

POLYMORPHISM

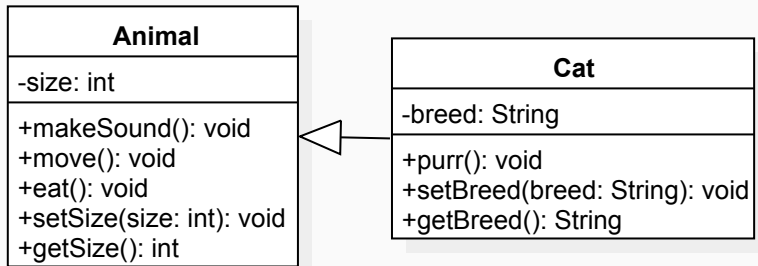
Object behavior in different contexts

Florian Warg, Max Staff

May 10, 2017

WHAT IS POLYMORPHISM?

- 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
```

DIFFERENT BEHAVIOR

- 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()
```

OVERRIDING METHODS

- 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

```
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

PURE VIRTUAL FUNCTIONS

- pure virtual functions are incomplete virtual functions
- the implementation has to be provided by a derived class by overriding

```
struct Parent {  
    virtual void implementMe() = 0;  
};  
struct Child : public Parent {  
    void implementMe() override {  
        cout << "implemented by child\n";  
    }  
};
```

- classes are abstract if they contain at least 1 pure virtual function
 - abstract classes cannot be instantiated
 - however pointers and references are allowed
-

```
/* ... */  
struct Parent;  
struct Child;  
Parent p; // error - Parent is abstract  
Child c;  // ok  
Parent& alsoC = c;  
alsoC.implementMe();
```

- classes can also be made abstract by having a pure virtual destructor
- however, since every class needs a dtor, we still have to provide one

```
class Abstract {  
public:  
    virtual ~Abstract() = 0;  
};  
Abstract::~~Abstract() {}
```

- interfaces are abstract classes that only contain pure virtual functions

```
struct IMath {  
    virtual int max(int a, int b) = 0;  
    virtual int min(int a, int b) = 0;  
    virtual int square(int x) = 0;  
};  
class IntMath : public IMath {  
public:  
    int max(int a, int b) override { /* ... */}  
    int min(int a, int b) override { /* ... */}  
    int square(int x) override { /* ... */ }  
};
```
