# Object Oriented Programming

## CS143: lecture 16

Konstantinos Ameranis, July 30

# Object-Oriented Programming

## Recap: Polymorphism

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise()) # meow
print(animal2.reduce_noise()) # woof
```

# Object-Oriented Programming

## Recap: Polymorphism

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise()) # meow
print(animal2.reduce_noise()) # woof
```

- When `Animal` calls `self.noise()`, it seems like it will call its own `noise`, i.e. `"Generic sine wave"`.

# Object-Oriented Programming

## Recap: Polymorphism

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise()) # meow
print(animal2.reduce_noise()) # woof
```

- When `Animal` calls `self.noise()`, it seems like it will call its own `noise`, i.e. `"Generic sine wave"`.
- Not only can a subclass call a superclass's method, a superclass can also call a subclass's method

# Object-Oriented Programming

## Recap: Polymorphism

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise()) # meow
print(animal2.reduce_noise()) # woof
```

- When `Animal` calls `self.noise()`, it seems like it will call its own `noise`, i.e. `"Generic sine wave"`.
- Not only can a subclass call a superclass's method, a superclass can also call a subclass's method
- But, somehow the `Animal` class knows what it really is precisely

# Object-Oriented Programming

## Recap: Polymorphism

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise())  # meow
print(animal2.reduce_noise())  # woof
```

- When `Animal` calls `self.noise()`, it seems like it will call its own `noise`, i.e. `"Generic sine wave"`.
- Not only can a subclass call a superclass's method, a superclass can also call a subclass's method
- But, somehow the `Animal` class knows what it really is precisely

# Object-Oriented Programming

## Recap: Polymorphism

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise())  # meow
print(animal2.reduce_noise())  # woof
```

- When `Animal` calls `self.noise()`, it seems like it will call its own `noise`, i.e. `"Generic sine wave"`.
- Not only can a subclass call a superclass's method, a superclass can also call a subclass's method
- But, somehow the `Animal` class knows what it really is precisely
- There is only one definition of `reduce_noise`, how does it behave differently?

# Object-Oriented Programming

## Under the hood

# Object-Oriented Programming

## Under the hood

- How do we implement this?

# Object-Oriented Programming

## Under the hood

- How do we implement this?
- Can we implement this in C, which doesn't have any support for OOP?

# Object-Oriented Programming

## Under the hood

- How do we implement this?
- Can we implement this in C, which doesn't have any support for OOP?
  - Yes

# Object-Oriented Programming

## Under the hood

- How do we implement this?
- Can we implement this in C, which doesn't have any support for OOP?
  - Yes
- When people say "X is an OOP language," what they mean is that X has good support for OOP paradigm.

# Object-Oriented Programming

## Under the hood

- How do we implement this?
- Can we implement this in C, which doesn't have any support for OOP?
  - Yes
- When people say "X is an OOP language," what they mean is that X has good support for OOP paradigm.
- We can still capture OOP concepts even in a language that has no support (i.e. C).

# But... we know that function calls are just jumps

```c
int accum = 0;

int sum(int x, int y)
{
        int t = x + y;
        accum += t;
        return t;
}
```

```c
int sum(int x, int y);

int main(void)
{
        return sum(1, 3);
}
```

```
0000000000401110 <sum>:
  401110:89 f8                    mov    %edi,%eax
  401112:01 f0                    add    %esi,%eax
  401114:01 05 12 2f 00 00        add    %eax,0x2f12(%rip)        # 40402c <accum>
  40111a:c3                       retq
  40111b:0f 1f 44 00 00           nopl   0x0(%rax,%rax,1)

0000000000401120 <main>:
  401120:bf 01 00 00 00           mov    $0x1,%edi
  401125:be 03 00 00 00           mov    $0x3,%esi
  40112a:e9 e1 ff ff ff           jmpq   401110 <sum>
  40112f:90                       nop
```

# But... we know that function calls are just jumps

```c
int accum = 0;

int sum(int x, int y)
{
        int t = x + y;
        accum += t;
        return t;
}
```

```c
int sum(int x, int y);

int main(void)
{
        return sum(1, 3);
}
```

```
0000000000401110 <sum>:
  401110:89 f8                    mov    %edi,%eax
  401112:01 f0                    add    %esi,%eax
  401114:01 05 12 2f 00 00        add    %eax,0x2f12(%rip)        # 40402c <accum>
  40111a:c3                       retq
  40111b:0f 1f 44 00 00           nopl   0x0(%rax,%rax,1)

0000000000401120 <main>:
  401120:bf 01 00 00 00           mov    $0x1,%edi
  401125:be 03 00 00 00           mov    $0x3,%esi
  40112a:e9 e1 ff ff ff           jmpq   401110 <sum>
  40112f:90                       nop
```

Put 1 and 3 into registers

# But... we know that function calls are just jumps

```c
int accum = 0;

int sum(int x, int y)
{
        int t = x + y;
        accum += t;
        return t;
}
```

```c
int sum(int x, int y);

int main(void)
{
        return sum(1, 3);
}
```

```
0000000000401110 <sum>:
  401110:89 f8                    mov    %edi,%eax
  401112:01 f0                    add    %esi,%eax
  401114:01 05 12 2f 00 00        add    %eax,0x2f12(%rip)        # 40402c <accum>
  40111a:c3                       retq
  40111b:0f 1f 44 00 00           nopl   0x0(%rax,%rax,1)

0000000000401120 <main>:
  401120:bf 01 00 00 00           mov    $0x1,%edi
  401125:be 03 00 00 00           mov    $0x3,%esi
  40112a:e9 e1 ff ff ff           jmpq   401110 <sum>
  40112f:90                       nop
```

Put 1 and 3 into registers

Jump to 401110

# Object-Oriented Programming

## Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"


class Cat(Animal):
    def noise(self):
        return "Meow"
```

```
animal1 = Cat("bob")
animal2 = Animal("alice")
print(animal1.noise()) # Meow
print(animal2.noise()) # Generic sine wave
```

# Object-Oriented Programming

## Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"


class Cat(Animal):
    def noise(self):
        return "Meow"
```

```
animal1 = Cat("bob")
animal2 = Animal("alice")
print(animal1.noise()) # Meow
print(animal2.noise()) # Generic sine wave
```

- Now that we have two implementations of `noise`, which one do we jump to?

# Object-Oriented Programming

## Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"


class Cat(Animal):
    def noise(self):
        return "Meow"
```

```
animal1 = Cat("bob")
animal2 = Animal("alice")
print(animal1.noise()) # Meow
print(animal2.noise()) # Generic sine wave
```

- Now that we have two implementations of `noise`, which one do we jump to?
- We can look at the types!

# Object-Oriented Programming

## Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"


class Cat(Animal):
    def noise(self):
        return "Meow"
```

```
animal1 = Cat("bob")
animal2 = Animal("alice")
print(animal1.noise()) # Meow
print(animal2.noise()) # Generic sine wave
```

- Now that we have two implementations of `noise`, which one do we jump to?
- We can look at the types!
- *BTW C doesn't have this problem because you can't have more than one functions with the same name*

# Object-Oriented Programming

## Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"


class Cat(Animal):
    def noise(self):
        return "Meow"
```

```
animal1 = Cat("bob")
animal2 = Animal("alice")
print(animal1.noise()) # Meow
print(animal2.noise()) # Generic sine wave
```

- Now that we have two implementations of `noise`, which one do we jump to?
- We can look at the types!
- *BTW C doesn't have this problem because you can't have more than one functions with the same name*
- *But C++, which does have classes, do need to solve this problem*

# Object-Oriented Programming

## Implementation

# Object-Oriented Programming

## Implementation

```
#include <cstdio>
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
};
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
};

class Cat : public Animal {
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
};

class Cat : public Animal {
public:
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
};

class Cat : public Animal {
public:
    void noise();
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
};

class Cat : public Animal {
public:
    void noise();
};
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
};

class Cat : public Animal {
public:
    void noise();
};

void Animal::noise() {
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
};

class Cat : public Animal {
public:
    void noise();
};

void Animal::noise() {
    printf("%s\n", "Generic sine wave");
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
};

class Cat : public Animal {
public:
    void noise();
};

void Animal::noise() {
    printf("%s\n", "Generic sine wave");
}
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
};

class Cat : public Animal {
public:
    void noise();
};

void Animal::noise() {
    printf("%s\n", "Generic sine wave");
}

void Cat::noise() {
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
};

class Cat : public Animal {
public:
    void noise();
};

void Animal::noise() {
    printf("%s\n", "Generic sine wave");
}

void Cat::noise() {
    printf("%s\n", "Meow");
```

# Object-Oriented Programming

## Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
};

class Cat : public Animal {
public:
    void noise();
};

void Animal::noise() {
    printf("%s\n", "Generic sine wave");
}

void Cat::noise() {
    printf("%s\n", "Meow");
}
```

# Object-Oriented Programming

## Implementation

```cpp
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
};

class Cat : public Animal {
public:
    void noise();
};

void Animal::noise() {
    printf("%s\n", "Generic sine wave");
}

void Cat::noise() {
    printf("%s\n", "Meow");
}
```

```cpp
int main() {
    Animal animal1;
    Cat animal2;

    animal1.noise();
    animal2.noise();

    return 0;
}
```

# Object-Oriented Programming

## Implementation

```
int main() {
    Animal animal1;
    Cat animal2;

    animal1.noise();
    animal2.noise();

    return 0;
}
```

```
0000000000401190 <main>:

 ...
  4011b5:e8 76 ff ff ff          callq  401130 <_ZN6Animal5noiseEv>
  4011ba:48 8d 7d f0             lea    -0x10(%rbp),%rdi
  4011be:e8 9d ff ff ff          callq  401160 <_ZN3Cat5noiseEv>
 ...
```

# Object-Oriented Programming

## Implementation

```
int main() {
    Animal animal1;
    Cat animal2;

    animal1.noise();
    animal2.noise();

    return 0;
}
```

First call jumps to the `Animal`'s definition

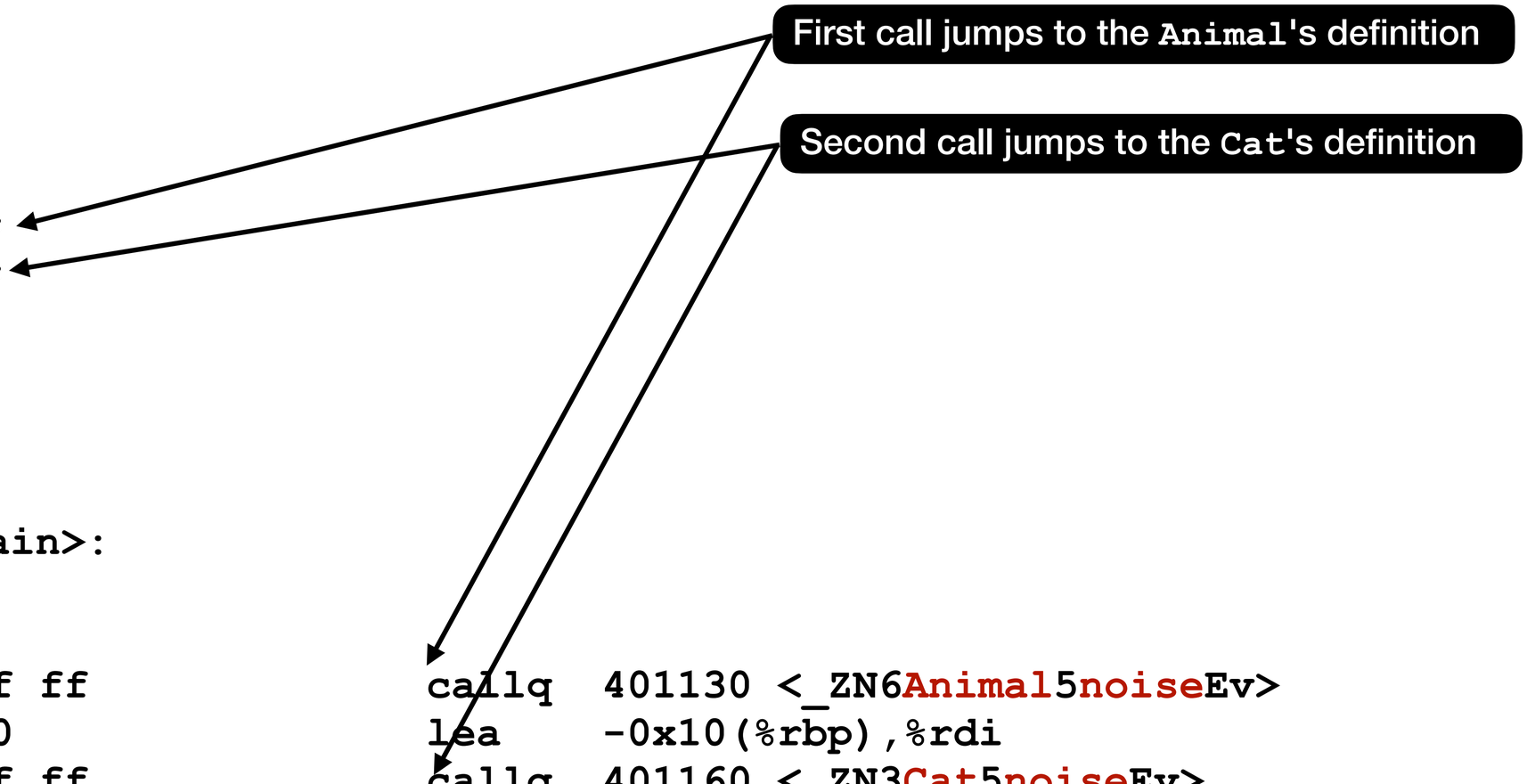
```
0000000000401190 <main>:

 ...
  4011b5:e8 76 ff ff ff      callq  401130 <_ZN6Animal5noiseEv>
  4011ba:48 8d 7d f0         lea    -0x10(%rbp),%rdi
  4011be:e8 9d ff ff ff      callq  401160 <_ZN3Cat5noiseEv>
 ...
```

# Object-Oriented Programming

## Implementation

```
int main() {
    Animal animal1;
    Cat animal2;

    animal1.noise();
    animal2.noise();

    return 0;
}
```

```
0000000000401190 <main>:

 ...
  4011b5:e8 76 ff ff ff      callq  401130 <_ZN6Animal5noiseEv>
  4011ba:48 8d 7d f0         lea    -0x10(%rbp),%rdi
  4011be:e8 9d ff ff ff      callq  401160 <_ZN3Cat5noiseEv>
 ...
```



# Object-Oriented Programming

## Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise())  # meow
print(animal2.reduce_noise())  # woof
```

# Object-Oriented Programming

## Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise())  # meow
print(animal2.reduce_noise())  # woof
```

- But that trick breaks down in this example :(

# Object-Oriented Programming

## Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise())  # meow
print(animal2.reduce_noise())  # woof
```

- But that trick breaks down in this example :(
- In `Animal`, `self` has type `Animal`. Wouldn't it just jump to `Animal`'s `noise`?

# Object-Oriented Programming

## Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise())  # meow
print(animal2.reduce_noise())  # woof
```

- But that trick breaks down in this example :(
- In `Animal`, `self` has type `Animal`. Wouldn't it just jump to `Animal`'s `noise`?
- In fact, C++ does get this wrong in this case!

# Object-Oriented Programming

## Implementation

```
int main() {
    Dog animal1;
    Cat animal2;

    printf("%s\n", animal1.reduce_noise().c_str());
    printf("%s\n", animal2.reduce_noise().c_str());

    return 0;
}
```

```
byronzhong@linux1:~$ ./a.out
generic sine wave
generic sine wave
```

# Object-Oriented Programming

## Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise())  # meow
print(animal2.reduce_noise())  # woof
```

- But that trick breaks down in this example :(
- In `Animal`, `self` has type `Animal`. Wouldn't it just jump to `Animal`'s `noise`?
- In fact, C++ does get this wrong in this case!

# Object-Oriented Programming

## Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise())  # meow
print(animal2.reduce_noise())  # woof
```

- But that trick breaks down in this example :(
- In `Animal`, `self` has type `Animal`. Wouldn't it just jump to `Animal`'s `noise`?
- In fact, C++ does get this wrong in this case!
- Compiler cannot decide at compile-time what functions to call

# Object-Oriented Programming

## Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise())  # meow
print(animal2.reduce_noise())  # woof
```

- But that trick breaks down in this example :(
- In `Animal`, `self` has type `Animal`. Wouldn't it just jump to `Animal`'s `noise`?
- In fact, C++ does get this wrong in this case!
- Compiler cannot decide at compile-time what functions to call
- We need to call the appropriate function based on the object's type during runtime

# Object-Oriented Programming

## Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise())  # meow
print(animal2.reduce_noise())  # woof
```

- But that trick breaks down in this example :(
- In `Animal`, `self` has type `Animal`. Wouldn't it just jump to `Animal`'s `noise`?
- In fact, C++ does get this wrong in this case!
- Compiler cannot decide at compile-time what functions to call
- We need to call the appropriate function based on the object's type during runtime
- Dynamic dispatch!

# Object-Oriented Programming

## Union

```
struct number {
        int i;
        float f;
        long l;
        double d;
};
```

```
union number {
        int     i;
        long    l;
        float   f;
        double d;
};
```

# Object-Oriented Programming

## Union

```
struct number {
        int i;
        float f;
        long l;
        double d;
};
```

- A structure has all the fields

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

# Object-Oriented Programming

## Union

```
struct number {
        int i;
        float f;
        long l;
        double d;
};
```

- A structure has all the fields

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

- A union has one of the field at a time

# Object-Oriented Programming

## Union

```
struct number {
        int i;
        float f;
        long l;
        double d;
};
```

- A structure has all the fields
- The size of of a structure is (roughly) the sum of the sizes of all the fields

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

- A union has one of the field at a time

# Object-Oriented Programming

## Union

```
struct number {
        int i;
        float f;
        long l;
        double d;
};
```

- A structure has all the fields
- The size of of a structure is (roughly) the sum of the sizes of all the fields

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

- A union has one of the field at a time
- The size of a union is the maximum of the sizes of all the fields

# Object-Oriented Programming

## Union

```
struct number {
        int i;
        float f;
        long l;
        double d;
};
```

- A structure has all the fields
- The size of of a structure is (roughly) the sum of the sizes of all the fields

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

- A union has one of the field at a time
- The size of a union is the maximum of the sizes of all the fields

# Object-Oriented Programming

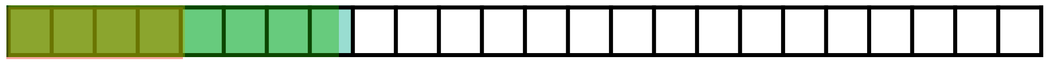
## Union

```
struct number {
        int i;
        float f;
        long l;
        double d;
};
```

- A structure has all the fields
- The size of of a structure is (roughly) the sum of the sizes of all the fields

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

- A union has one of the field at a time
- The size of a union is the maximum of the sizes of all the fields



# Object-Oriented Programming

## Union

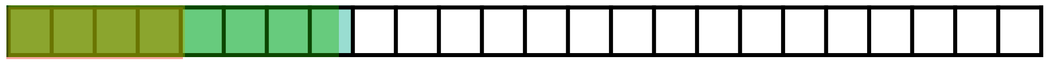
```
struct number {
        int i;
        float f;
        long l;
        double d;
};
```

- A structure has all the fields
- The size of of a structure is (roughly) the sum of the sizes of all the fields

- `n.l` selects the `l` field from the struct. (address offset from the start)

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

- A union has one of the field at a time
- The size of a union is the maximum of the sizes of all the fields



# Object-Oriented Programming

## Union

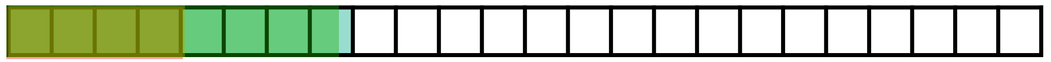
```
struct number {
        int i;
        float f;
        long l;
        double d;
};
```

- A structure has all the fields
- The size of of a structure is (roughly) the sum of the sizes of all the fields

- `n.l` selects the `l` field from the struct. (address offset from the start)

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

- A union has one of the field at a time
- The size of a union is the maximum of the sizes of all the fields



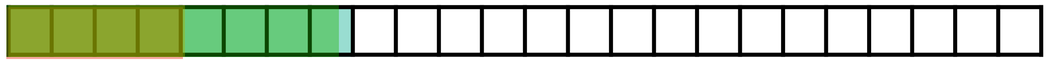
- `n.l` interprets the bits as a `long` (same piece of data)

# Object-Oriented Programming

## Union

```
union number {
        int     i;
        long    l;
        float   f;
        double d;
};
```

- A union has one of the field at a time
- The size of a union is the maximum size of all the fields



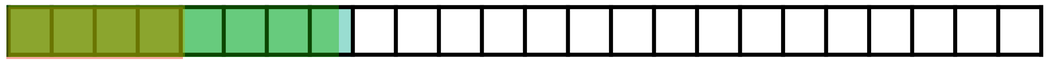
- `n.l` interprets the bits as a `long` (same piece of data)

# Object-Oriented Programming

## Union

```
union number {
        int     i;
        long    l;
        float   f;
        double d;
};
```

- A union has one of the field at a time
- The size of a union is the maximum size of all the fields



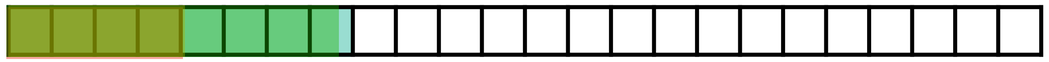
- `n.l` interprets the bits as a `long` (same piece of data)
- But... how do we know what is the correct way to interpret the data?

# Object-Oriented Programming

## Union

```
union number {
        int     i;
        long    l;
        float   f;
        double d;
};


void print_number(union number n)
{
        printf("??", n.??);
}
```

- A union has one of the field at a time
- The size of a union is the maximum size of all the fields



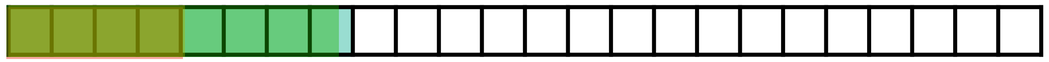
- `n.l` interprets the bits as a `long` (same piece of data)
- But... how do we know what is the correct way to interpret the data?

# Object-Oriented Programming

## Union

```
union number {
        int     i;
        long    l;
        float   f;
        double d;
};


void print_number(union number n)
{
        printf("??", n.??);
}
```

- A union has one of the field at a time
- The size of a union is the maximum size of all the fields



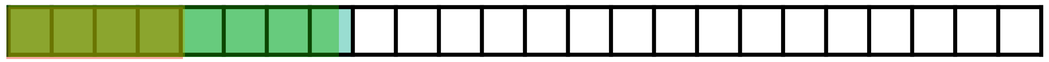
- `n.l` interprets the bits as a `long` (same piece of data)
- But... how do we know what is the correct way to interpret the data?
- We don't!

# Object-Oriented Programming

## Union

```
union number {
        int     i;
        long    l;
        float   f;
        double d;
};


void print_number(union number n)
{
        printf("??", n.??);
}
```

- A union has one of the field at a time
- The size of a union is the maximum size of all the fields



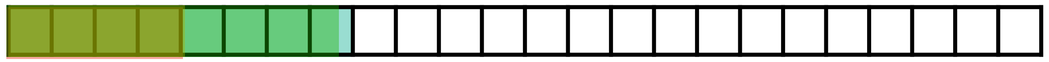
- `n.l` interprets the bits as a `long` (same piece of data)
- But... how do we know what is the correct way to interpret the data?
- We don't!
- There is no way to ask C which of the union it was set to before

# Object-Oriented Programming

## Union

```
union number {
        int     i;
        long    l;
        float   f;
        double d;
};


void print_number(union number n)
{
        printf("??", n.??);
}
```

- A union has one of the field at a time
- The size of a union is the maximum size of all the fields



- `n.l` interprets the bits as a `long` (same piece of data)
- But... how do we know what is the correct way to interpret the data?
- We don't!
- There is no way to ask C which of the union it was set to before
- But we can/must keep track of this ourselves

# Object-Oriented Programming

## Tagged Union

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
int main(void)
{
        struct tagged_number n;
        n.number.i = 4;
        n.tag = INT;

        print_number(n);

        return 0;
}
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
int main(void)
{
        struct tagged_number n;
        n.number.i = 4;
        n.tag = INT;

        print_number(n);

        return 0;
}
```

The programmer needs to make sure the tag is set correctly.

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int     i;
        long    l;
        float   f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int     i;
        long    l;
        float   f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int     i;
        long    l;
        float   f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int     i;
        long    l;
        float   f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int     i;
        long    l;
        float   f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int     i;
        long    l;
        float   f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
```

```
case DOUBLE:
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
        case DOUBLE:
                printf("%f\n", tn.number.d);
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
        case DOUBLE:
                printf("%f\n", tn.number.d);
                break;
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
        case DOUBLE:
                printf("%f\n", tn.number.d);
                break;
        default:
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
        case DOUBLE:
                printf("%f\n", tn.number.d);
                break;
        default:
                assert(0);
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
        case DOUBLE:
                printf("%f\n", tn.number.d);
                break;
        default:
                assert(0);
        }
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
        case DOUBLE:
                printf("%f\n", tn.number.d);
                break;
        default:
                assert(0);
        }
}
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

We only choose to interpret it as an integer after matching on the tag!

```
void print_nu
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
        case DOUBLE:
                printf("%f\n", tn.number.d);
                break;
        default:
                assert(0);
        }
}
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
        case DOUBLE:
                printf("%f\n", tn.number.d);
                break;
        default:
                assert(0);
        }
}
```

# Object-Oriented Programming

## Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
        case DOUBLE:
                printf("%f\n", tn.number.d);
                break;
        default:
                assert(0);
        }
}
```

Hold on, we're inspecting the type of a value and doing something different based on its type. DYNAMIC DISPATCH!!

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

```
enum animal_tag {
        CAT,
        DOG,
};
```

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

```
enum animal_tag {
        CAT,
        DOG,
};

struct cat {
        const char *name;
        /* other fields */
};

struct dog {
        const char *name;
        /* other fields */
};
```

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

```
enum animal_tag {
        CAT,
        DOG,
};

struct cat {
        const char *name;
        /* other fields */
};

struct dog {
        const char *name;
        /* other fields */
};

union animal {
        struct cat c;
        struct dog d;
};
```

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

```
enum animal_tag {
        CAT,
        DOG,
};

struct cat {
        const char *name;
        /* other fields */
};

struct dog {
        const char *name;
        /* other fields */
};

union animal {
        struct cat c;
        struct dog d;
};
```

```
struct tagged_animal {
        enum animal_tag tag;
        union animal animal;
};
```

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

```
enum animal_tag {
        CAT,
        DOG,
};

struct cat {
        const char *name;
        /* other fields */
};

struct dog {
        const char *name;
        /* other fields */
};

union animal {
        struct cat c;
        struct dog d;
};
```

```
struct tagged_animal {
        enum animal_tag tag;
        union animal animal;
};

const char *noise(struct tagged_animal animal)
{
        switch (animal.tag) {
        case CAT:
                return "Meow";
        case DOG:
                return "Woof";
        default:
                return "Generic sine wave";
        }
}
```

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

```
#include <tagged-union-demo>
```

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

- A union is a type that can be one of the declared fields at a given time

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

- A union is a type that can be one of the declared fields at a given time
- The fields overlap in memory

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

- A union is a type that can be one of the declared fields at a given time
- The fields overlap in memory
- C doesn't keep track of which field was set in a union and C doesn't prevent you from selecting the wrong union fields.

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

- A union is a type that can be one of the declared fields at a given time
- The fields overlap in memory
- C doesn't keep track of which field was set in a union and C doesn't prevent you from selecting the wrong union fields.
- When you select the wrong field, you choose a wrong interpretation of the bits and potentially read some uninitialized bits.

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

- A union is a type that can be one of the declared fields at a given time
- The fields overlap in memory
- C doesn't keep track of which field was set in a union and C doesn't prevent you from selecting the wrong union fields.
- When you select the wrong field, you choose a wrong interpretation of the bits and potentially read some uninitialized bits.
- A common way to keep track of the correct interpretation is to use *tagged union*.

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

- A union is a type that can be one of the declared fields at a given time
- The fields overlap in memory
- C doesn't keep track of which field was set in a union and C doesn't prevent you from selecting the wrong union fields.
- When you select the wrong field, you choose a wrong interpretation of the bits and potentially read some uninitialized bits.
- A common way to keep track of the correct interpretation is to use *tagged union*.
  - structure of an enum and a union

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

- A union is a type that can be one of the declared fields at a given time
- The fields overlap in memory
- C doesn't keep track of which field was set in a union and C doesn't prevent you from selecting the wrong union fields.
- When you select the wrong field, you choose a wrong interpretation of the bits and potentially read some uninitialized bits.
- A common way to keep track of the correct interpretation is to use *tagged union*.
  - structure of an enum and a union
  - Enum keeps track of the alternatives, and the union stores the data of one of them.

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

- However, a tagged union is not extensible

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

- However, a tagged union is not extensible
- If we want to add another animal, every function needs to be changed.

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

- However, a tagged union is not extensible
- If we want to add another animal, every function needs to be changed.

```
enum animal_tag {
        CAT,
        DOG,
        ALLIGATOR,
};
```

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

- However, a tagged union is not extensible
- If we want to add another animal, every function needs to be changed.

```
enum animal_tag {
        CAT,
        DOG,
        ALLIGATOR,
};
```

```
const char *noise(struct animal *a)
{
        switch (a->tag) {
        ...
        case ALLIGATOR:
                ...
        ...
        }
}
```

# Object-Oriented Programming

## Dynamic Dispatch via Tagged Union

- However, a tagged union is not extensible
- If we want to add another animal, every function needs to be changed.

```
enum animal_tag {
        CAT,
        DOG,
        ALLIGATOR,
};
```

```
const char *noise(struct animal *a)
{
        switch (a->tag) {
        ...
        case ALLIGATOR:
                ...
        ...
        }
}
```

```
void walk(struct animal *a)
{
        switch (a->tag) {
        ...
        case ALLIGATOR:
                ...
        ...
        }
}
```

# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table

# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table

- Use pointers!

# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table

- Use pointers!
- This is how most languages implement dynamic dispatch.

# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table

- Use pointers!
- This is how most languages implement dynamic dispatch.
- Each *subclass* gets a *virtual table* (*vtable*), filled with pointers to its overridden functions.

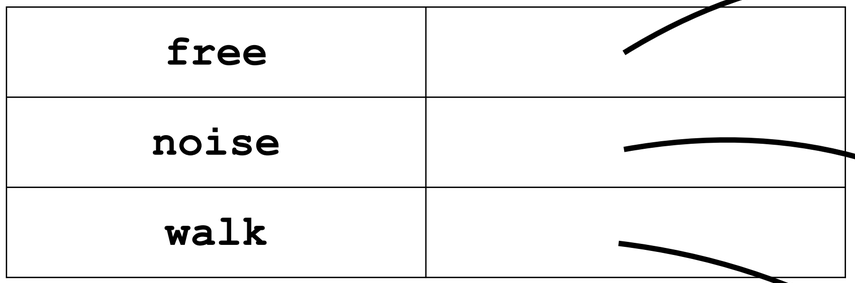
# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table

- Use pointers!
- This is how most languages implement dynamic dispatch.
- Each *subclass* gets a *virtual table* (*vtable*), filled with pointers to its overridden functions.
- Each subclass *object* has a pointer to its class's virtual table, which is used to resolve function calls at runtime.

# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table




Cat's virtual table

```c
void cat_free(struct animal *animal)
{
        struct cat *c = (struct cat *)animal;
        free(c->cat_breed);
        free(c);
}

void cat_noise(struct animal *animal)
{
        (void) animal;
        printf("%s\n", "Meow");
}

void cat_walk(struct animal *animal)
{
        struct cat *c = (struct cat *)animal;

        printf("cat %s, %s, walking\n", c->name, c->cat_breed);
}
```

# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table

| free | |
|---|---|
| noise | |
| walk | |

Cat's virtual table

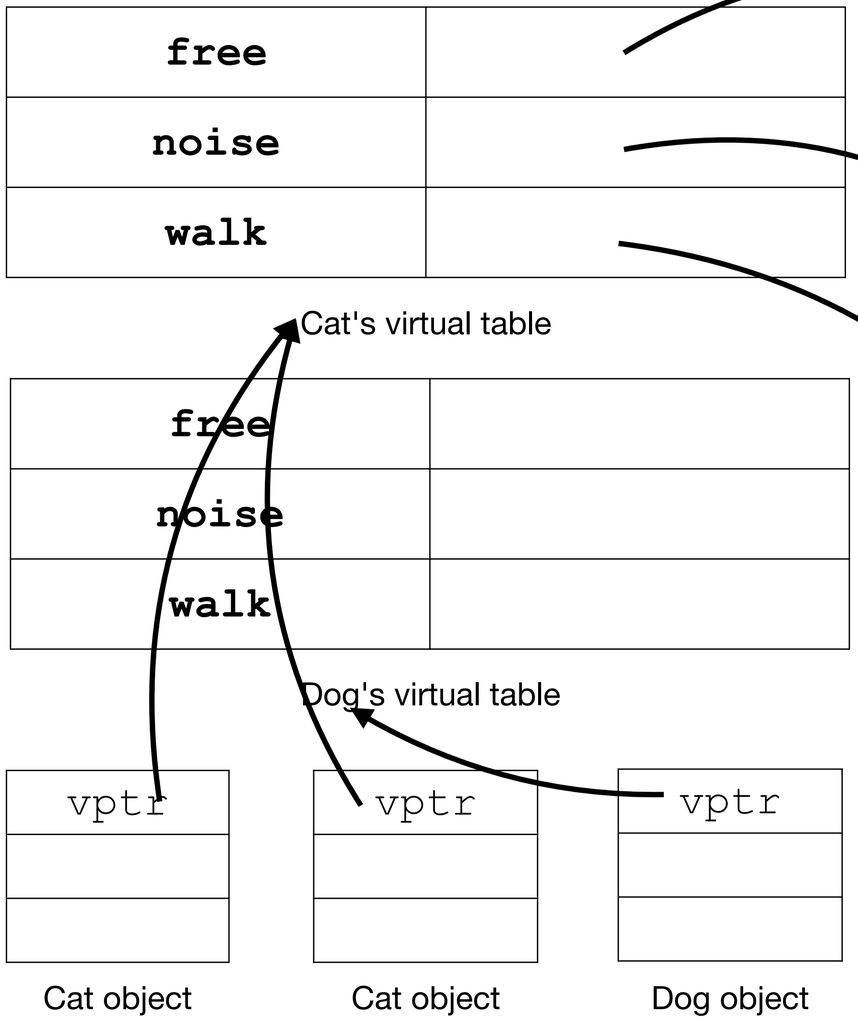
| free | |
|---|---|
| noise | |
| walk | |

Dog's virtual table

```c
void cat_free(struct animal *animal)
{
        struct cat *c = (struct cat *)animal;
        free(c->cat_breed);
        free(c);
}

void cat_noise(struct animal *animal)
{
        (void) animal;
        printf("%s\n", "Meow");
}

void cat_walk(struct animal *animal)
{
        struct cat *c = (struct cat *)animal;

        printf("cat %s, %s, walking\n", c->name, c->cat_breed);
}
```

# Object-Oriented Programming

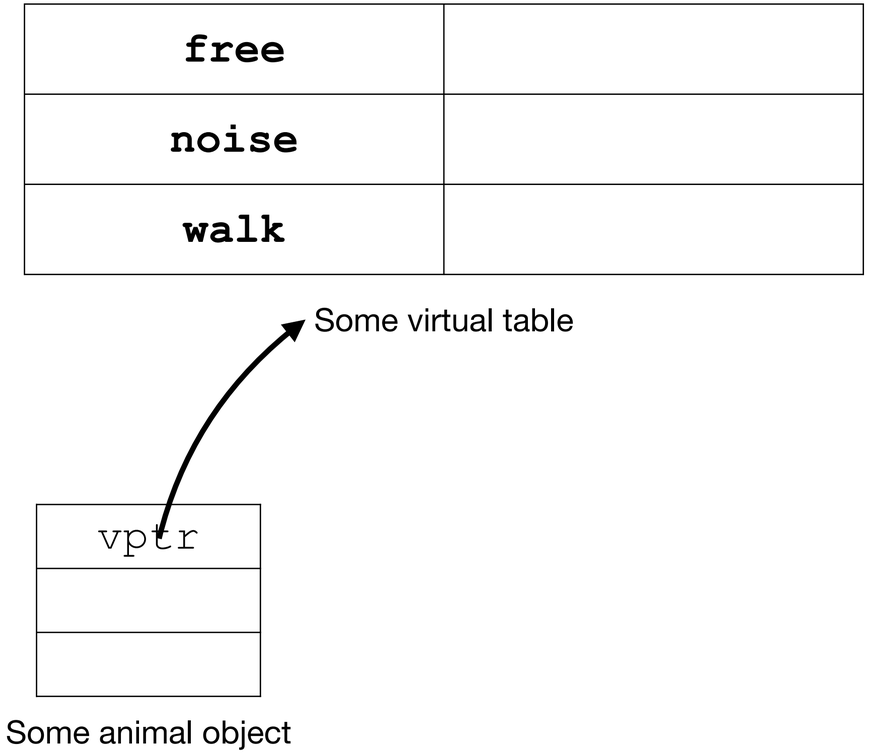
## Dynamic Dispatch via Virtual Table




```c
void cat_free(struct animal *animal)
{
        struct cat *c = (struct cat *)animal;
        free(c->cat_breed);
        free(c);
}

void cat_noise(struct animal *animal)
{
        (void) animal;
        printf("%s\n", "Meow");
}

void cat_walk(struct animal *animal)
{
        struct cat *c = (struct cat *)animal;

        printf("cat %s, %s, walking\n", c->name, c->cat_breed);
}
```

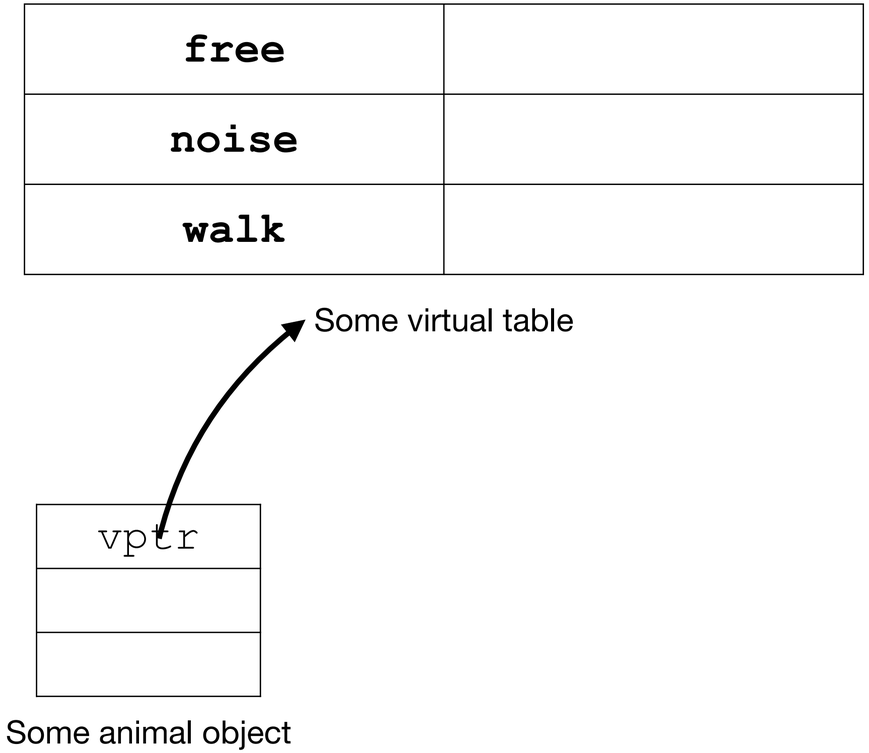
# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table



# Object-Oriented Programming

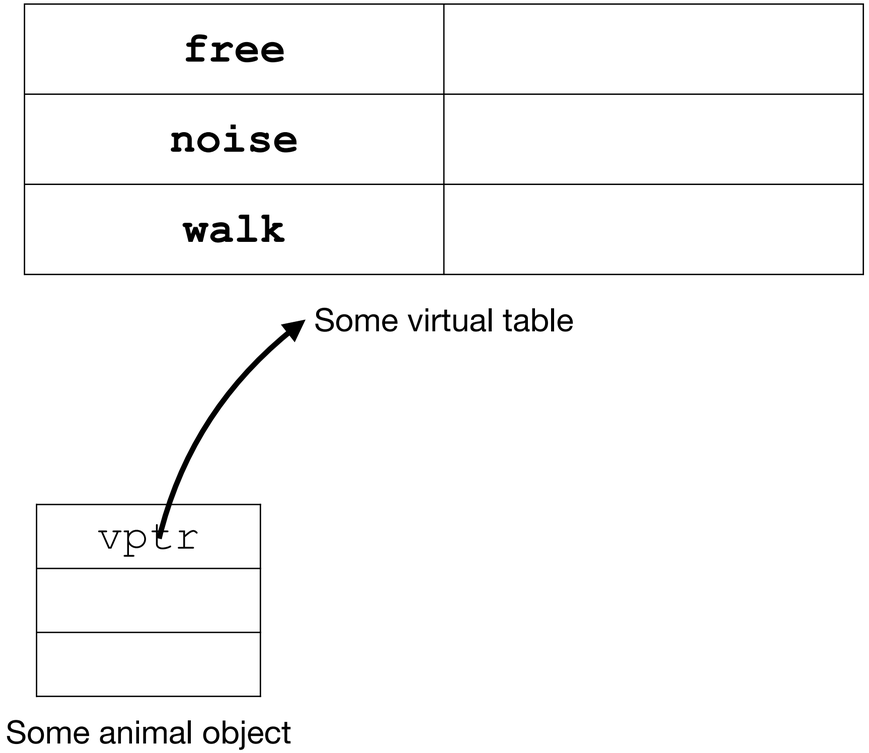
## Dynamic Dispatch via Virtual Table




- If the animal is a cat, we know that the vptr points to Cat's virtual table

# Object-Oriented Programming

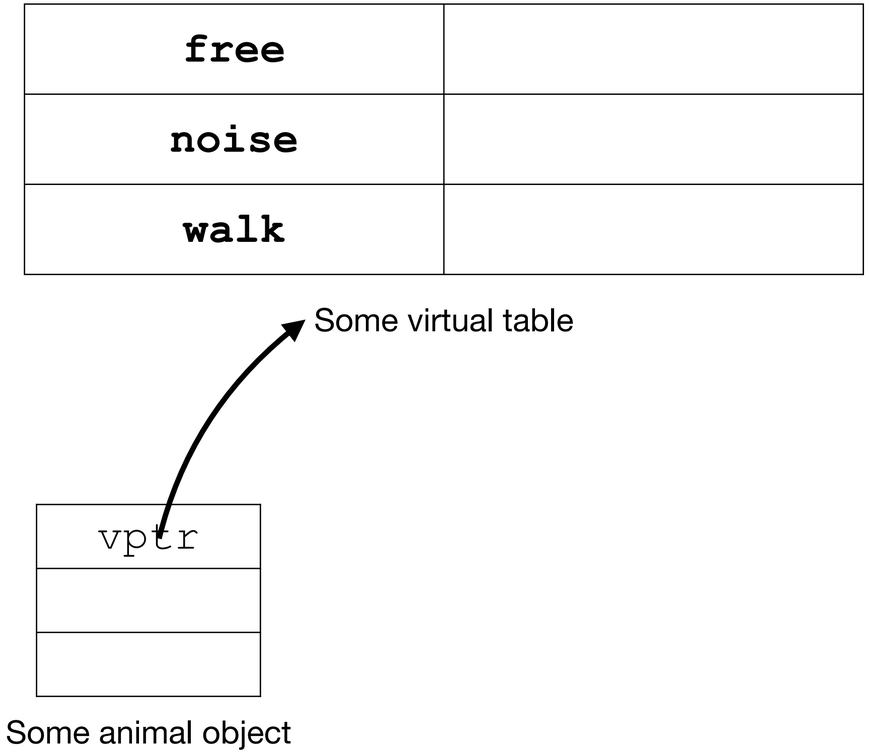
## Dynamic Dispatch via Virtual Table




- If the animal is a cat, we know that the vptr points to Cat's virtual table
- If the animal is a dog, we know that the vptr points to Dog's virtual table

# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table



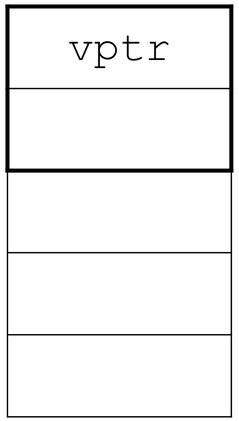

- If the animal is a cat, we know that the vptr points to Cat's virtual table
- If the animal is a dog, we know that the vptr points to Dog's virtual table
- `animal->vptr->noise()` calls the correct noise function via the virtual table

# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table

# Object-Oriented Programming

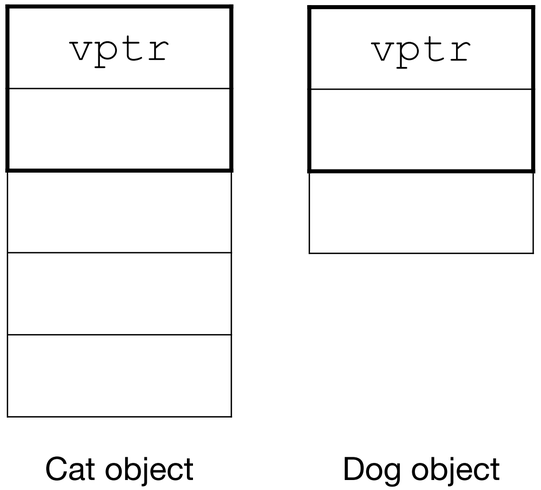
## Dynamic Dispatch via Virtual Table




Cat object

# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table

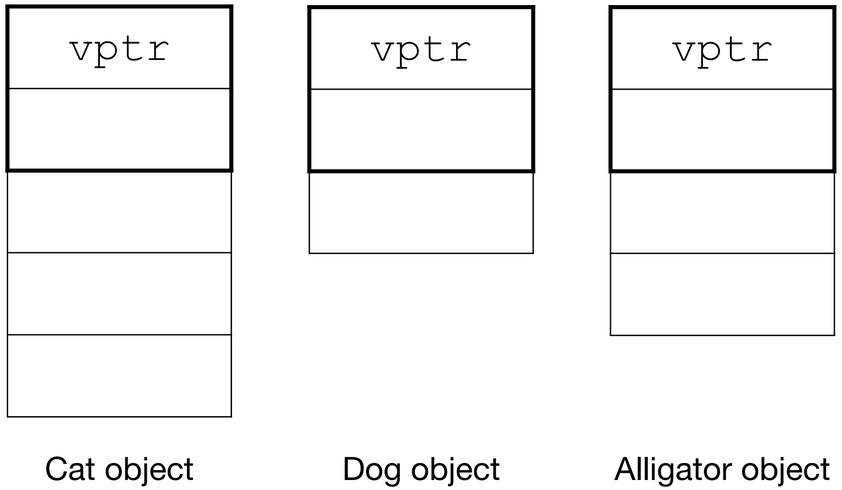



Cat object

Dog object

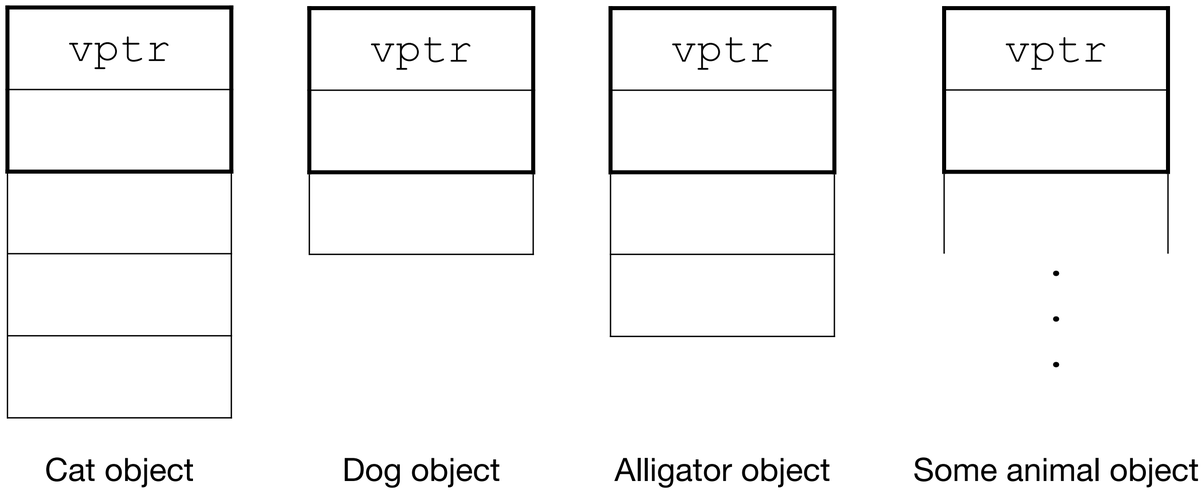
# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table



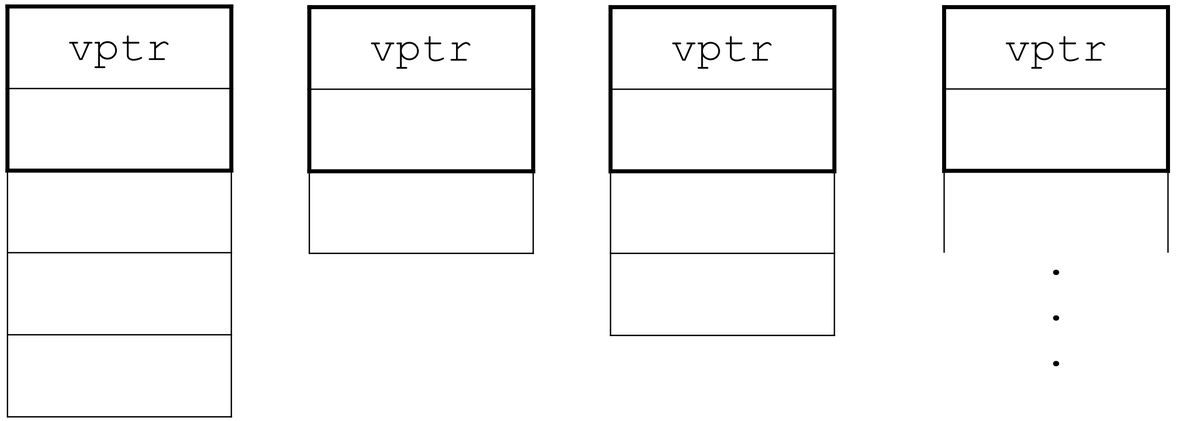
# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table



# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table

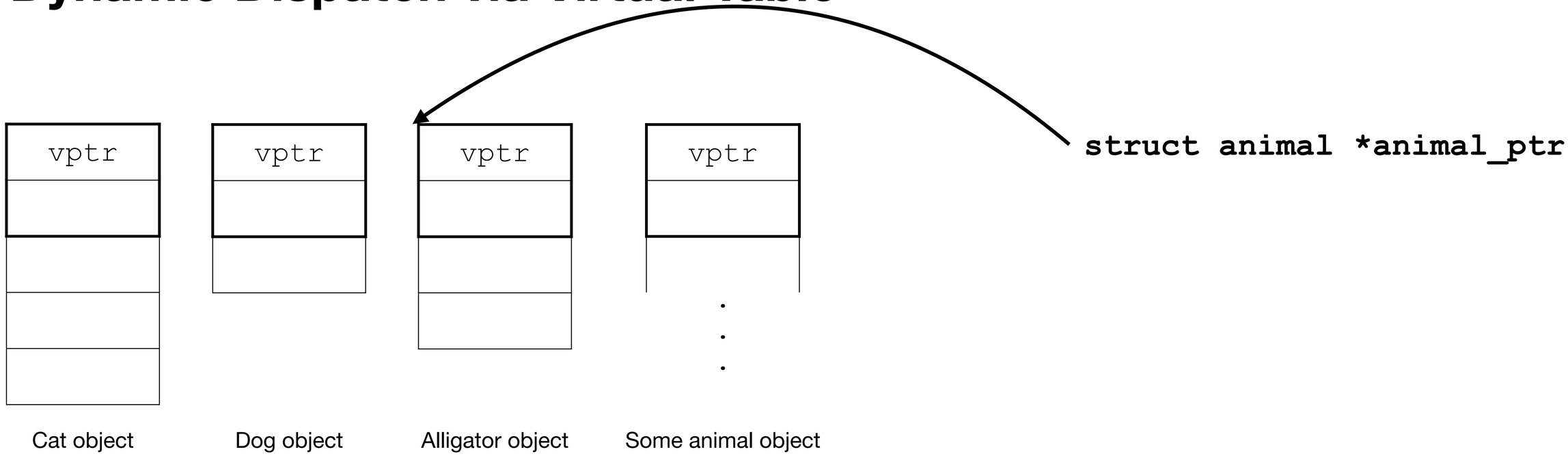



Cat object Dog object Alligator object Some animal object

```
struct animal *animal_ptr
```

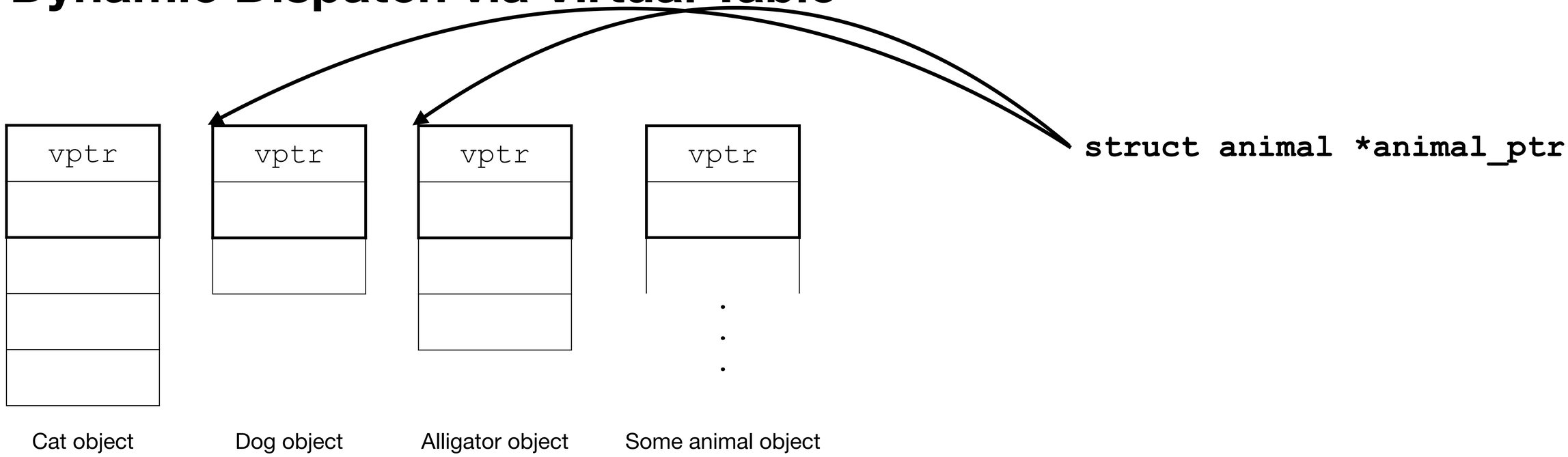
# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table



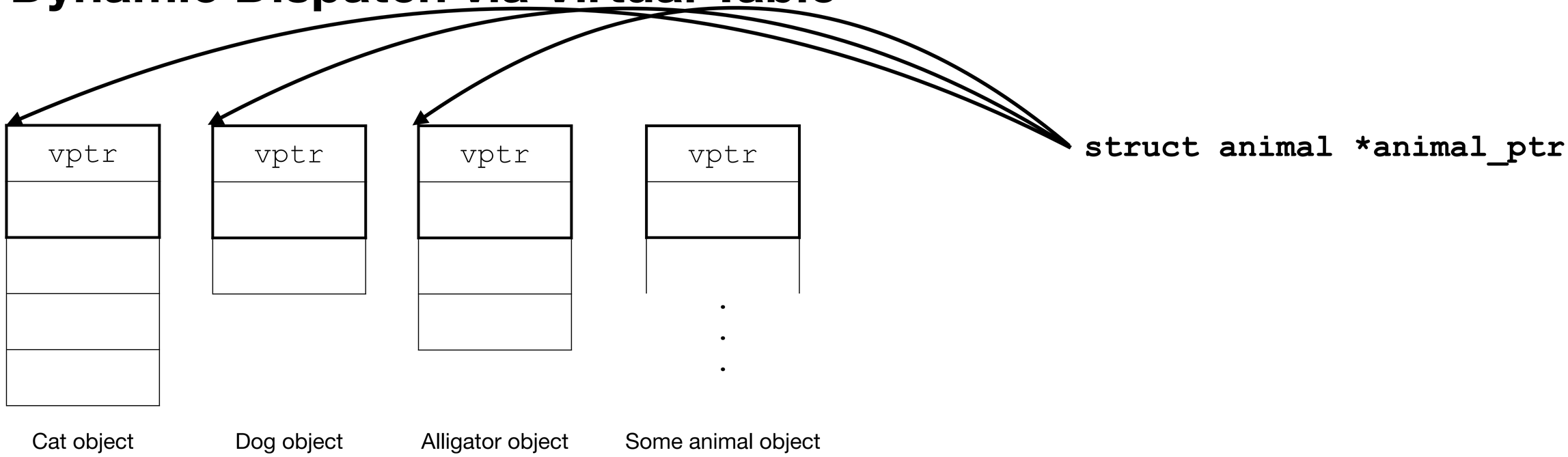
# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table



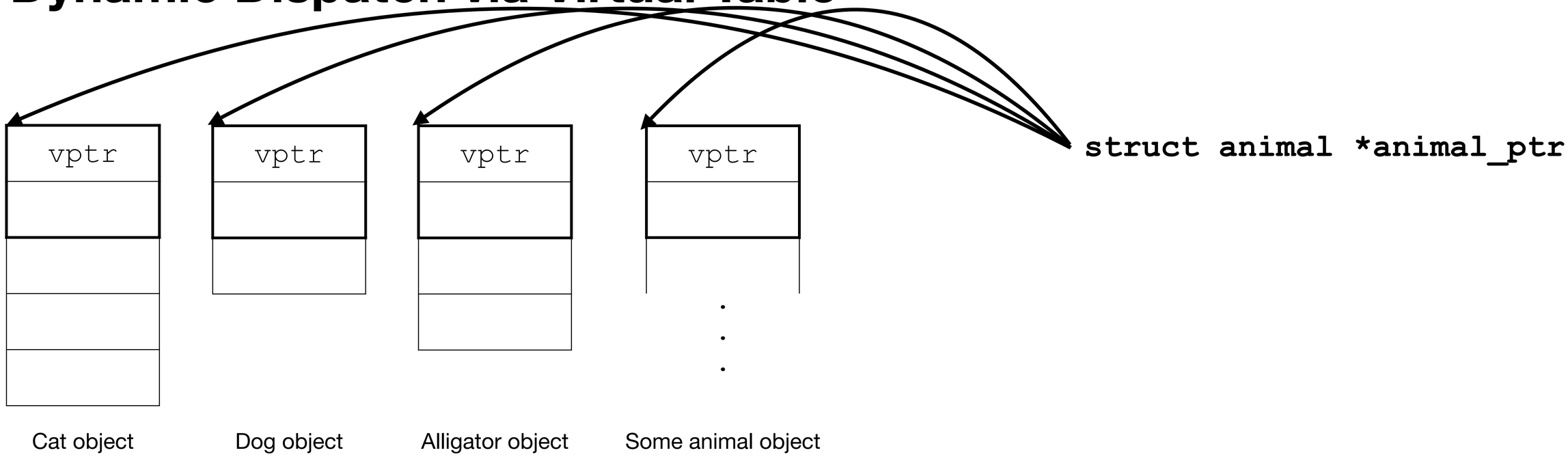
# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table



# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table



# Object-Oriented Programming

## Dynamic Dispatch via Virtual Table

```
#include <vtable-demo>
```