Introductory Robot Programming

ENPM809Y

Lecture 09 – Object-oriented Programming (OOP) – Part III

Zeid Kootbally

zeidk@umd.edu

Fall 2020

Overview I

01 UML

Classes

02 Inheritance

Real-life Example

In C++

Domain

Base and Derived Constructors

Destructor

03 Polymorphism

Static Binding

Dynamic Binding

override Specifier

final Specifier

Overview II

- Abstract and Concrete Classes
- Abstract Classes
- Concrete Classes

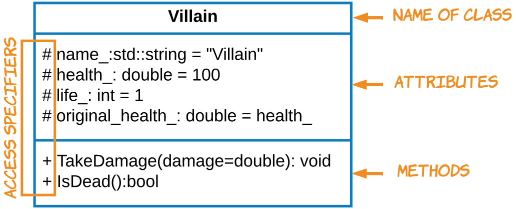
Highlights

- Final lecture on OOP.
- We will learn about the last 2 pillars of OOP: Inheritance and Polymorphism.
- Before we start, create a new project and copy main.cpp, villain.cpp, and villain.h from ELMS into this project.
    - ➤ CLion:Lecture9

UML | Classes

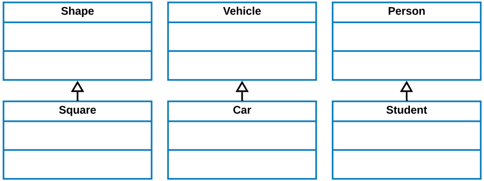
- UML (Unified Modeling Language) is a graphical language for modeling the structure and behavior of object-oriented systems. UML is widely used in industry to design, develop and document complex software.
- Class: A class diagram contains a rectangle for each class. It is divided into three parts.
  - The name of the class.
  - The names and types of the fields.

    + for `public` fields.
    - for `private` fields.
    # for `protected` fields.
  - The names, return types, and parameters of the methods.



Inheritance

- Inheritance is one of the 4 pillars of OOP and refers to a type of relationship wherein one associated class is a child of another by virtue of assuming the same functionalities of the parent class.
- In other words, the child class is a specific type of the parent class.
- Related classes have common attributes. Inheritance leverages these common attributes.
- It represents a is a relationship.
    - "A Square is a Shape"
    - "A Car is a Vehicle"
    - "A Student is a Person"

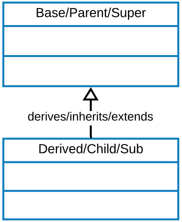


Inheritance

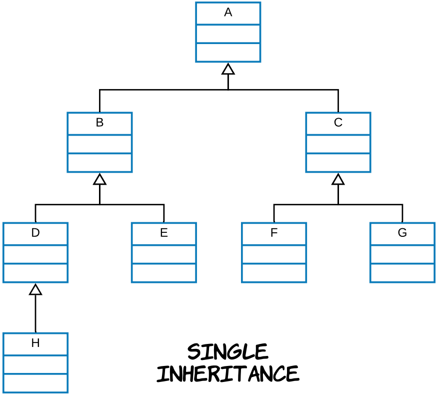
- There are 2 types of inheritance:
  - Single inheritance: A new class is created from another single class (most common).
  - Multiple inheritance: A new class can inherit characteristics and features from more than one parent class (not covered in this course).
    - C++ allows multiple inheritance but I strongly recommend you avoid it until you are really comfortable with single inheritance.
    - Once you are really comfortable with single inheritance, I still recommend you avoid multiple inheritance.

Inheritance

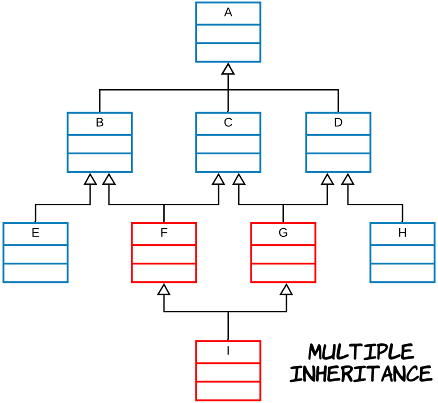
- A class that inherits from another class is called a derived class, a child class, or a subclass.
- A class from which other classes are derived is called a base class, a parent class, or a superclass.
- A derived class is said to derive, inherit, or extend a base class.



Inheritance

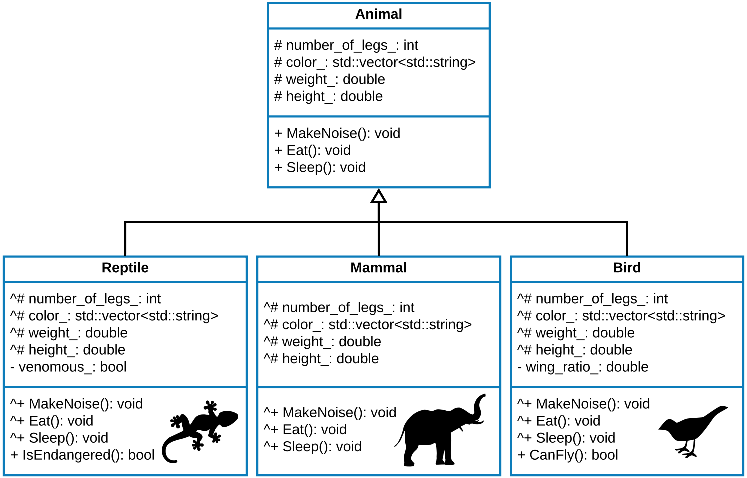


SINGLE INHERITANCE



MULTIPLE INHERITANCE

Inheritance | Real-life Example



Animal

# number_of_legs_: int
# color_: std::vector<std::string>
# weight_: double
# height_: double

+ MakeNoise(): void
+ Eat(): void
+ Sleep(): void

Reptile

^# number_of_legs_: int
^# color_: std::vector<std::string>
^# weight_: double
^# height_: double
- venomous_: bool

^+ MakeNoise(): void
^+ Eat(): void
^+ Sleep(): void
+ IsEndangered(): bool

Mammal

^# number_of_legs_: int
^# color_: std::vector<std::string>
^# weight_: double
^# height_: double

^+ MakeNoise(): void
^+ Eat(): void
^+ Sleep(): void

Bird

^# number_of_legs_: int
^# color_: std::vector<std::string>
^# weight_: double
^# height_: double
- wing_ratio_: double

^+ MakeNoise(): void
^+ Eat(): void
^+ Sleep(): void
+ CanFly(): bool

Inheritance | In C++

```
class Base{
    //--Base class members
};

class Derived: <access specifier> Base{
    //--Derived class members
};
```

- `<access specifier>` is used to specify the type of inheritance. In C++ we can have `public`, `private`, or `protected` inheritance. In this course we will only work with `public` inheritance.

```
class Derived: public Base{
    //--Derived class members
};
```

Inheritance | In C++ | The protected Access Specifier

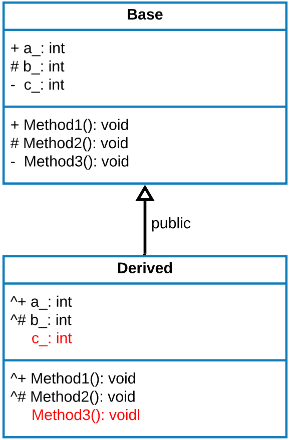
- The `protected` keyword specifies access to class members in the member-list up to the next access specifier (`public` or `private`) or the end of the class definition. Class members declared as protected can be used only by the following:
  - Methods of the class that originally declared these members.
  - Friends of the class that originally declared these members.
  - Classes derived with `public` or `protected` access from the class that originally declared these members.
- `protected` members are not as private as `private` members, which are accessible only to members of the class in which they are declared, but they are not as public as `public` members, which are accessible in any function (member and non-member functions).

Inheritance | In C++ | public Inheritance

```
class Base{
public://--represented with + in UML
    int a_;
    void Method1();
protected://--represented with # in UML
    int b_;
    void Method2();
private://--represented with - in UML
    int c_;
    void Method3();
};
```

- Note: private Base class members are inherited but not accessible.

```
#include "base.h"
class Derived: public Base{
};
```



Base

+ a_: int
# b_: int
- c_: int

+ Method1(): void
Method2(): void
- Method3(): void

public

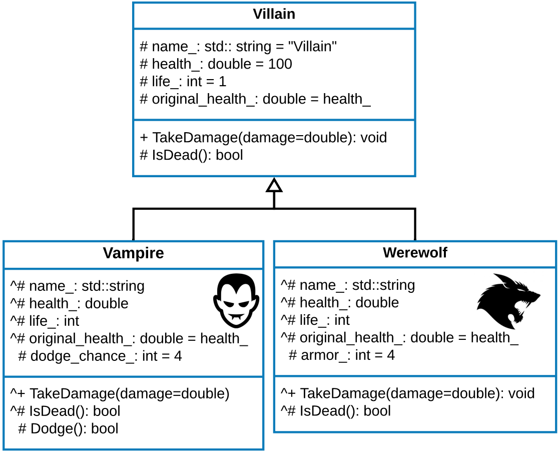
Derived

^+ a_: int
^# b_: int
c_: int

^+ Method1(): void
^# Method2(): void
Method3(): voidl

Inheritance | Domain

1. `Villain`: Base class.
   - Make all attributes of `Villain` `protected`.
2. `Vampire` and `Werewolf`: Derived classes.
   - They will inherit all `public` and `protected` members from `Villain`.
3. `Vampire` has a `protected` attribute and a `protected` method.
4. `Werewolf` has a `protected` attribute.
5. Test derived class objects in `main()` with the debugger.



Villain

name_: std:: string = "Villain"
health_: double = 100
life_: int = 1
original_health_: double = health_

+ TakeDamage(damage=double): void
IsDead(): bool

Vampire

^# name_: std::string
^# health_: double
^# life_: int
^# original_health_: double = health_
# dodge_chance_: int = 4

^+ TakeDamage(damage=double)
^# IsDead(): bool
Dodge(): bool

Werewolf

^# name_: std::string
^# health_: double
^# life_: int
^# original_health_: double = health_
# armor_: int = 4

^+ TakeDamage(damage=double): void
^# IsDead(): bool

Inheritance

TODO

- Using the class diagram from the previous slide:
  - Change `Villain` `private` attributes to `protected`.
  - Create classes `Werewolf` and `Vampire`.
  - They derive `public`ly from `Villain`.
  - Add new attributes in the derived classes.
  - Create of objects from the derived classes in `main()`.

Inheritance | Base and Derived Constructors

```
class Base{
public:
    Base(int b=0):b_{0}{}
protected:
    int b_;
};
```

```
int main(){
    Base base{1};
}
```

- Memory for `Base` object is set aside.
- The appropriate `Base` constructor is called.
- `b_` is initialized.
- The body of the constructor executes.
- Control is returned to the caller (`main()` function).

Inheritance | Base and Derived Constructors

```
class Base{
public:
    Base(int b=0):
        b_{b}{/*--body*/}
protected:
    int b_;
};//--class Base
```

```
class Derived: public Base{
public:
    Derived(double d=0.0):
        d_{d}{/*--body*/}
protected:
    double d_;
};//--class Derived
```

```
int main(){
    Derived derived(10.5);
    return 0;
}//--main
```

- Note: Constructors have to worry about their own members.
- Before the constructor of the Derived class can do anything substantial, the constructor of the Base class is called first.
- The Base class constructor sets up the Base portion of the object.
- Control is returned to the Derived constructor.
- The Derived constructor is allowed to finish up its job.
- Note: Since no argument is used with the constructor of Base, the default value (0) is used.
- Question: How to provide a value to initialize b_?

Inheritance | Base and Derived Constructors

- Approach #1: Uniform initialization of the Base class attribute through the Derived class constructor.

  ```
  Derived(double d=0.0, int b=0)
      : d_{d}, b_{b} {} //--error
  ```

  - C++ prevents classes from initializing inherited attributes in the initialization list of a constructor.
  - The value of an attribute can only be set in an initialization list of a constructor belonging to the same class as the attribute.
  - Only non-inherited attributes can be initialized in the initialization list.
- Approach #1: ✗

Inheritance | Base and Derived Constructors

- Approach #2: Set value in the body of the Derived constructor.

```
Derived(double d=0.0, int b=0)
    :d_{d}{
        b_ = b; //--no error
    }
```

  - This would not work if b_ were a const (because const values have to be initialized in the initialization list of the constructor).
  - Inefficient because b_ gets assigned a value twice: once in the initialization list of the Base constructor, and then again in the body of the Derived constructor.
- Approach #2: ✗

Inheritance | Base and Derived Constructors

- Approach #3: Explicitly call `Base` constructor from the `Derived` constructor.

```
Derived(double d=0.0, int b=0)
    : Base(b),//--Base constructor call
    d_{d} {}
```

- Note: The order `:Base(b),d_{d}{}` or `:d_{d},Base(b){}` does not matter as `Base(b)` will always be called first. The base portion is always constructed first.
- Approach #3: ✓

Inheritance | Base and Derived Constructors

- Approach #3: Explicitly call `Base` constructor from `Derived` constructor.
- Now we can safely use the following code.

```
int main(){
    //--initialize both Base::b_ and Derived::d_
    Derived derived(1.3,5);//--no error
    return 0;
}
```

  - `Base` constructor will initialize `b_` to `5`.
  - `Derived` constructor will initialize `d_` to `1.3`.

Inheritance | Base and Derived Constructors

- Approach #3: Explicitly call `Base` constructor from `Derived` constructor.

```
Derived derived(1.3,5);
```

1. Memory for `derived` is allocated (including the Base portion).
2. `Derived(double d=0.0,int b=0)` constructor is called with `d = 1.3` and `b = 5`.
3. The compiler checks if we have asked for a particular `Base` constructor.
   - 3.1 We have! So it calls `Base(5)`.
   - 3.2 `Base` constructor sets `b_` to `5`.
   - 3.3 `Base` constructor body executes, which does nothing.
   - 3.4 `Base` constructor returns to the `Derived` constructor.
4. `Derived` constructor sets `d_` to `1.3`.
5. `Derived` constructor body executes, which does nothing.
6. `Derived` constructor returns to `main()`.

Inheritance | Destructor

- When a derived object is destroyed, each destructor is called in the reverse order of construction.
- Destructors are called from the most derived to the more base classes.
- In the example below, the destructor from the `Derived` class is called first, followed by the destructor from the `Base` class.

```
int main(){
    Derived derived{4.5,3};
}
```

Inheritance

TODO

Rewrite the constructors for `Werewolf` and `Vampire` classes

Polymorphism

- The word polymorphism means having many forms.
- Polymorphism is one of the pillars of OOP.
- Two types of polymorphism:
  - Compile-time/early binding/static binding: default type
    - Function overloading.
    - Operator overloading: `a + b` (+ is a polymorphic operator).
  - Run-time/late binding/dynamic binding: not default type
    - Being able to assign different meanings to the same method at run-time.
    - Occurs with:
      1. Inherited classes.
      2. Pointer or reference to a Base class object.
      3. Virtual methods.

Polymorphism | Static Binding

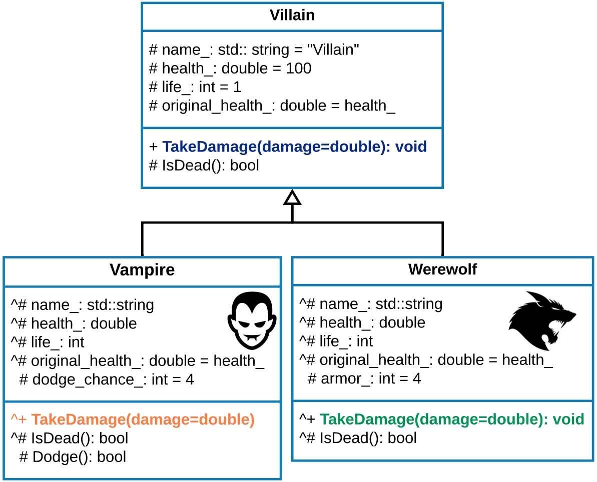
- By default, derived classes inherit all of the behaviors defined in a base class.
- When a method is called with a derived class object:
  - The compiler first looks to see if that method exists in the derived class.
  - If not, it begins walking up the inheritance chain and checking whether the method has been defined in any of the parent classes. It uses the first one it finds.
- Adding to existing functionality.
  - Sometimes we do not want to completely replace a base class method, but instead we want to add additional functionality to it.
  - It is possible to have our derived method call the base version of the method (in order to reuse code) and then add additional functionality to it.

Polymorphism | Static Binding

```
//--class Villain
void Villain::TakeDamage(double damage){
  // code for Villain
}
```

```
//--class Vampire
void Vampire::TakeDamage(double damage){
  // code for Vampire
}
```

```
//--class Werewolf
void Werewolf::TakeDamage(double damage){
  // code for Werewolf
}
```



Villain

name_: std:: string = "Villain"
health_: double = 100
life_: int = 1
original_health_: double = health_

+ TakeDamage(damage=double): void
IsDead(): bool

Vampire

^# name_: std::string
^# health_: double
^# life_: int
^# original_health_: double = health_
# dodge_chance_: int = 4

^+ TakeDamage(damage=double)
^# IsDead(): bool
Dodge(): bool

Werewolf

^# name_: std::string
^# health_: double
^# life_: int
^# original_health_: double = health_
# armor_: int = 4

^+ TakeDamage(damage=double): void
^# IsDead(): bool

Polymorphism

TODO

Redefine the method `TakeDamage()` in the `Werewolf` and `Vampire` classes.

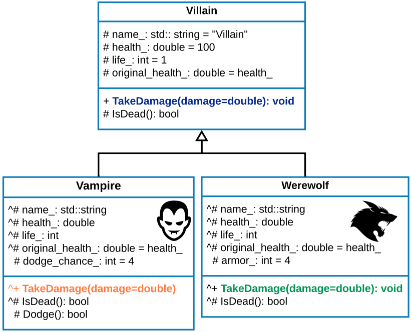
Polymorphism | Static Binding

- After redefining the method `TakeDamage()` for both derived classes we can do the following.

```
int main(){
    game::Villain bad{"Bad"};
    game::Vampire vlad{"Vlad"};
    game::Werewolf wolf{"Wolf"};
    bad.TakeDamage(3);//Villain::TakeDamage
    vlad.TakeDamage(3);//Vampire::TakeDamage
    wolf.TakeDamage(3);//Werewolf::TakeDamage
}
```

- What about the following? Let's take a deeper look in the next slide.

```
int main{
    //--A base class pointer pointing to a derived object
    std::shared_ptr<game::Villain> vlad = std::make_shared<game::Vampire>{3,"Vlad"};
    vlad->TakeDamage(3);//--Villain::TakeDamage()
    //--A base class reference to a derived object
    game::Werewolf wolf{3,"Wolf"};
    game::Villain &wolf_ref = wolf;
    wolf_ref.TakeDamage(3);//--Villain::TakeDamage()
}
```



Villain

name_: std:: string = "Villain"
health_: double = 100
life_: int = 1
original_health_: double = health_

+ TakeDamage(damage=double): void
IsDead(): bool

Vampire

^# name_: std::string
^# health_: double
^# life_: int
^# original_health_: double = health_
# dodge_chance_: int = 4

^+ TakeDamage(damage=double)
^# IsDead(): bool
Dodge(): bool

Werewolf

^# name_: std::string
^# health_: double
^# life_: int
^# original_health_: double = health_
# armor_: int = 4

^+ TakeDamage(damage=double): void
^# IsDead(): bool

Polymorphism | Static Binding

- First of all, are we allowed to write the code below?

```
//--A base class pointer to a derived object
std::shared_ptr<game::Villain> vlad = std::make_shared<game::Vampire>{3,"Vlad"};
//--A base class reference to a derived object
game::Werewolf wolf{4,"Wolf"};
game::Villain &wolf_ref = wolf;
```

- Yes, we can! The `Vampire` class is a `Villain` and the `Werewolf` class is a `Villain`.
- A derived object is a base class object, so it can be pointed to by a base class pointer (or a reference).
- Why would we want to write this kind of code? Why not the following?

```
//--A derived class pointer to a derived object
std::shared_ptr<game::Vampire> vampire =
    std::make_shared<game::Vampire>{3,"Vlad"};
//--A base class reference to a base object
game::Werewolf wolf{4,"Wolf"};
game::Werewolf &wolf_ref = wolf;
```

Polymorphism | Static Binding

- Consider a function which takes a `Villain` object as one of its parameters.
- The body of the function should use the correct method based on the type of the argument.

```
void Func(std::shared_ptr<game::Villain> villain, double damage){
    villain->TakeDamage(damage);
}

int main(){
 std::shared_ptr<game::Villain> bad = std::make_shared<game::Villain>{"Bad"};
 std::shared_ptr<game::Villain> vlad = std::make_shared<game::Vampire>{4,"Vlad"};
 std::shared_ptr<game::Villain> wolf = std::make_shared<game::Werewolf>{3,"Wolf"};
 Func(bad,5);//--will use Villain::TakeDamage
 Func(vlad,1);//--will use Vampire::TakeDamage
 Func(wolf,7);//--will use Werewolf::TakeDamage
}
```

Polymorphism | Static Binding

- We can also consider the following situation.

```
void Func(std::vector<std::shared_ptr<game::Villain>>& vect, double damage){
    for (auto villain: vect)
        villain->TakeDamage(damage);
}

int main(){
 std::shared_ptr<game::Villain> bad = std::make_shared<game::Villain>{"Bad"};
 std::shared_ptr<game::Villain> vlad = std::make_shared<game::Vampire>{3,"Vlad"};
 std::shared_ptr<game::Villain> wolf = std::make_shared<game::Werewolf>{3,"Wolf"};
 std::vector<std::shared_ptr<game::Villain>> vect;
 vect.push_back(bad);
 vect.push_back(vlad);
 vect.push_back(wolf);
 Func(vect,3);
}
```

Polymorphism | Dynamic Binding

- A redefined method is a method in a derived class which has a different definition than a non-virtual method in a base class.
  - Redefined methods are bound statically.
  - So far we have redefined `TakeDamage()` in the derived classes `Vampire` and `Werewolf`.
- An overridden method is a method in a derived class that has a different definition than a virtual method in a base class. The compiler chooses which method is desired based upon the type of the object being used to call the method.
  - Overridden methods are bound dynamically.
  - To write the `TakeDamage()` method as an overriden method instead of an redefined method we need to make `TakeDamage()` `virtual` in the base class `Villain`.

Polymorphism | Dynamic Binding | virtual Keyword

- Run-time/late binding/dynamic binding: not default type
  - Being able to assign different meanings to the same method at run-time.
  - Occurs with:
    1. Inherited classes. ✓
    2. Pointer or reference to a base class.✓
    3. Virtual methods.✓
- When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual method for that object and execute the derived class version of the method.
- A virtual method is declared within a base class and is overriden by a derived class.

Polymorphism | Dynamic Binding | virtual Keyword

- How to declare a virtual method?
  - Declare the method you want to override as virtual in the base class using the `virtual` keyword.
  - A virtual method is virtual all the way down the hierarchy from this point.

```
class Villain{
public:
  virtual void TakeDamage(double damage);//-- virtual method
                         //-- can be overridden in the derived classes
                         //-- will be bound dynamically at run-time
};
```

Polymorphism | Dynamic Binding | virtual Keyword

- *Best practice*: The `virtual` keyword is optional for the overridden methods but it is best practice to use it.

```
class Villain {
public:
  virtual void TakeDamage(double);//--virtual keyword mandatory
};
```

```
class Vampire: public Villain{
public:
  virtual void TakeDamage(double);//--virtual keyword optional
};
```

```
class Werewolf: public Villain{
public:
  virtual void TakeDamage(double);//--virtual keyword optional
};
```

Polymorphism | Dynamic Binding | `virtual` Keyword

- If a class has at least one `virtual` method then a `public virtual` destructor must be provided.
- Behavior is undefined in the C++ standard if you do not provide a `public virtual` destructor.
- If a base class destructor is `virtual` then all derived class destructors are also `virtual`.
- *Best practice*: Explicitly make the destructor in each derived class `virtual`.

```cpp
class Villain{
public:
  virtual void TakeDamage(double);
  virtual ~Villain(){}
};
```

Polymorphism

TODO

Make the necessary methods `virtual`.

Make the destructors `virtual`.

Test with and without `virtual` destructors.

Polymorphism | override Specifier

- To override base class `virtual` methods, the methods in the derived classes must have the same header (return type + method name + parameter list + keywords) as the base class methods.
- If the header is different in the derived classes, then we have a redefinition and not override.

```
class Villain{
public:
  virtual void TakeDamage(double);
};
class Vampire{
public:
  virtual void TakeDamage(int);//--redefinition
};
//--main()
game::Vampire vlad{4,"Vlad"};
game::Villain &vlad_ref = vlad;
vlad.TakeDamage(4.5);//Villain::TakeDamage
```

Polymorphism | override Specifier

- C++11 provides an `override` specifier to have the compiler ensure the overriding.
- The `override` keyword serves two purposes:
  1. It shows the reader of the code that "this is a virtual method, that is overriding a virtual method of the base Class."
  2. The compiler also knows that it's an override, so it can "check" that you are not altering/adding new methods that you think are overrides (compile-time error is generated otherwise).

Polymorphism | override Specifier

```
class Villain{
public:
  virtual void TakeDamage(double);
};
```

```
class Vampire{
public:
  virtual void TakeDamage(int) override;//--compiler error
};
```

- With the `override` specifier you are telling the compiler that the method `TakeDamage` in `Vampire` overrides the method `TakeDamage` in `Villain`.
- In this case the compiler will throw an error. `TakeDamage` in `Vampire` redefines and does not overrides.

Polymorphism

TODO

Override derived class methods with the `override` specifier.

Polymorphism | final Specifier

- C++11 provides the `final` specifier to specify an intention.
  - When used at the class level, it prevents a class from being derived from.

```
class Villain final{//--no possible inherited classes from Villain
};
```

```
class Vampire: public Villain{//--compiler error
};
```

  - When used at the method level, it prevents the method from being overridden in the derived classes.

```
class Villain{
public:
    virtual void TakeDamage(double) final;//--no further overriding
};
```

```
class Vampire: public public:
    virtual void TakeDamage(double) override;//--compiler error
};
```

Polymorphism | Abstract and Concrete Classes

- Abstract class:
  - Too generic to instantiate objects from.
  - Used only as base classes in inheritance hierarchies.
  - Often referred to abstract base class.
  - Contains at least one pure virtual method.
  - Examples: Shape, Person,. . . , Villain.
- Concrete class:
  - Used to instantiate objects from.
  - All their methods are defined.

Polymorphism | Abstract Classes

- Pure virtual methods:
  - Used to make a class abstract.
  - Abstract class has at least one pure virtual method.
  - Specified with =0 in their declaration.

```cpp
class A {//--abstract class
public:
    virtual void FirstMethod()=0; //--pure virtual method
    virtual void SecondMethod();
};
```

  - Typically do not provide any method implementation.
  - Used when it does not make sense for a base class to have an implementation.

Polymorphism | Abstract Classes

```cpp
class Villain {//--abstract class
public:
    virtual void TakeDamage(double)=0;//--pure virtual method
    virtual bool IsDead(){
        //--code
    }
    virtual ~Villain(){}
};
```

```cpp
int main(){
    game::Villain bad{"Bad"};//--error game::Villain is an abstract class
}
```

Polymorphism | Concrete Classes

- Classes that derive from abstract classes must override all pure virtual methods.
- If not all pure virtual methods are overridden, then the derived Classes are abstract Classes.

```
class Villain {//--abstract class
public:
    virtual void TakeDamage(double)=0;//--pure virtual method
    virtual bool IsDead(){
        //--code
    }
    virtual ~Villain(){}
};
```

```
class Vampire: public Villain {//--abstract class
public:
    virtual ~Villain(){}
};
```

- Vampire does not override all pure virtual methods of `Villain`.
- Vampire is considered an abstract class (can't instantiate an object from Vampire).

Polymorphism | Concrete Classes

- Classes that derive from abstract classes must override all pure virtual methods.
- If not all pure virtual methods are overridden, then the derived Classes are abstract Classes.

```cpp
class Villain {//--abstract class
public:
    virtual void TakeDamage(double)=0;//--pure virtual method
    virtual bool IsDead(){
        //--code
    }
    virtual ~Villain(){}
};
```

```cpp
class Vampire: public Villain {//--concrete class
public:
    virtual void TakeDamage(double) override;
    virtual ~Villain(){}
};
```

- Vampire overrides all pure virtual methods of `Villain`.
- Vampire is considered a concrete class (can instantiate an object from Vampire).

Polymorphism

TODO

Make `Villain` an abstract class.
Make `Vampire` and `Werewolf` concrete classes.

Next Class | 11/05

- Lecture10: Final Project discussion.
- Quiz on OOP – Part I & III only.
- Submit the assignment before the due date.
- Stay safe!