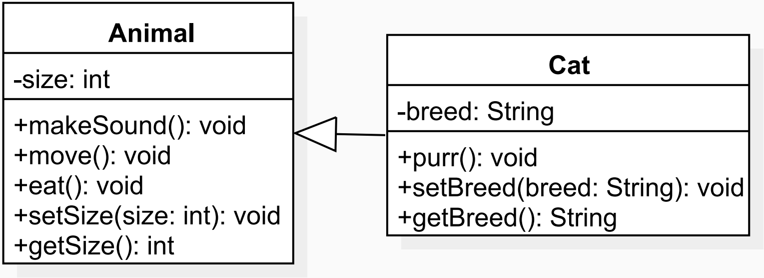
POLYMORPHISM

Object behavior in different contexts

Florian Warg, Max Staff

May 2, 2017

- polymorphism is the ability of an object to take on many forms
- if an object passes more than 1 ”is-a” tests, it is considered polymorphic
- i.e. if a class is derived from another class, its objects are polymorphic
- 

- objects of ”Cat” are polymorphic because they are of type ”Cat” and ”Animal”

- polymorphism only works with reference types (in all languages)

```
Cat felix;
felix.purr();
Animal& alsoFelix = felix;
alsoFelix.purr(); // error
felix.getSize();  // OK
```

- felix is a cat and also an animal
- i.e. felix is polymorphic
- we can treat felix as a generic animal instead of a cat by using references or pointers
- now only the ”animal” interface is exposed

IMPLICIT REFERENCE CASTS

- the compiler can implicitly cast a reference to a parent-type reference
- this allows functions to accept multiple types

```
void printSize(const Animal& animal) {
    cout << animal.getSize();
}
Cat felix;
printSize(felix); // OK
```

- what happens when Cat has a method with the same name

```
class Animal {
public:
    void makeSound() {
        cout << "Animal::makeSound()\n"; }
};
class Cat : public Animal {
public:
    void makeSound() {
        cout << "Cat::makeSound()\n"; }
};
Cat felix;
Animal& alsoFelix = felix;
felix.makeSound();     // Cat::makeSound()
alsoFelix.makeSound(); // Animal::makeSound()
```

- a child class can override a method of its parent
- a method is overridable if it is virtual
- use `override` keyword for compiler checks

```
class Animal {
public:
    virtual void makeSound();
};
class Cat : public Animal {
public:
    void makeSound() override {
        cout << "Cat::makeSound()\n"; }
};
Cat felix;
Animal& alsoFelix = felix;
felix.makeSound();     // Cat::makeSound()
alsoFelix.makeSound(); // Cat::makeSound()
```

```
struct A {
    int i;
    string str;
};
```

- construct i
- construct str
- call `A()`
- call `~A()`
- destruct str
- destruct i

LIFETIME WITH INHERITANCE

```
struct A { /* ... */ };
struct B : public A {
    int i;
    string str;
};
```

- construct A
- construct i
- construct str
- call `B()`
- call `~B()`
- destruct str
- destruct i
- destruct A

- consider this primitive implementation of a delete function

```
delete(T* ptr) {
    ptr->~T();
    free(ptr);
}
class Parent {};
class Child : public Parent {};
Parent* p = new Child();
delete(p); // what happens here?
```

- our defined delete function behaves like the real keyword
- since p points to a Parent object, only the parent dtor is invoked
- we already know how to fix this (-> virtual functions)

NO MORE IMPLICIT MEMORY LEAKS

```
class Parent {
public:
    virtual ~Parent() = default;
};
class Child : public Parent {};
Parent* p = new Child();
delete p;
```

- now the child dtor overrides the parent dtor
- make sure to have a virtual destructor if you want to derive from a class
- introducing virtual functions comes with an overhead
- each class needs a vtable, objects need a vtable pointer
- when calling functions, virtual dispatch is needed