CS-202

C++ Classes – Inheritance (Pt.2)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	PASS Session		PASS Session

Your 5th Project will be announced today Thursday 2/28.

4th Project Deadline was this Wednesday 3/6.

- NO Project accepted past the 24-hrs delayed extension (@ 20% grade penalty).
- Send what you have in time!
- Always check out **WebCampus** CS-202 Samples for some **help** !

Today's Topics

Method Overriding

- Overriding *vs* Overloading

Inheritance Rules

- Base–Derived Constructors
- Derived–Base Destructors
- Assignment Operator

Polymorphism Prelude

- Base class pointers to address Derived class Objects

Advanced Examples

- Derived–to–Base forwarding via `static_cast`-ing
- `protected` Interface: Constructors, Destructor, Assignment Operator

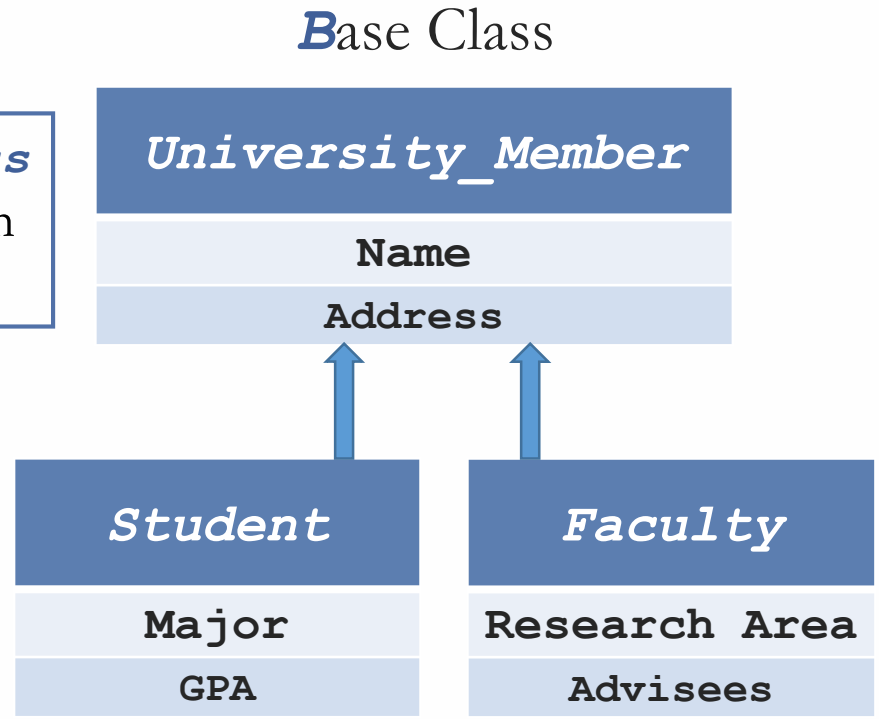
Inheritance Relationship

Inheritance Syntax:

```
class BaseClass {
    public:
        //operations
    private:
        //data
};
```

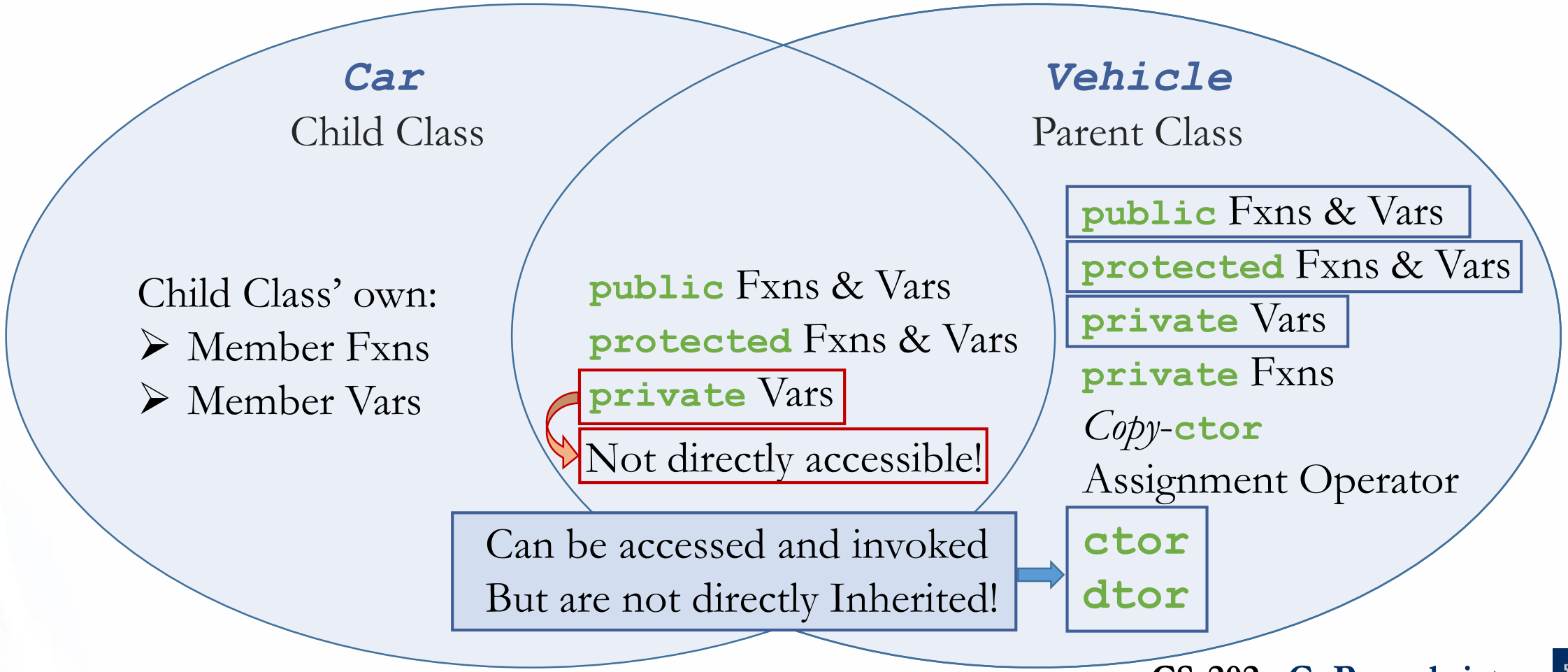
```
class DerivedClass : public BaseClass {
    public:
        //operations
    private:
        //data
};
```

Indicates that this *DerivedClass* Inherits data and operations from this *BaseClass*



Inheritance

Inherited Member(s)



Handling Access

Derived/Child Class has access to Base/Parent Class's:

- `protected` Member Variables/Functions.
- `public` Member Variables/Functions (as everything else also does).

No access to Base/Parent Class's `private` Member Variables/Functions:

- Not even through Derived/Child Class' own Member Function.

Remember:

`private` Member Variables are only directly accessible ("by name") in Member Functions of their own Class (they one they are defined in).

Method Overriding

Overriding

Remember: Interface of a Derived/Child Class:

- *Extends*: Contains declarations for its own new Member Functions.
- *Overrides*: Contains declarations for Inherited Member Functions to be changed.

Implementation of a Derived/Child Class will:

- Define new Member Functions.
- Redefine Inherited Functions *when you Declare them*!

```
class Vehicle {
  public:
    int getMileage(){ return m_mileage; }
  private:
    int m_mileage;
} ;
```

```
class Car : public Vehicle {
  public:
    int getMileage();
} ;
```

Now that you re-Declared it, you have to Define it!

Method Overriding

Overriding *vs* **Overloading**

Overriding in a Derived/Child class means "*Redefining what it does*":

- The same parameters list.
- Essentially "crossing-out & re-writing" what the one-and-same function does!
- Overridden functions share the same signature (because they are one function)!

Overloading a Function means "*Reusing its name*":

- Using a different parameter(s) list.
- Essentially defining a "new version of" a function (that takes different parameters).
- Overloaded functions must have different signatures!

Method Overriding

Overriding *vs* **Overloading**

Overriding in a Derived/Child class means "*Redefining what it does*":

- Overridden functions share the same signature (because they are one function)!

Overloading a Function means "*Reusing its name*":

- Overloaded functions must have different signatures!

Function "*Signature*" (or the information that participates in *Overload Resolution*):

- The *unqualified* name of the function.
- The specific sequence of types (names are irrelevant) in parameters list (including order, number, types).
- Signature does NOT include: `return` type (later however we will encounter an issue here), `const` keyword for non-Reference type parameters (e.g. `const int` *vs* `int`)
- Signature DOES include: Class method `cv`-qualifiers (e.g. `const` keyword at the end)

Method Overriding

Overriding *vs* Overloading

Method Overriding (uses exact *same signature*):

- Derived Class Method can modify, add to, or replace Base Class methods.
- Derived Method will be called for Derived Objects.
- Base Method will be called for Base Objects.

```
class Animal {
  public:
    void eat(){
      cout << "I eat stuff" << endl;
    }
};
class Lion : public Animal {
  void eat(){
    cout << "I eat meat" << endl;
  }
};
```

```
int main(){
  Animal animal;
  animal.eat();  // I eat stuff
  Lion lion;
  lion.eat();    // I eat meat
}
```

Method Overriding

Overriding *vs* Overloading

Method Overloading (uses exact *different signature*):

- A different function (which however carries the same name!)
- Derived/Child Class has access to both functions.

```
class Animal {
  public:
    void eat(){
      cout << "I eat stuff" << endl;
    }
};
class Lion : public Animal {
  public:
    void eat(const char * food)
      cout << "I ate a " << food << endl;
    }
};
```

```
int main(){
  Lion lion;
  lion.eat();          // I eat stuff
  lion.eat("Steak"); // I ate a Steak
}
```

Inheritance Rules - Further

All "normal" functions in Base/Parent class are Inherited in Derived/Child Class.
Inheritance exceptions to the rules so far are:

- Constructor(s) - `ctor`
- Destructor(s) - `dtor`
- *Copy*-`ctor`
  If none is specified for Derived Class, compiler will still generate an "automatically synthesized" one.
- Assignment Operator (`=`)
  If none is specified for Derived Class, compiler will still generate an "automatically synthesized" one (which will also invoke the Base Class assignment `operator=`) .
  If one is implemented for the Derived Class that one will be called alone .

Inheritance Rules - Further

All "normal" functions in Base/Parent class are Inherited in Derived/Child Class.
Inheritance exceptions to the rules so far are:

- Constructor(s) - `ctor`
- Destructor(s) - `dtor`
- *Copy*-`ctor`
  If none is specified for Derived Class, compiler will still generate an "automatically synthesized" one.
- Assignment Operator (`=`)
  If none is specified for Derived Class, compiler will still generate an "automatically synthesized" one (which will also invoke the Base Class assignment `operator=`) .
  If one is implemented for the Derived Class that one will be called alone .

Inheritance Rules

Assignment `operator=` in Derived Class(es)

Assignment Operator (=)

- If none is specified for Derived Class, compiler will still generate an "automatically synthesized" one (which will also invoke the Base Class assignment `operator=`) .
- If one is implemented for the Derived Class that one will be called alone .

We can also directly call the Base Class Assignment from the Derived Class one !

```
class Animal{
  Animal & operator=(const Animal & rhs_animal ){
    cout << "Animal assignment" << endl;
  }
};
class Lion : public Animal{
  Lion & operator=(const Lion & rhs_lion ){
    Animal::operator=( rhs_lion );
    cout << "Lion assignment" << endl;
  }
};
```

Inheritance Rules

Constructor & Destructor in Derived Class(es)

The Base/Parent Class `ctor`s are *not* Inherited in Derived(Child) Classes.

- They can however be invoked within Derived(Child) Class' `ctor`.

Nothing more is required:

- A Base(Parent) Class `ctor` must instantiate all Base Class Member Variables.
- These are Inherited by Derived(Child) Class!

The "First thing" that *any* Derived(Child) Class `ctor` does is to try and invoke the Base(Parent) Class *Default* `ctor` !

- Unless we otherwise explicitly specify to call another Base(Parent) Class `ctor` .
  (We will see how in a little while…)

Inheritance Rules

Constructor & Destructor in Derived Class(es)

Constructor(s)

- Base/Parent Class `ctor` is called *before* Derived/Child Class `ctor`.

```
DerivedClass dClass;
```

```
BaseClass();
DerivedClass();
```

DerivedClass	BaseClass
DerivedClass();	BaseClass();
…	…

Destructor

- Derived/Child Class `dtor` is called *before* Base/Parent Class `dtor`.

```
dClass.~DerivedClass();
```

```
~DerivedClass();
~BaseClass();
```

DerivedClass	BaseClass
~DerivedClass();	~BaseClass();
…	…

Inheritance Rules

Constructor & Destructor in Derived Class(es)

Sequence of Base-Derived `ctor` & Derived-Base `dtor` calls

Example:

```
class Animal {
  public:
    Animal(){cout << "Base constructor"<<endl;}
    ~Animal(){cout << "Base destructor"<<endl;}
};
class Lion : public Animal {
 public:
    Lion(){cout << "Derived constructor"<<endl;}
    ~Lion(){cout << "Derived destructor"<<endl;}
};
```

```
int main(){
  Lion lion;
  return 0;
}
```

Output:

```
Base constructor
Derived constructor
Derived destructor
Base destructor
```

Inheritance Rules

Parametrized Constructor in Derived Class(es)

Calling the Base `ctor` from a Derived `ctor` explicitly

Example:

```
class Animal {
  public:
    Animal(const char * name) {
       strcpy(m_name, name);
       cout<<m_name<<endl;
     }
  protected:
     char m_name[MAXNAME];
};
class Lion : public Animal {
  public:
    Lion(const char * name) : Animal(name){
      /* Rest of Lion Parametrized ctor statements*/
     }
};
```

```
int main(){
   Lion lion("King");
   return 0;
}
```

Output:

```
King
```

Note: Initializer-list is the only way that allows calling the Base `ctor` from a Derived `ctor`.

Inheritance Rules

Parametrized Constructor in Derived Class(es)

Calling the Base `ctor` from a Derived `ctor` explicitly

Example:

```
class Animal {
  public:
    Animal(const char * name) {
       strcpy(m_name, name); cout<<m_name<<endl;
    }
  protected:
    char m_name[MAXNAME];
};
class Lion : public Animal {
  public:
    Lion(const char * name) : Animal(name){
       strcat(m_name," Lion"); cout<<m_name<<endl;
    }
};
```

```
int main(){
  Lion lion("King");
  return 0;
}
```

Output:

```
King
King Lion
```

Note: Calls Parametrized Base `ctor` by passing down argument from the Derived `ctor`.

Polymorphism (prelude)

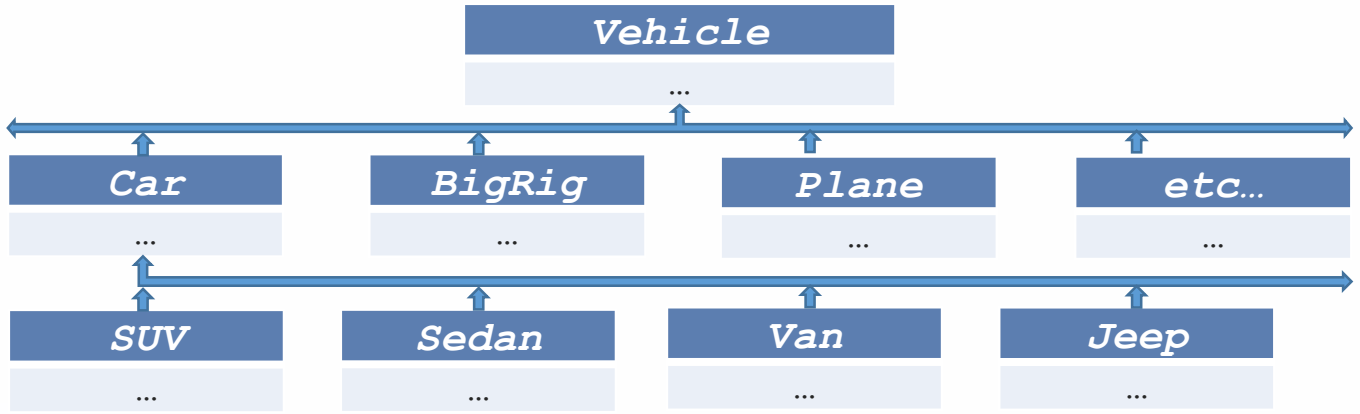
Inheritance & Polymorphism

Polymorphism means "the ability to take many forms".

- Allowing a single (*Remember*: Overriding *vs* Overloading) behavior to take on many type-dependent forms.
- Hence, grants the ability to manipulate Objects in a type-independent way.

Pointers of Base Class-type:

- They are used to address "*Common Ancestor*"-type Objects from a Class Hierarchy.

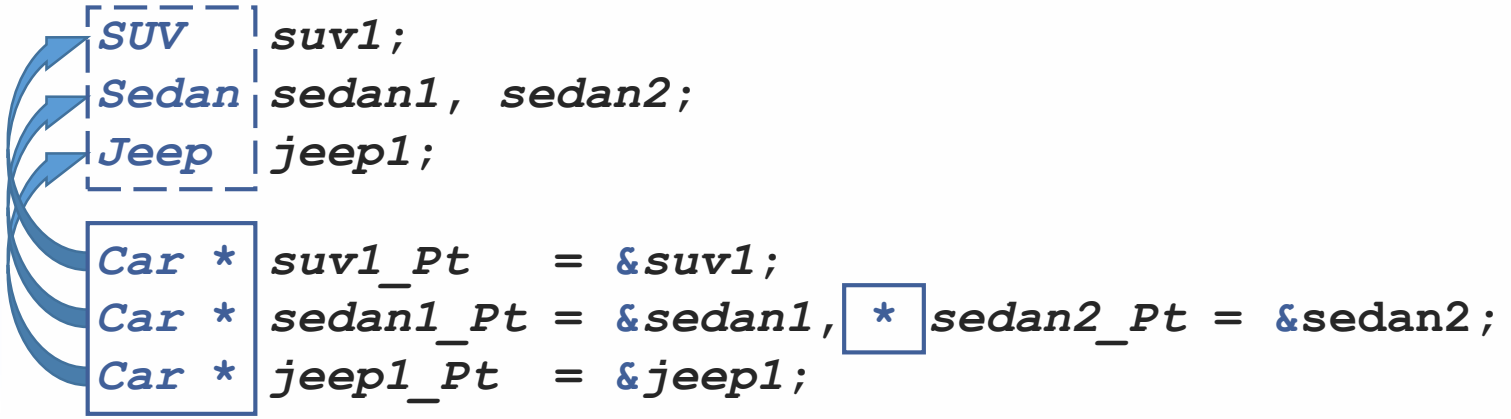


Polymorphism (prelude)

Inheritance & Polymorphism

Base Class-type Pointers :

- A Pointer of a Base Class type can point to an Object of a Derived Class type.



```
SUV   suv1;
Sedan sedan1, sedan2;
Jeep  jeep1;

Car * suv1_Pt   = &suv1;
Car * sedan1_Pt = &sedan1, * sedan2_Pt = &sedan2;
Car * jeep1_Pt  = &jeep1;
```

This is valid: A Derived Class (***SUV***, ***Sedan***, ***Van***, ***Jeep***) "is a type of" Base Class (***Car***).

- *Note*: A 1-way relationship:

  A Derived Class Pointer cannot point to a Base Class Object.

Advanced Examples

Derived to Base Class forwarding

Parent Class "hides" data even from Children (but has its own fully defined *Interface* !) :

```
class Animal {
  friend ostream & operator<<(ostream & os, const Animal & animal){
    os << "I have "  << animal.m_legs << " legs" << endl;
    return os;
  }
  private:
    int m_legs;
};
class Lion : public Animal {
  friend ostream & operator<<(ostream & os, const Lion & lion){
    os << "I am a "  << lion.m_color << " lion and\n"
       << "I have "  << lion.m_legs << " legs" << endl;
    return os;
  }
  private:
    char m_color[256];
};
```

Advanced Examples

Derived to Base Class forwarding

Parent Class "hides" data even from Children (but has its own fully defined *Interface* !) :

```
class Animal {
  friend ostream & operator<<(ostream & os, const Animal & animal){
    os << "I have "  << animal.m_legs << " legs" << endl;
    return os;
  }
  private:
    int m_legs;
};
class Lion : public Animal {
  friend ostream & operator<<(ostream & os, const Lion & lion){
    os << "I am a "  << lion.m_color << " lion and\n"
       << "I have "  << lion.m_legs << " legs" << endl;
    return os;
  }
  private:
    char m_color[256];
};
```

Not possible !
The function is a **friend** of ***Lion***,
not a **friend** of ***Animal*** !

Advanced Examples

Derived to Base Class forwarding

Parent Class "hides" data even from Children (but has its own fully defined *Interface* !) :

```
class Animal {
  friend ostream & operator<<(ostream & os, const Animal & animal){
    os << "I have "  << animal.m_legs << " legs" << endl;
    return os;
  }
  private:
    int m_legs;
};
class Lion : public Animal {
  friend ostream & operator<<(ostream & os, const Lion & lion){
    os << "I am a "  << lion.m_color << " lion and\n"
       << static_cast< const Animal & >( lion );
    return os;
  }
  private:
    char m_color[256];
};
```

Casting to a *Reference* of Base class type invokes the corresponding function overload for ***Animal*** !

CS-202

Time for Questions !