13 - Polymorphism

March 8, 2023

COMP2404

Darryl Hill

Contents

1. Overview
2. Pointers and Class Hierarchy
3. Dynamic Binding
4. Abstract Classes
5. Dynamic Casting (Bad)
6. Design Patterns
7. Encapsulating Behaviour
8. Strategy Design Pattern

Overview

What is polymorphism?

poly - many

morph - shape

- Different behaviour (but same category of behaviour) from different objects using the same function call

Example: a call to `update()` on a video game object can result in different behaviour

- player character takes player input to determine how to `update()`
- enemies use AI to determine how to `update()`

Overview

Polymorphism is a construct for making programming easier.

- Polymorphism has no meaning at lower (compiled) levels
  - It is at most a table lookup for which function to call
  - It is a compiler trick.

- An abstraction layer hiding details so that the programmer can focus on algorithms.
  - We call a similar function shared between related classes.
  - The ***compiler*** decides which function to call.

Polymorphism is often the mechanism behind design patterns.

- We create generalized interfaces.
- Different implementations are substituted at runtime.

Overview

Polymorphism is implemented slightly differently in different languages.

- Though all use a generalized handle to call on a specific implementation.
- C++ uses ***inheritance***:
  - A ***base class*** handle (pointer or reference) to call on a ***derived class*** functionality.
  - Multiple inheritance allows objects to wear as many "different hats" as we need.
- Java may use `Superclasses` or `Interfaces` to achieve the same result.

Overview

Polymorphism - when different objects implement the same category of behaviour

- but in a specialized manner.

An `Animal` can `speak()`, but

- `Cat`s `speak()` by saying "meow"
- `Dog`s `speak()` by saying "woof".

Overview

Polymorphism:

- Allows each class in the hierarchy to implement its own behaviour
- Adding new derived classes do not effect existing classes
- Client classes do not need to know what the derived class is
  - Casting is not polymorphism.
  - Casting requires knowledge of the derived classes.
- Simply call the base class function, and the compiler figures out which derived class implementation to call.
- Polymorphism allows the writing of general algorithms.

Terminology:

- handle: an identifier used to access an object
  - variable
  - reference
  - pointer

Overview

In C++, client classes call a function on a **base class** handle.

- The handle points to a member of a **derived class**.
- The derived class has overridden the function.
- Correct function in the class hierarchy is chosen at runtime.
  - Usually through a table look-up.
  - This is known as ***dynamic binding***.
- Client class does not know which derived class is being used.
  - Good Encapsulation.
- The core algorithm stays the same
  - We can change or add implementation without rewriting core code
  - Implementation is separated from the public interface.

Overview

Polymorphism in C++:

- Must use inheritance.
  - We can have inheritance without polymorphism, but polymorphism requires inheritance.
  - There must be an "is-a" relationship between the base class handle and the derived class implementation.
  - In C++ "is-a" means ***inheritance***.
- Polymorphism can use handles that are pointers or references.
  - Derived class treated like a base class object using "is-a" relationship
  - We've seen calling overridden functions using scope resolution operator - this is different.

Static Binding

"Normal" inheritance in C++ uses ***static binding.***

- The compiler calls the function based on the ***handle type***, not the ***object type***.
- Since derived classes have an "is-a" relationship, they can use base class pointers and references.

```
Chicken* chicky = new Chicken;
chicky->speak();
Animal* chickyP = chicky;
chickyP->speak();
```

- This is the default behaviour in C++.

coding example <p1> - static binding

Pointers and Class Hierarchy

Type of Pointer (Static Binding)

	Type of Object	
	Base Class	Derived Class
Base Class		
Derived Class		

Pointers and Class Hierarchy

Type of Pointer (Static Binding)

	Type of Object	
	Base Class	Derived Class
Base Class	Base class function	Base class function
Derived Class	Error	Derived class function

Dynamic Binding

Problem:

- How do we invoke the derived class behaviour with a base class handle?
  - Scope resolution operator allows us to go up the hierarchy, but not down.

Solution:

- Dynamic binding.
- We must notify the compiler that we want dynamic binding.

coding example <p2>

Coding p2

When we call a function on a class, there are two types:

Static binding

- Happens at compile time - a single function address is supplied for a function call.
- If you use base class pointer, you get base class function, even with a derived class

Dynamic binding

- We use the `virtual` keyword in base class function to enable dynamic binding.
- There is a `vptr` in each variable that points to a table of function addresses of the underlying object.

Dynamic Binding

What is function ***binding***?

- Selecting the correct function to execute.
- Given overridden functions in a class hierarchy, compiler must select one.

Static binding

- Selection is made at compile time
- Always used when called on an object variable
- It *can* be used with pointers (using non-virtual functions)

Dynamic binding

- Selection is made at runtime
- Implemented using
  - pointers and virtual functions
  - references and virtual functions

Virtual Functions

What is a virtual function?

- A function that is selected for execution at runtime
  - All Java functions are virtual
  - C++ allows us to choose
- A lookup table is used based on the object type, not handle type
  - base class object calls base class function
  - derived class object calls derived class function

- "Virtual"-ness is inherited by the `virtual` keyword on the base class, all the way down the heirarchy

- By convention we repeat the `virtual` keyword on all derived classes
  - This is for the programmer - self-documenting code
  - A user of your subclass should be informed it is a virtual function

Virtual Functions

We can make Virtual ***destructors***

- Call the correct destructor based on ***type***

Non-virtual destructor:

- Calling destructor based on type of handle
  - So a base class destructor can be called on a derived class
  - Good idea or bad idea?
  - Can result in unpredictable behaviour
    - All destructors should be virtual if using polymorphism

coding example <p3>

- Non-virtual destructors - only top portion is called
- virtual destructors - all destructors are called

Virtual Functions

			Type of Object	
			Base Class	Derived Class
Non-Virtual	Pointer	Base Class		
		Derived Class		
Virtual	Pointer	Base Class		
		Derived Class		

Virtual Functions

			Type of Object	
			Base Class	Derived Class
Non-Virtual	Pointer	Base Class	`Base`	`Base`
		Derived Class	ERROR	`Derived`
Virtual	Pointer	Base Class	`Base`	Derived (Polymorphism)
		Derived Class	ERROR	`Derived`

Abstract Classes

Abstract class - any class with an abstract function.

- In C++, these are known as ***pure virtual*** functions
- It is a ***virtual function*** but with no implementation.

- In UML we represent these using *italics*

An abstract class ***cannot be instantiated***.

- It is typically used as a base class to specify an interface.

Abstract Classes

A ***pure virtual*** function is a virtual function with no implementation.

- In the class declaration (`.h` file) we would declare:

```
virtual void print() = 0;
```

- Assigning the function the value 0 means the implementation has a NULL memory address.
  - Thus we ***do not*** put an implementation in the `.cc` file.
- If a class has ***at least one*** pure virtual function it is an abstract class.
- A derived class ***must override*** this function or ***also be abstract***.

coding example <p4>

Virtual Functions and Data Structures

When using a data structure of objects, be careful. Consider the code:

```
vector<Animal> vec;

Chicken little;

vec.push_back(little);

vec[0].speak();
```

The `Animal::speak()` function is called

- Not `Chicken::speak()`
- `vec[0]` is not a `Chicken`, it is an `Animal`
  - Not a `Chicken` with an `Animal` handle
- We are copying the `Animal` parts of `Chicken little` into an `Animal` object
- Uses the ***assignment operator***.

coding example <p5>

Dynamic Casting

Polymorphism is the most effective and robust runtime binding. Alternatives:

RTTI - Run Time Type Information

- `typeid` operator used on an object
- returns a `type_info` object that we can ask for the class name
- not all compilers enable this by default

Dynamic casting (a type of RTTI)

- Attempt to cast an object to a particular handle

RTTI techniques are brittle

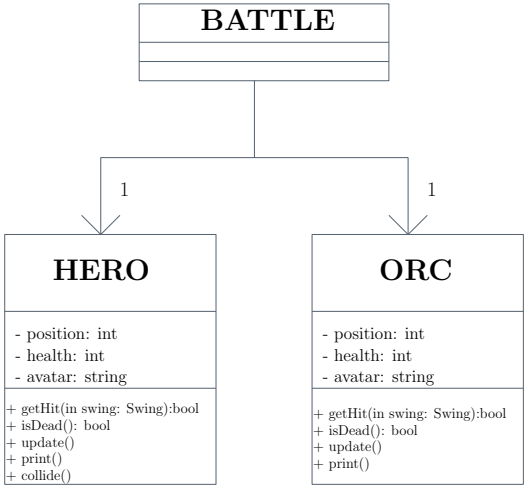
- Adding new subclasses can result in unexpected behaviour

coding example <p6>

Refactoring

Often we start implementing design patterns at a certain stage of development.

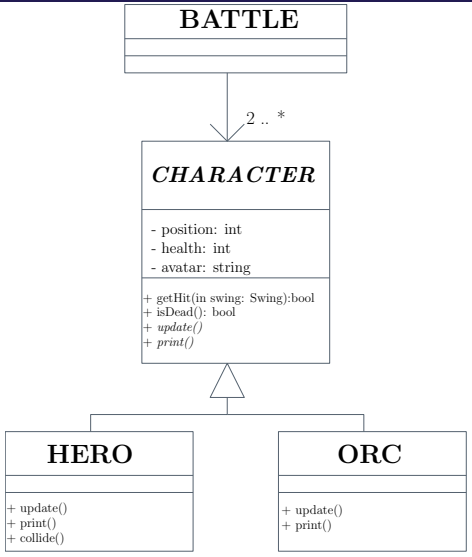
- We successfully make a game where a Hero fights an Orc
- Once we have debugged, we decide we want to add new characters.
  - This is not easy.
  - Time to refactor!



Refactoring

When we refactor, we want the functionality to stay the same.

- We are introducing ***abstraction layers*** to make updating easier.
- In this case we make an abstraction of `Hero` and `Orc` called `Character`.
- That way we can write general algorithms for `Character` and write details into the derived classes.
- New class - inherit from `Character` and write `update` and `print`.



Polymorphism Example

- We will have a battle between an Orc and a Hero
- The Hero is fast with good range, but does less damage
- The Orc is slow with short range, but does more damage
- Also has a special "collide" attack
- This is also an exercise in ***refactoring***
- **refactoring** - once your code reaches a certain complexity, re-organize the class structure
- often by extracting a base class
- Our goal is to be able to (easily) add new character classes

- **coding example <p7>**