. Nesneleri şablon argümanları olarak kullanmanın yaygın bir örneği, bir sınıfın nesnesini şablon argümanı olarak geçirerek, bu nesneyi sınıf içinde kullanmak veya belirli özelliklerine dayalı olarak işlem yapmaktır.

Örnek:

Bu örnekte, bir `Policy` sınıfı tanımlayıp, bu sınıfın nesnesini bir şablon argümanı olarak kullanacağız.

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

// Policy sınıfı
class Policy {
public:
    void applyPolicy() const {
        cout << "Applying policy..." << endl;
    }
};

// Template sınıfı
template <typename T, T obj>
class Manager {
public:
    void performAction() const {
        obj.applyPolicy();
    }
};

int main() {
    Policy policy;

    // Policy nesnesini şablon argümanı olarak geçirme
    Manager<Policy, policy> manager; // Hata: policy bir nesne olduğu için
böyle kullanılamaz

    manager.performAction(); // "Applying policy..." çıktısını verir

    return 0;
}
```

Ancak yukarıdaki kod çalışmaz, çünkü C++'da bir nesneyi doğrudan bir şablon argümanı olarak geçiremezsiniz. Bunun yerine, `constexpr` ile sabit bir nesne veya `constexpr` fonksiyonunu şablon argümanı olarak kullanabilirsiniz. Alternatif olarak, işaretçileri veya referansları kullanabilirsiniz. İşte düzeltilmiş bir örnek:

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

// Policy sınıfı
class Policy {
public:
    void applyPolicy() const {
        cout << "Applying policy..." << endl;
    }
};
```

```
// Template sınıfı
template <typename T>
class Manager {
private:
    const T& obj;
public:
    Manager(const T& policyObj) : obj(policyObj) {}

    void performAction() const {
        obj.applyPolicy();
    }
};

int main() {
    Policy policy;

    // Policy nesnesini referans olarak geçirme
    Manager<Policy> manager(policy);

    manager.performAction(); // "Applying policy..." çıktısını verir

    return 0;
}
```

Bu düzeltilmiş örnekte, `Manager` şablon sınıfı, `Policy` nesnesini bir referans olarak alır ve bu nesneyi şablonun içinde kullanır. Bu şekilde, `Policy` nesnesinin işlevselliği `Manager` sınıfında kullanılabilir hale gelir.

Multiple Template Arguments

Şablonlar (templates) birden fazla tür veya değer argümanı alabilir. Bu, kodun daha esnek ve genel hale gelmesini sağlar. Hem fonksiyon hem de sınıf şablonları birden fazla şablon argümanını destekler.

Fonksiyon Şablonları

Fonksiyon şablonları, birden fazla tür argümanı kabul edebilir. Örneğin, iki farklı türdeki argümanı alıp, bu türleri işleyen bir fonksiyon şablonu tanımlayabiliriz.

Örnek:

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

template <typename T1, typename T2>
void print(T1 a, T2 b) {
    cout << "First: " << a << ", Second: " << b << endl;
}
```

```
int main() {
    print(5, 3.2);          // int ve double
    print("Hello", 42);     // const char* ve int
    print(3.14, "world");   // double ve const char*

    return 0;
}
```

Bu örnekte, `print` fonksiyonu iki farklı türde argüman alır (`T1` ve `T2`). Bu, fonksiyonun hem tamsayılar, hem ondalıklı sayılar, hem de karakter dizileri gibi farklı türlerdeki verilerle çalışmasını sağlar.

Sınıf Şablonları

Sınıf şablonları da birden fazla tür veya değer argümanı alabilir. Bu, daha karmaşık veri yapıları ve algoritmalar için çok kullanışlıdır.

Örnek:

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

template <typename T1, typename T2>
class Pair {
private:
    T1 first;
    T2 second;
public:
    Pair(T1 a, T2 b) : first(a), second(b) {}

    void display() const {
        cout << "First: " << first << ", Second: " << second << endl;
    }

    T1 getFirst() const {
        return first;
    }

    T2 getSecond() const {
        return second;
    }
};

int main() {
    Pair<int, double> p1(1, 2.5);
    p1.display();  // First: 1, Second: 2.5

    Pair<string, int> p2("Age", 30);
    p2.display();  // First: Age, Second: 30
```

```
    return 0;
}
```

Bu örnekte, `Pair` sınıfı iki tür argümanı (`T1` ve `T2`) kabul eder. Bu sınıf, iki farklı türde veriyi bir arada tutabilen genel bir veri yapısı sağlar. `display` metodu, bu verileri ekrana yazdırır.

Birden Fazla Tür ve Değer Argümanı ile Karma Örnek

Hem tür hem de değer argümanlarını aynı anda kullanan bir örnek:

Örnek:

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

template <typename T, int size>
class Array {
private:
    T arr[size];
public:
    void set(int index, T value) {
        if (index >= 0 && index < size) {
            arr[index] = value;
        }
    }

    T get(int index) const {
        if (index >= 0 && index < size) {
            return arr[index];
        }
        return T(); // default value
    }

    void display() const {
        for (int i = 0; i < size; ++i) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    Array<int, 5> intArray;
    for (int i = 0; i < 5; ++i) {
        intArray.set(i, i * 10);
    }
    intArray.display();  // 0 10 20 30 40

    Array<double, 3> doubleArray;
```

```
    for (int i = 0; i < 3; ++i) {
        doubleArray.set(i, i * 1.1);
    }
    doubleArray.display();  // 0 1.1 2.2

    return 0;
}
```

Bu örnekte, `Array` sınıfı hem bir tür argümanı (`T`) hem de bir tamsayı değeri (`size`) kabul eder. Bu, belirli bir boyutta ve belirli bir türdeki öğeleri depolayabilen genel bir dizi yapısı sağlar.

Explicit Template Arguments

C++'da şablon (template) argümanları genellikle derleyici tarafından çıkarım yapılır (deduction), ancak bazen bu argümanları açıkça belirtmeniz gerekebilir. Bu duruma "explicit template arguments" denir.

Şablon argümanlarını açıkça belirtmek, şablon parametrelerinin türünü veya değerini doğrudan belirtmek anlamına gelir. Bu, derleyicinin argümanları otomatik olarak çıkaramayacağı veya yanlış çıkarabileceği durumlarda özellikle yararlıdır.

Fonksiyon Şablonları ile Explicit Template Arguments

Bir fonksiyon şablonu kullanırken, tür argümanlarını açıkça belirtebilirsiniz.

Örnek:

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

template <typename T>
T multiply(T a, T b) {
    return a * b;
}

int main() {
    // Otomatik çıkarım
    cout << multiply(2, 3) << endl;           // 6 (int)
    cout << multiply(2.5, 1.5) << endl;       // 3.75 (double)

    // Explicit template arguments
    cout << multiply<int>(2.5, 3.5) << endl;  // 6 (int), 2.5 ve 3.5 double
olmasına rağmen int türüne dönüştürülür
    cout << multiply<double>(2, 3) << endl;   // 6.0 (double), 2 ve 3 int o
lmasına rağmen double türüne dönüştürülür

    return 0;
}
```

Bu örnekte, `multiply` fonksiyonu için tür argümanları otomatik olarak çıkarılabilir. Ancak, `multiply<int>` ve `multiply<double>` ile tür argümanlarını açıkça belirterek, fonksiyon çağrısının türünü zorlayabiliriz.

Sınıf Şablonları ile Explicit Template Arguments

Sınıf şablonlarında da şablon argümanlarını açıkça belirtebilirsiniz.

Örnek:

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

template <typename T1, typename T2>
class Pair {
private:
    T1 first;
    T2 second;
public:
    Pair(T1 a, T2 b) : first(a), second(b) {}

    void display() const {
        cout << "First: " << first << ", Second: " << second << endl;
    }
};

int main() {
    // Otomatik çıkarım yok, şablon argümanlarını açıkça belirtmeliyiz
    Pair<int, double> p1(1, 2.5);
    p1.display();  // First: 1, Second: 2.5

    Pair<string, int> p2("Age", 30);
    p2.display();  // First: Age, Second: 30

    return 0;
}
```

Sınıf şablonlarında, şablon parametrelerini her zaman açıkça belirtmeniz gerekir, çünkü derleyici sınıf örneği oluştururken tür çıkarımı yapamaz.

Karmaşık Durumlarda Explicit Template Arguments

Bazı durumlarda, özellikle fonksiyon şablonlarıyla çalışırken, tür çıkarımı karmaşık hale gelebilir ve explicit template arguments kullanmak gerekebilir.

Örnek:

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

template <typename T1, typename T2>
```

```
void printPair(T1 a, T2 b) {
    cout << "First: " << a << ", Second: " << b << endl;
}

int main() {
    // Otomatik çıkarım
    printPair(1, 2.5);        // First: 1, Second: 2.5 (int, double)

    // Explicit template arguments
    printPair<int, double>(1, 2.5);  // First: 1, Second: 2.5 (explicitly s
pecifying types)

    // Explicit template arguments for different types
    printPair<double, string>(3.14, "Pi");  // First: 3.14, Second: Pi

    return 0;
}
```

Bu örnekte, `printPair` fonksiyonu iki farklı türde argüman alabilir. Tür çıkarımı genellikle doğrudur, ancak explicit template arguments ile türleri belirterek, fonksiyon çağrısının kesinlikle istediğiniz türde olmasını sağlayabilirsiniz.

Özet

Explicit template arguments, şablon argümanlarını açıkça belirtmenin bir yoludur ve aşağıdaki durumlarda kullanışlı olabilir:

- Derleyicinin tür çıkarımı yapamadığı veya yanlış yaptığı durumlar.
- Türleri zorla belirlemek istediğiniz durumlar.
- Daha karmaşık şablon yapılandırmalarında doğruluğu sağlamak.

Hem fonksiyon hem de sınıf şablonlarında, explicit template arguments kullanarak kodunuzu daha esnek ve doğru hale getirebilirsiniz.

Non-Type Template Parameters

C++'da şablonlar sadece tür parametreleri (type parameters) değil, aynı zamanda tür dışı parametreleri (non-type parameters) de kabul edebilir. Tür dışı şablon parametreleri, sabit değerleri şablon argümanları olarak kullanmanıza olanak tanır. Bu parametreler, genellikle tamsayı sabitleri, sabit işaretçiler (pointers) veya referanslar gibi değerlere sahip olabilirler.

Tür dışı şablon parametreleri, şablonun daha esnek ve belirli durumlara uyarlanabilir olmasını sağlar. Bu, özellikle veri yapıları ve algoritmalar için faydalıdır.

Örnek: Sabit Boyutlu Dizi (Array)

Bir dizinin boyutunu tür dışı şablon parametresi olarak kullanarak sabit boyutlu bir dizi şablonu tanımlayabiliriz.

Örnek:

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

template <typename T, int size>
class Array {
private:
    T arr[size];
public:
    void set(int index, T value) {
        if (index >= 0 && index < size) {
            arr[index] = value;
        } else {
            cout << "Index out of bounds!" << endl;
        }
    }

    T get(int index) const {
        if (index >= 0 && index < size) {
            return arr[index];
        }
        cout << "Index out of bounds!" << endl;
        return T(); // default value
    }

    void display() const {
        for (int i = 0; i < size; ++i) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    // 5 elemanlı int dizisi
    Array<int, 5> intArray;
    for (int i = 0; i < 5; ++i) {
        intArray.set(i, i * 10);
    }
    intArray.display();  // 0 10 20 30 40

    // 3 elemanlı double dizisi
    Array<double, 3> doubleArray;
    for (int i = 0; i < 3; ++i) {
        doubleArray.set(i, i * 1.1);
    }
    doubleArray.display();  // 0 1.1 2.2
```

```
    return 0;
}
```

Bu örnekte, `Array` sınıfı iki şablon parametresi alır: bir tür (`T`) ve bir tamsayı sabiti (`size`). Bu, belirli bir türde ve belirli bir boyutta sabit uzunlukta bir dizi oluşturmanıza olanak tanır.

Örnek: Sabit İşaretçi (Constant Pointer)

Bir işaretçiyi tür dışı şablon parametresi olarak kullanmak, işaretçi adreslerinin sabit kalmasını sağlar.

Örnek:

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

template <typename T, T* ptr>
class PointerWrapper {
public:
    void display() const {
        cout << "Pointer value: " << *ptr << endl;
    }
};

int main() {
    int value = 42;
    PointerWrapper<int, &value> pw;
    pw.display();  // Pointer value: 42

    return 0;
}
```

Bu örnekte, `PointerWrapper` sınıfı bir tür (`T`) ve bir işaretçi (`T*`) parametresi alır. Şablonun `ptr` parametresi, işaret edilen değeri `display` fonksiyonu içinde kullanır.

Özet

Tür dışı şablon parametreleri (non-type template parameters), şablonları daha esnek ve özelleştirilebilir hale getiren güçlü bir araçtır. Bu parametreler:

- Tamsayı sabitleri (`int`, `char`, `bool`, vb.)
- Sabit işaretçiler (`const T*`)
- Sabit referanslar (`const T&`)

Tür dışı şablon parametreleri, özellikle sabit boyutlu veri yapıları, sabit işaretçiler veya referanslar gibi durumlarda kullanışlıdır. Bu parametreler, şablonların belirli durumlara uyarlanmasını sağlar ve kodun daha esnek, yeniden kullanılabilir ve etkili olmasına katkıda bulunur.

The `auto` Keyword and Return Type Deduction in Templates

C++11 ile birlikte tanıtılan `auto` anahtar kelimesi ve C++14'te eklenen dönüş türü çıkarımı (return type deduction), şablonlarla çalışırken kodunuzu daha okunabilir ve bakımı daha kolay hale getirir. Bu özellikler, özellikle karmaşık türlerin otomatik olarak belirlenmesinde ve fonksiyonların dönüş türlerinin otomatik olarak çıkarılmasında yardımcı olur.

`auto` Keyword

`auto` anahtar kelimesi, derleyicinin değişkenin türünü otomatik olarak belirlemesine olanak tanır. Bu, özellikle şablonlarla çalışırken yararlıdır çünkü türler genellikle karmaşık ve uzun olabilir.

Örnek: `auto` Keyword

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

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};

    // `auto` kullanarak tür çıkarımı
    for (auto it = vec.begin(); it != vec.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

    // Geleneksel yöntemle tür belirtme
    for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

    return 0;
}
```

Bu örnekte, `auto` anahtar kelimesi `vec.begin()` ve `vec.end()` çağrılarının dönüş türlerini otomatik olarak belirler. Bu, kodun daha kısa ve daha okunabilir olmasını sağlar.

Return Type Deduction in Templates

C++14 ile birlikte, `auto` anahtar kelimesi, fonksiyonların dönüş türlerini otomatik olarak çıkarmak için de kullanılabilir. Bu, özellikle şablonlarla çalışırken fonksiyonların dönüş türlerini belirlemek için faydalıdır.

Örnek: Return Type Deduction in Templates

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

// C++11 öncesi, dönüş türünü açıkça belirtmek gerekiyor
template <typename T1, typename T2>
```

```
auto add(T1 a, T2 b) -> decltype(a + b) {
    return a + b;
}

// C++14 ve sonrası, dönüş türü otomatik olarak çıkarılır
template <typename T1, typename T2>
auto add(T1 a, T2 b) {
    return a + b;
}

int main() {
    cout << add(5, 3) << endl;       // 8
    cout << add(2.5, 1.5) << endl;   // 4.0
    cout << add(3, 4.5) << endl;     // 7.5

    return 0;
}
```

Bu örnekte, `add` fonksiyonu iki farklı türde argüman alır ve bu argümanların toplamını döndürür. C++14 ve sonrasında, dönüş türünü belirtmek zorunda kalmadan `auto` anahtar kelimesi ile dönüş türü otomatik olarak çıkarılabilir.

`auto` in Function Parameters

C++20 ile birlikte, `auto` anahtar kelimesi fonksiyon parametrelerinde de kullanılabilir, bu da tür çıkarımını daha da güçlendirir.

Örnek: `auto` in Function Parameters (C++20)

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

auto add(auto a, auto b) {
    return a + b;
}

int main() {
    cout << add(5, 3) << endl;       // 8
    cout << add(2.5, 1.5) << endl;   // 4.0
    cout << add(3, 4.5) << endl;     // 7.5

    return 0;
}
```

Bu örnekte, `add` fonksiyonu `auto` anahtar kelimesi ile tanımlanmış parametreleri kullanır. Bu, fonksiyonun herhangi bir türde argüman alabilmesini sağlar ve dönüş türünü otomatik olarak çıkarır.

Özet

- `auto` **Keyword**: Değişkenlerin türlerini otomatik olarak belirler, kodu daha kısa ve okunabilir hale getirir.
- **Return Type Deduction**: C++14 ile birlikte, `auto` anahtar kelimesi fonksiyonların dönüş türlerini otomatik olarak çıkarmak için kullanılabilir.
- **Function Parameters**: C++20 ile birlikte, `auto` anahtar kelimesi fonksiyon parametrelerinde de kullanılabilir.

Bu özellikler, C++ şablonlarıyla çalışırken kodunuzu daha esnek ve güçlü hale getirir. Tür çıkarımını derleyiciye bırakarak, daha az kod yazarken daha fazla iş yapabilirsiniz.

Abbreviated Function Templates

C++20 ile tanıtılan "Abbreviated Function Templates" (kısaltılmış fonksiyon şablonları), `auto` anahtar kelimesi kullanarak fonksiyon şablonlarını daha kısa ve daha okunabilir hale getirmemize olanak tanır. Bu özellik, şablon parametrelerini açıkça belirtmeden, tür çıkarımını fonksiyon parametrelerinde yaparak fonksiyon şablonları tanımlamayı sağlar.

Abbreviated function templates, fonksiyon parametrelerinde `auto` anahtar kelimesi kullanarak tür çıkarımını otomatik hale getirir. Bu, şablon parametrelerini belirtmek zorunda kalmadan şablon fonksiyonları tanımlamanın daha kısa ve basit bir yoludur.

Örnek: Abbreviated Function Templates

Aşağıda, iki sayıyı toplayan ve sonucu döndüren bir fonksiyon şablonunun hem geleneksel hem de kısaltılmış hali gösterilmektedir:

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

// Geleneksel fonksiyon şablonu
template <typename T, typename U>
auto add(T a, U b) -> decltype(a + b) {
    return a + b;
}

// Kısaltılmış fonksiyon şablonu (C++20)
auto add(auto a, auto b) {
    return a + b;
}

int main() {
    cout << add(5, 3) << endl;       // 8
    cout << add(2.5, 1.5) << endl;   // 4.0
    cout << add(3, 4.5) << endl;     // 7.5

    return 0;
}
```

Bu örnekte, kısaltılmış fonksiyon şablonu `auto` anahtar kelimesi ile tanımlanmıştır ve tür çıkarımı fonksiyon parametreleri üzerinden yapılır. Bu, şablon parametrelerini açıkça belirtmeden aynı işlevselliği sağlar.

Abbreviated Function Templates ile Tür Çıkarımı

Abbreviated function templates kullanarak, fonksiyon parametrelerinde `auto` kullanımı sayesinde, derleyici parametre türlerini otomatik olarak çıkarır ve uygun şablon örneklerini oluşturur. Bu, daha temiz ve okunabilir kod yazmanızı sağlar.

Örnek: Farklı Türler ile Kullanım

```
#include <iostream>
#include <string>
using namespace std;

// Kısaltılmış fonksiyon şablonu ile iki değeri birleştirme
auto concatenate(auto a, auto b) {
    return a + b;
}

int main() {
    cout << concatenate(5, 3) << endl;            // 8
    cout << concatenate(2.5, 1.5) << endl;        // 4.0
    cout << concatenate(string("Hello "), "World!") << endl;  // Hello Worl
d!

    return 0;
}
```

Bu örnekte, `concatenate` fonksiyonu iki farklı türdeki argümanı alır ve bunları birleştirir. `auto` anahtar kelimesi sayesinde, fonksiyon parametrelerinin türleri otomatik olarak çıkarılır ve uygun şablon örnekleri oluşturulur.

Özellikler ve Faydalar

- **Kısaltılmış Sözdizimi**: Abbreviated function templates, geleneksel şablon tanımlamalarına göre daha kısa ve okunabilir bir sözdizimi sağlar.
- **Otomatik Tür Çıkarımı**: Fonksiyon parametrelerinde `auto` kullanımı, tür çıkarımını derleyiciye bırakır ve kodu daha esnek hale getirir.
- **Daha Az Kod Tekrarı**: Şablon parametrelerini belirtmeden aynı işlevselliği sağlayarak, kod tekrarını azaltır ve kod bakımını kolaylaştırır.

Özet

Abbreviated function templates, C++20 ile gelen ve fonksiyon şablonlarını daha kısa ve okunabilir hale getiren bir özelliktir. `auto` anahtar kelimesi kullanarak fonksiyon parametrelerinde tür çıkarımı yapılır, bu da şablon parametrelerini belirtmeden esnek ve güçlü şablon fonksiyonları tanımlamanızı sağlar. Bu özellik, özellikle karmaşık türlerle çalışırken kodunuzu daha temiz ve yönetilebilir hale getirir.

OOP10

The Standard Library

The Standard Library (Standart Kütüphane), C++ programlama dili ile birlikte gelen ve birçok yaygın programlama görevini kolaylaştıran bir koleksiyon olarak tanımlanabilir. Bu kütüphane, çeşitli veri yapıları, algoritmalar, girdi/çıktı işlemleri ve yardımcı işlevler sunar. Standart Kütüphane, C++ programcılarına tekrar tekrar kullanılabilir bileşenler sağlayarak kod yazma sürecini hızlandırır ve kodun güvenilirliğini artırır.

The Standard Library'nin Bileşenleri

1. **Containers (Kapsayıcılar)**:
   - Veri yapıları sağlar ve verilerin depolanması ve yönetilmesi için kullanılır.
   - Örnekler: `vector`, `list`, `deque`, `set`, `map`, `unordered_map`, `unordered_set`.
2. **Iterators (Yineleyiciler)**:
   - Kapsayıcılar üzerinde gezinmek için kullanılır.
   - Pointer benzeri işlevler görür.
   - Örnekler: `begin()`, `end()`, `rbegin()`, `rend()`.
3. **Algorithms (Algoritmalar)**:
   - Yaygın algoritmalar sağlar.
   - Kapsayıcılarla birlikte çalışır.
   - Örnekler: `sort`, `find`, `copy`, `for_each`, `transform`.
4. **Functional (Fonksiyonel)**:
   - Fonksiyon objeleri ve lambda ifadeleri için araçlar sağlar.
   - Örnekler: `std::function`, `std::bind`, `std::mem_fn`.
5. **Utilities (Yardımcılar)**:
   - Çeşitli yardımcı işlevler ve sınıflar sağlar.
   - Örnekler: `pair`, `tuple`, `optional`, `variant`, `any`.
6. **Strings (Dizgiler)**:
   - Karakter dizileri ve string işleme için sınıflar sağlar.
   - Örnekler: `std::string`, `std::wstring`.
7. **Streams (Akışlar)**:
   - Girdi ve çıktı işlemleri için araçlar sağlar.
   - Örnekler: `std::cin`, `std::cout`, `std::cerr`, `std::ifstream`, `std::ofstream`.
8. **Time (Zaman)**:
   - Zaman ve tarih ile ilgili sınıflar ve işlevler sağlar.

- Örnekler: `std::chrono`, `std::time`, `std::duration`.

9. **Concurrency (Eşzamanlılık)**:
   - Çoklu iş parçacığı ve paralel programlama için araçlar sağlar.
   - Örnekler: `std::thread`, `std::mutex`, `std::async`, `std::future`.

The Standard Library'nin Kullanımına Örnekler

1. Vector ve Algoritmalar

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {1, 4, 2, 8, 5, 7};

    // Sort the vector
    std::sort(numbers.begin(), numbers.end());

    // Print sorted vector
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Bu örnekte, `std::vector` kullanılarak bir dizi sayı depolanır ve `std::sort` algoritması kullanılarak sıralanır.

2. Map ve Iteratorlar

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> ageMap;
    ageMap["Alice"] = 30;
    ageMap["Bob"] = 25;
    ageMap["Charlie"] = 35;

    // Iterate over the map
    for (const auto& pair : ageMap) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }
```

```
    return 0;
}
```

Bu örnekte, `std::map` kullanılarak isim ve yaş değerleri depolanır ve iterator kullanılarak bu değerler yazdırılır.

Özet

The Standard Library, C++ programcılarına birçok yaygın programlama görevini gerçekleştirebilecekleri hazır araçlar ve bileşenler sağlar. Kapsayıcılar, algoritmalar, iteratorlar, fonksiyonel araçlar, dizgiler, akışlar, zaman yönetimi ve eşzamanlılık gibi çeşitli alanlarda geniş bir işlevsellik sunar. Bu kütüphane, C++ programlarının daha verimli, güvenilir ve bakımı kolay olmasını sağlar.

The Standard Template Library (STL)

The Standard Template Library (STL), C++ programlama dilinin önemli bir parçasıdır ve koleksiyon veri yapıları ve algoritmalarını içeren bir kütüphane setidir. STL, şablonlar kullanarak yeniden kullanılabilir ve genel amaçlı bileşenler sağlar, bu da daha hızlı ve güvenilir yazılım geliştirme süreci anlamına gelir.

STL'nin Bileşenleri

STL'nin üç ana bileşeni vardır: Containers (Kapsayıcılar), Algorithms (Algoritmalar) ve Iterators (Yineleyiciler). Bu bileşenler bir arada kullanılarak güçlü ve esnek veri işleme yapıları oluşturulabilir.

1. Containers (Kapsayıcılar)

Kapsayıcılar, verilerin depolanması ve düzenlenmesi için kullanılan veri yapılarıdır. STL, çeşitli ihtiyaçlara yönelik farklı kapsayıcı türleri sunar.

- **Sequence Containers (Sıralı Kapsayıcılar)**: Verileri sıralı olarak depolar.
  - `vector`: Dinamik boyutlu dizi.
  - `deque`: Çift uçlu kuyruk.
  - `list`: Çift yönlü bağlantılı liste.
  - `array`: Sabit boyutlu dizi (C++11 ile birlikte).
- **Associative Containers (Bağlantılı Kapsayıcılar)**: Verileri anahtar-değer çiftleri olarak depolar ve belirli bir sıraya göre düzenler.
  - `set`: Benzersiz öğeleri sıralı olarak depolar.
  - `map`: Anahtar-değer çiftlerini sıralı olarak depolar.
  - `multiset`: Aynı anahtardan birden fazla öğe depolayabilir.
  - `multimap`: Aynı anahtardan birden fazla anahtar-değer çifti depolayabilir.
- **Unordered Containers (Sırasız Kapsayıcılar)**: Verileri hash tablosu kullanarak depolar.
  - `unordered_set`: Benzersiz öğeleri sırasız olarak depolar.
  - `unordered_map`: Anahtar-değer çiftlerini sırasız olarak depolar.

- `unordered_multiset`: Aynı anahtardan birden fazla öğe depolayabilir.
- `unordered_multimap`: Aynı anahtardan birden fazla anahtar-değer çifti depolayabilir.

2. Algorithms (Algoritmalar)

STL, kapsayıcılarla birlikte kullanılabilen çeşitli algoritmalar sağlar. Bu algoritmalar, genellikle veri işleme ve manipulasyonu için kullanılır.

- **Arama ve Sıralama**: `sort`, `find`, `binary_search`
- **Dönüştürme**: `transform`, `copy`
- **Birleştirme**: `merge`, `unique`
- **Yineleme**: `for_each`

3. Iterators (Yineleyiciler)

Yineleyiciler, kapsayıcılar üzerinde gezinmek ve öğelere erişmek için kullanılır. Pointer benzeri bir işlev görürler ve algoritmalar ile kapsayıcılar arasındaki köprüyü oluştururlar.

- **Input Iterators (Girdi Yineleyiciler)**: Verileri sırayla okur.
- **Output Iterators (Çıktı Yineleyiciler)**: Verileri sırayla yazar.
- **Forward Iterators (İleri Yineleyiciler)**: Verileri tek yönde okur/yazar.
- **Bidirectional Iterators (Çift Yönlü Yineleyiciler)**: Verileri iki yönde okur/yazar.
- **Random Access Iterators (Rastgele Erişim Yineleyiciler)**: Verilere rastgele erişim sağlar.

STL Kullanımına Örnekler

1. Vector ve Algoritmalar

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {1, 4, 2, 8, 5, 7};

    // Vectorü sıralama
    std::sort(numbers.begin(), numbers.end());

    // Sıralanmış vectorü yazdırma
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Bu örnekte, `std::vector` kullanılarak bir dizi sayı depolanır ve `std::sort` algoritması kullanılarak sıralanır.

2. Map ve Iteratorlar

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> ageMap;
    ageMap["Alice"] = 30;
    ageMap["Bob"] = 25;
    ageMap["Charlie"] = 35;

    // Map üzerinde yineleme
    for (const auto& pair : ageMap) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}
```

Bu örnekte, `std::map` kullanılarak isim ve yaş değerleri depolanır ve yineleyici kullanılarak bu değerler yazdırılır.

STL'nin Faydaları

- **Yeniden Kullanılabilirlik**: STL bileşenleri, geniş bir yelpazede farklı projelerde yeniden kullanılabilir.
- **Genel Amaçlılık**: Şablonlar kullanarak, farklı türlerle çalışabilen genel amaçlı veri yapıları ve algoritmalar sağlar.
- **Verimlilik**: STL, yüksek performanslı ve optimize edilmiş bileşenler sunar.
- **Standartlaşma**: STL, C++ standardının bir parçasıdır ve tüm standart uyumlu C++ derleyicileri tarafından desteklenir.

Özet

The Standard Template Library (STL), C++ programlama dilinde veri işleme ve algoritmalar için güçlü ve esnek araçlar sunar. Kapsayıcılar, algoritmalar ve yineleyiciler gibi bileşenleri kullanarak, C++ programcıları daha verimli ve okunabilir kodlar yazabilirler. STL'nin sunduğu bu araçlar, yazılım geliştirme sürecini hızlandırır ve kodun güvenilirliğini artırır.

C++ Standard Library (Standart Kütüphane) ve Standard Template Library (STL) terimleri genellikle birbiriyle karıştırılır, ancak aralarında önemli farklar vardır. STL, C++ Standard Library'nin bir parçasıdır ve ona büyük katkıda bulunur. İlişkilerini ve farklarını anlamak için her birini detaylı bir şekilde inceleyelim.

Standard Library

C++ Standard Library, C++ programlama dili ile birlikte gelen geniş bir kütüphane koleksiyonudur. Bu kütüphane, çeşitli programlama görevlerini kolaylaştırmak için bir dizi hazır bileşen sunar. Bu bileşenler şunları içerir:

1. **STL (Standard Template Library)**:
   - Kapsayıcılar (Containers)
   - Algoritmalar (Algorithms)
   - Yineleyiciler (Iterators)
2. **I/O Streams**: Girdi ve çıktı işlemleri için araçlar sağlar.
   - `std::cin`, `std::cout`, `std::cerr`
   - `fstream`, `ifstream`, `ofstream`
3. **Strings**: Karakter dizileri ve string işleme için sınıflar.
   - `std::string`, `std::wstring`
4. **Utility Libraries**: Yardımcı işlevler ve sınıflar.
   - `std::pair`, `std::tuple`, `std::optional`
5. **Concurrency**: Çoklu iş parçacığı ve paralel programlama için araçlar.
   - `std::thread`, `std::mutex`, `std::future`
6. **Math Libraries**: Matematiksel işlemler ve fonksiyonlar.
   - `cmath`, `complex`, `random`
7. **Time Libraries**: Zaman ve tarih ile ilgili sınıflar ve işlevler.
   - `std::chrono`, `std::time`
8. **Localization**: Yerelleştirme ve uluslararasılaştırma araçları.
   - `std::locale`, `std::codecvt`

Standard Template Library (STL)

STL, C++ Standard Library'nin bir alt kümesi olup, özellikle şablonlarla yazılmış genel amaçlı veri yapıları ve algoritmalar içerir. STL, üç ana bileşenden oluşur:

1. **Containers (Kapsayıcılar)**: Verilerin depolanması ve yönetilmesi için kullanılır.
   - Örnekler: `vector`, `list`, `deque`, `set`, `map`
2. **Algorithms (Algoritmalar)**: Kapsayıcılarla birlikte çalışan yaygın algoritmalar sağlar.
   - Örnekler: `sort`, `find`, `copy`, `for_each`
3. **Iterators (Yineleyiciler)**: Kapsayıcılar üzerinde gezinmek ve öğelere erişmek için kullanılır.
   - Örnekler: `begin`, `end`, `rbegin`, `rend`

STL ve Standard Library Arasındaki İlişki

- **STL, Standard Library'nin Bir Parçasıdır**: STL, C++ Standard Library'nin bir alt kümesidir ve özellikle veri yapıları ve algoritmalar konusunda uzmanlaşmıştır.

- **Standartların Genişletilmesi**: C++ Standard Library, STL bileşenlerine ek olarak I/O işlemleri, zaman yönetimi, çoklu iş parçacığı yönetimi, string işlemleri ve daha pek çok şeyi kapsar.
- **Tarihsel Bağlam**: STL, orijinal olarak Alexander Stepanov ve Meng Lee tarafından geliştirildi ve daha sonra C++ Standart Kütüphane'nin bir parçası olarak benimsendi. Bu, STL'nin daha geniş bir standart kütüphaneye entegrasyonunu sağladı.
- **Kapsam**: STL, temel veri yapıları ve algoritmalara odaklanırken, Standard Library daha geniş bir yelpazede yardımcı işlevler ve araçlar sağlar.

Özet

- **STL (Standard Template Library)**, C++ Standard Library'nin bir parçasıdır ve özellikle şablonlarla yazılmış genel amaçlı veri yapıları ve algoritmalar içerir.
- **C++ Standard Library**, STL'yi içerir, ancak aynı zamanda I/O işlemleri, zaman yönetimi, çoklu iş parçacığı yönetimi, string işlemleri gibi pek çok başka bileşeni de kapsar.
- STL, C++ Standard Library'nin verimli ve yeniden kullanılabilir bileşenlerinin temelini oluşturur, ancak Standard Library STL'den çok daha geniş bir yelpazede işlevsellik sunar.

Bu ilişki, C++ programcılarının çeşitli programlama görevlerini verimli bir şekilde gerçekleştirmelerini sağlayan güçlü ve kapsamlı bir araç seti sunar.

Smart Pointers

Smart pointerlar, C++'ta bellek yönetimini kolaylaştıran ve bellek sızıntılarını önleyen akıllı bir bellek yönetimi aracıdır. C++11'den itibaren standart kütüphanede bulunurlar ve `unique_ptr`, `shared_ptr` ve `weak_ptr` olmak üzere üç ana türü vardır. Her biri farklı bellek yönetim senaryoları için tasarlanmıştır.

1. `unique_ptr<T>`

- `unique_ptr`, yalnızca bir nesneye (belirli bir bellek alanına) sahip olabilecek tek sahipli bir akıllı işaretçidir.
- Sahip olduğu nesne ömrü bittiğinde otomatik olarak yok edilir.
- Diğer akıllı işaretçilere (`shared_ptr`, `weak_ptr`) dönüştürülemez.

Örnek:

```
#include <iostream>
#include <memory>

int main() {
    // unique_ptr kullanımı
    std::unique_ptr<int> ptr(new int(42));

    std::cout << *ptr << std::endl; // Output: 42

    return 0; // ptr, otomatik olarak yok edilecek
}
```

2. `shared_ptr<T>`

- `shared_ptr`, birden fazla akıllı işaretçi tarafından paylaşılabilen bir nesneye sahip olabilen çok sahipli bir akıllı işaretçidir.
- Nesnenin sahipliği son `shared_ptr` nesnesi yok olduğunda sona erer ve bellek otomatik olarak serbest bırakılır.
- `make_shared` işlevi, bellek tahsisi ve nesne oluşturma işlemlerini birleştirir ve hafızada daha verimli bir yapı sağlar.

Örnek:

```
#include <iostream>
#include <memory>

int main() {
    // shared_ptr kullanımı
    std::shared_ptr<int> ptr1 = std::make_shared<int>(42);
    std::shared_ptr<int> ptr2 = ptr1;

    std::cout << *ptr1 << std::endl; // Output: 42
    std::cout << *ptr2 << std::endl; // Output: 42

    return 0; // ptr1 ve ptr2, otomatik olarak yok edilecek
}
```

3. `weak_ptr<T>`

- `weak_ptr`, `shared_ptr`'den türetilen ve döngüsel referanslara yol açabilecek döngüsel bağlantıları önlemek için kullanılan bir zayıf akıllı işaretçidir.
- `weak_ptr`, nesneyi paylaşmaz, sadece paylaşılan bir nesneye erişim sağlar. Dolayısıyla, nesnenin ömrü `weak_ptr`'nin sahipliğiyle sınırlı değildir.
- `lock()` yöntemi, `weak_ptr`'yi bir `shared_ptr`'ye dönüştürür ve nesneye güvenli bir şekilde erişim sağlar.

Örnek:

```
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> sharedPtr = std::make_shared<int>(42);
    std::weak_ptr<int> weakPtr(sharedPtr);

    // weak_ptr'den shared_ptr'ye erişim
    if (auto shared = weakPtr.lock()) {
        std::cout << *shared << std::endl; // Output: 42
    } else {
        std::cout << "weak_ptr, güçsüz" << std::endl;
```

```
    }

    return 0;
}
```

Bu örnekler, `unique_ptr`, `shared_ptr` ve `weak_ptr`'nin kullanımını göstermektedir. Her biri farklı bellek yönetim senaryolarına uygundur ve C++ programcılarına bellek yönetimini daha güvenli ve etkili hale getirme imkanı sağlar.