CPP Module 04

```cpp
#include <iostream>

class Animal {
public:
  virtual void makeSound() {
    std::cout << "Hayvan ses çıkarıyor." << std::endl;
  }
};

class Dog : public Animal {
public:
  void makeSound() {
    std::cout << "Köpek havladı." << std::endl;
  }
};

int main() {
  Animal *a = new Animal();
  Animal *b = new Dog();

  a->makeSound();
  b->makeSound();

  return 0;
}
```

Bu örnekte, temel sınıf `Animal` ve `Animal` sınıfından miras alan alt sınıf `Dog` bulunmaktadır. `Animal` sınıfında, `makeSound` adında bir sanal fonksiyon bulunmaktadır ve `Dog` sınıfı bu fonksiyonun üzerine yazılmış bir uygulaması vardır.

`main` fonksiyonunda, `Animal` tipindeki iki nesne için göstericiler oluşturuyoruz, `a` ve `b`. İlki `a`, `Animal` tipinde bir nesnenin bir örneğine işaret ederken, ikincisi `b`, `Dog` tipinde bir nesnenin bir örneğine işaret eder.

`a` ve `b` üzerinde `makeSound` metodunu çağırdığımızda, aşağıdaki çıktı alıyoruz:

`animals[i] = new Dog();` ifadesi, `animals` adında bir dizi içinde bulunan `i` indeksli elemanın `Dog` tipinde bir nesne örneğine işaret etmesini sağlar. Bu, dinamik bellek yönetimi kullanılarak bir `Dog` nesnesi oluşturur ve oluşturulan nesnenin adresini `animals[i]` göstericisine atar.

Bu şekilde, `animals[i]` göstericisi artık `Dog` tipindeki bir nesnenin bir örneğine işaret eder ve `animals[i]` üzerinde `Dog` tipindeki nesnelere özgü metodlar çağrılabilir.

scss Copy code

```scss
Animal *animals[100];

// ...

animals[i] = new Dog();

// ...

animals[i]->makeSound();
```

Örnekte, `Animal` tipinde 100 elemanlı bir dizi oluşturduk ve dizinin `i` indeksli elemanını `Dog` tipinde bir nesne örneğine işaret etmesini sağladık. Daha sonra, `animals[i]` üzerinde `makeSound` metodunu çağırdık ve `Dog` tipindeki nesnenin üzerine yazılmış metodu çalıştı.

abstract sınıflardan doğrudan nesne oluşturulamaz. Çünkü abstract sınıflarda tanımlanmamış fonksiyonlar bulunur ve bu fonksiyonlar türeyen sınıflarda gerçekleştirilmelidir. Dolayısıyla, abstract sınıflardan türeyen sınıflardan nesne oluşturabilirsiniz. Örnek olarak, Shape sınıfından direk bir nesne oluşturamazsınız ancak Circle sınıfından bir nesne oluşturabilirsiniz.

```
Main.cpp: In function ‘int main()’:
Main.cpp:41:9: error: deleting object of polymorphic class type ‘Animal’ which has non-virtual destructor might cause undefined behavior [-Werror=delete-non-virtual-dtor]
   41 |         delete meta;
      |         ^~~~~~~~~~~
Main.cpp:43:9: error: deleting object of polymorphic class type ‘Animal’ which has non-virtual destructor might cause undefined behavior [-Werror=delete-non-virtual-dtor]
   43 |         delete j;
      |         ^~~~~~~~
Main.cpp:45:9: error: deleting object of polymorphic class type ‘Animal’ which has non-virtual destructor might cause undefined behavior [-Werror=delete-non-virtual-dtor]
   45 |         delete i;
      |         ^~~~~~~~
```

destructor virtual olmazsa derlemiyor (linux)!!! yani new ile oluşturulan delete ile silinemiyor! leakslere sebebiyet verir! alttaki resimde olduğu gibi destructor virtual olmalı!

```cpp
class Animal{
    protected:
        std::string type;
    public:
        Animal();
        Animal(const Animal& acopy);
        Animal& operator = (const Animal& acopy);
        virtual ~Animal(void);
        void setType(std::string sType);
        std::string getType(void) const;
        virtual void makeSound() const;
};
```

Abstract Classes

To write polymorphic functions wee need to have derived classes. But sometimes we don't need to create any base class objects, but only derived class objects. The base class exists only as a starting point for deriving other classes. This kind of base classes we can call an ***abstract class***, which means that no actual objects will be created from it. Abstract classes arise in many situations. A factory can make a sports car or a truck or an ambulance, but it can't make a generic vehicle. The factory must know the details about what *kind* of vehicle to make before it can actually make one. Similarly, you'll see sparrows, wrens, and robins flying around, but you won't see any generic birds. Actually, a class is an abstract class only in the eyes of humans. The compiler is ignorant of our decision to make it an abstract class.

Pure Virtual Functions

It would be nice if, having decided to create an abstract base class, I could instruct the compiler to actively *prevent* any class user from ever making an object of that class. This would give me more freedom in designing the base class because I wouldn't need to plan for actual objects of the class, but only for data and functions that would be used by derived classes. There is a way to tell the compiler that a class is abstract: You define at least one ***pure virtual function*** in the class.

A pure virtual function is a virtual function with no body. The body of the virtual function in the base class is removed, and the notation =0 is added to the function declaration.