UCLA Samueli Computer Science

CS32: Introduction to Computer Science II

Discussion Week 5

Yichao (Joey)

May 1, 2020

Credit: Carey Nachenberg, Junheng Hao

Announcements

UCLA Samueli Computer Science

- Homework3 is due 11:00PM Thursday, May 7.

- LA evaluation:

  https://docs.google.com/forms/d/e/1FAIpQLSddpMmDlJufdWSNioFtQpxxiqANraBcF15P2vA6pdvH_R_-Hw/viewform

Outline Today

UCLA Samueli Computer Science

- Inheritance and Polymorphism

Inheritance & Polymorphism

Motivation & Review

UCLA Samueli Computer Science

- **Inheritance**
  - Motivation & Definition: **Deriving a class from another**
  - Reuse, extension, specification (override)
  - Construction & Destruction
  - Override a member function
- **Polymorphism**
  - Virtual functions
  - Examples of polymorphism
  - Abstract base class

Inheritance

Motivation & Review

UCLA Samueli Computer Science

- **The basis of all *Object Oriented Programming***. And you'll almost certainly get grilled on it! --- From: Nachenberg, Slides L6P3
- The process of deriving a new class using another class as base.
- Difference of *"is a"(class hierarchy)* and *"has a"(has member/properties)*

```
class Person {
public:
  string getName(void);
  void setName(string & n);
  int getAge(void);
  void setAge(int age);
private:
  string m_sName;
  int m_nAge;
};
```

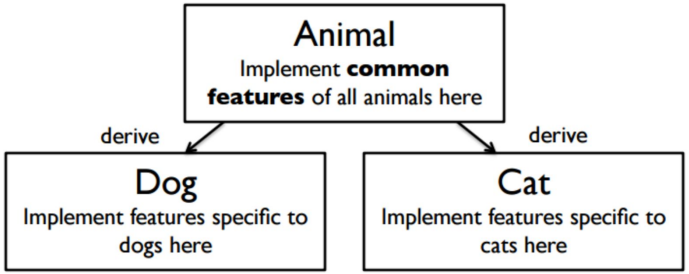
```
class Student {
public:
  string getName(void);
  void setName(string & n);
  int getAge(void);
  void setAge(int age);
  int getStudentID();
  void setStudentID();
  float getGPA();
private:
  string m_sName;
  int m_nAge;
  int m_nStudentID;
  float m_GPA;
};
```

```
class Professor {
public:
  string getName(void);
  void setName(string & n);
  int getAge(void);
  void setAge(int age);
  int getProfID();
  void setProfID();
  bool getIsTenured();
private:
  string m_sName;
  int m_nAge;
  int m_nStudentID;
  bool isTenured;
};
```

Inheritance

Example: Reuse and Extension

UCLA Samueli Computer Science

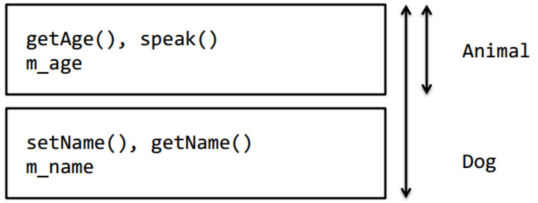



```
class Animal
{
  public:
    Animal();
    ~Animal();
    int getAge() const;
    void speak() const;
  private:
    int m_age;
};
```

base class

```
class Dog : public Animal
{
  public:
    Dog();
    ~Dog();
    string getName() const;
    void setName(string name);
  private:
    string m_name;
};
```

derived class




```
Dog d1;
d1.setName("puppy");
d1.getAge();
d1.speak();
```

```
Animal a1;
a1.speak();
a1.setName("abc");
```

Inheritance

Summary of Reuse and Extension

UCLA Samueli Computer Science

- Reuse
  - Every public method in the base class is automatically reused/exposed in the derived class (just as if it were defined).
  - Only public members in the base class are exposed/reused in the derived class(es)! Private members in the base class are hidden from the derived class(es)!
  - **Special case for protected members.**
- Extension
  - All public extensions may be used normally by the rest of your program.
  - Extended methods or data are unknown to your base class.

Inheritance

Summary of Reuse and Extension

UCLA Samueli Computer Science

When deriving a class from a public base class,

Public members of the base class become public members of the derived class

Protected members of the base class become protected members of the derived class.

A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

Inheritance

Specialization/Overriding member functions

UCLA Samueli Computer Science

- Overriding: same function name, return type and parameter list, defined again in derived classes and different from the base class.
- You can still call the member function of base classes, but it seems very rare.

```
Dog d1;
d1.Animal::speak();
```

```
void Animal::speak() const
{
  cout << "..." << endl;
}
```

```
class Dog : public Animal
{
  public:
    Dog();
    ~Dog();
    string getName() const;
    void setName(string name);
    void speak() const;
  private:
    string m_name;
};

void Dog::speak() const
{
  cout << "Woof!" << endl;
}
```

Inheritance

Construction

UCLA Samueli Computer Science

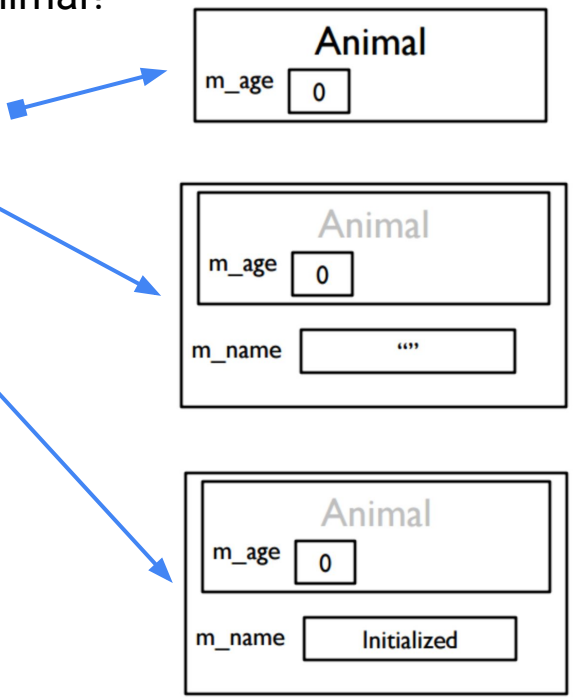
- How to construct a Dog, which is a derived class from Animal?
- Steps:
  - The base part of the class (Animal) is constructed.
  - The member variables of Dog are constructed.
  - The body of constructor (Dog) is executed.

```
class Animal
{
  public:
    Animal();
    ~Animal();
    int getAge() const;
    void speak() const;
  private:
    int m_age;
};
```

base class

```
class Dog : public Animal
{
  public:
    Dog();
    ~Dog();
    string getName() const;
    void setName(string name);
  private:
    string m_name;
};
```

derived class



Inheritance

Overload Constructor

UCLA Samueli Computer Science

- How to overload Dog's constructor to create `Dog::Dog(string initName, int initAge)` ?

// Wrong:

```
Dog::Dog(string initName, int initAge)
:m_age(initAge), m_name(initname)
{}
```

// Correct:

```
Dog::Dog(string initName, int initAge)
:Animal(initAge), m_name(initname)
{}

class Animal{
  public:
    Animal(init initAge);
    ...
}
```

Inheritance

Order of Construction and destruction

UCLA Samueli Computer Science

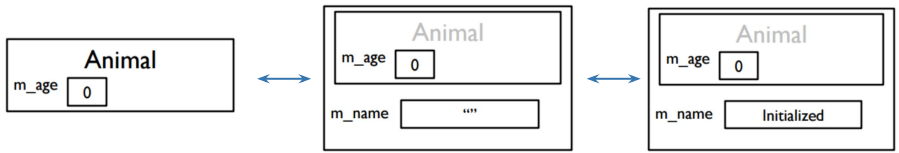
The order of destruction of a derived class: **Just reverse the order of construction.**

Order of construction:

1. Construct the base part, consulting the member initialization list (If not mentioned there, use base class's default constructor)
2. Construct the data members, consulting the member initialization list.(If not mentioned there, use member's default constructor if it's of a class type, else leave uninitialized.)
3. Execute the body of the constructor.

Order of destruction:

1. Execute the body of the destructor.
2. Destroy the data members (doing nothing for members of builtin types).
3. Destroy the base part.



```
class A
{
public:
    A(){cout<<"A()"<<endl;}
    A(int x){cout<<"A("<<x<<")"<<endl;id=x;}
    ~A(){cout<<"~A("<<id<<")"<<endl;}
private:
    int id;
};
class B
{
public:
    B():aa(10){cout<<"B()"<<endl;}
    ~B(){cout<<"~B()"<<endl;}
private:
    A aa;
};

int main()
{
    B b;
    return 0;
}
```

```
A(10)
B()
~B()
~A(10)
```

Note: There is a difference between class composition and class inheritance.

```
class A
{
public:
    A(){cout<<"A()"<<endl;}
    A(int x){cout<<"A("<<x<<")"<<endl;id=x;}
    ~A(){cout<<"~A("<<id<<")"<<endl;}
private:
    int id;
};
class B: public A
{
public:
    B():aa(10){cout<<"B()"<<endl;}
    ~B(){cout<<"~B()"<<endl;}
private:
    A aa;
};

int main()
{
    B b;
    return 0;
}
```

```
A()
A(10)
B()
~B()
~A(10)
~A(4196528)
```

Note: There is a difference between class composition and class inheritance.

Construction

One more test!

What is the output of

```
int main(){
    C c;
}
```

```
class A
{
  public:
    A() { cout << "A()" << endl; }
    A(int x) { cout << "A(" << x << ")" << endl; }
    ~A() { cout << "~A()" << endl; }
};

class B
{
  public:
    B() { cout << "B()" << endl; }
    B(int x) : m_a(x) { cout << "B(" << x << ")" << endl; }
    ~B() { cout << "~B()" << endl; }
  private:
    A m_a;
};

class C : public A
{
  public:
    C() : A(10), m_b2(5) { cout << "C()" << endl; }
    ~C() { cout << "~C()" << endl; }
  private:
    B m_b1;
    B m_b2;
};
```

```
A(10)
A()
B()
A(5)
B(5)
C()
~C()
~B()
~A()
~B()
~A()
~A()
```

*Philosophy/Inheritance

Another example

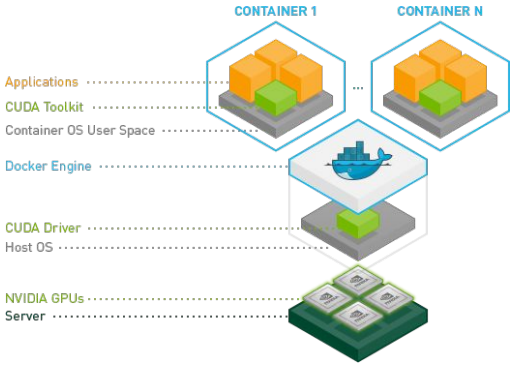
UCLA Samueli Computer Science

- There are many examples and applications of “inheritance”.
- One example: Commonly-used Docker Images

HIGH PERFORMANCE COMPUTING | DEEP LEARNING | MACHINE LEARNING | INFERENCE | VISUALIZATION | INFRASTRUCTURE

Publisher: All | Search containers

NAME	REPOSITORY	PUBLISHER	LATEST TAG	MODIFIED	SIZE	BUILT BY	PULL LATEST
Caffe2	nvidia/caffe2	Facebook	18.08-py3	August 27, 2018	1.3 GB	NVIDIA	
Chainer	partners/chainer	Preferred Ne...	4.0.0b1	December 12, 2017	963.75 MB	Preferred Ne...	
CUDA	nvidia/cuda	NVIDIA	9.0-devel-ub...	December 14, 2018	1.04 GB	NVIDIA	
CUDA Sample	nvidia/k8s/cuda-sample	NVIDIA	nbody	June 18, 2018	95.75 MB	NVIDIA	
Deep Cognition Studio	partners/deep-learning-studio	Deep Cogniti...	cuda9-2.5.0	October 18, 2018	2.03 GB	Deep Cogniti...	
DIGITS	nvidia/digits	NVIDIA	19.01-caffe	January 24, 2019	1.47 GB	NVIDIA	
H2O Driverless AI	partners/h2oai-driverless	H2O.ai	latest	March 9, 2018	2 GB	H2O.ai	
Kinetica	partners/kinetica	Kinetica	cuda9-6.1.0.9	March 22, 2018	5.62 GB	Kinetica	
MATLAB	partners/matlab	Mathworks	r2018b	February 6, 2019	8.1 GB	Mathworks	
Microsoft Cognitive Toolkit	nvidia/cntk	Microsoft Res...	18.08-py3	August 27, 2018	2.4 GB	NVIDIA	
MXNet	nvidia/mxnet	Apache Softw...	19.01-py3	January 24, 2019	1.46 GB	NVIDIA	
NVCaffe	nvidia/caffe	NVIDIA	19.01-py2	January 24, 2019	1.39 GB	NVIDIA	
OmniSci (MapD)	partners/mapd	OmniSci	3.2.2	December 1, 2017	662.43 MB	OmniSci	
PaddlePaddle	partners/paddlepaddle	Baidu	0.11-alpha	December 3, 2017	1.28 GB	Baidu	
PyTorch	nvidia/pytorch	Facebook	19.01-py3	January 24, 2019	2.7 GB	NVIDIA	
RAPIDS	nvidia/rapidsai/rapidsai	Open Source	cuda9.2-runti...	January 31, 2019	2.67 GB	NVIDIA	
TensorFlow	nvidia/tensorflow	Google Brain ...	19.01-py3	January 24, 2019	2.34 GB	NVIDIA	
TensorRT	nvidia/tensorrt	NVIDIA	19.01-py3	January 24, 2019	1.5 GB	NVIDIA	
TensorRT Inference Server	nvidia/tensorrtserver	NVIDIA	19.01-py3	January 24, 2019	1.49 GB	NVIDIA	
Theano	nvidia/theano	University of ...	18.08	August 27, 2018	1.49 GB	NVIDIA	



Inheritance does not exactly just means base/derived class in C++ programming. It is everywhere.

Polymorphism

Motivation & Definition

UCLA Samueli Computer Science

- Polymorphism is how you make Inheritance truly useful.
- Once I define a function that accepts `Animal a` (reference or pointer to **a**),
  not only can I pass `Animal` variables to that class,
  but I can also pass any variable that was derived from `Animal` (such as `Dogs`)!

Polymorphism

Virtual Functions and Binding

UCLA Samueli Computer Science

- **A virtual function** is a member function of a class, whose functionality can be over-ridden in its derived classes. It is one that is declared as virtual in the base class using the virtual keyword.
- **Binding** refers to the act of associating an object or a class with its member.

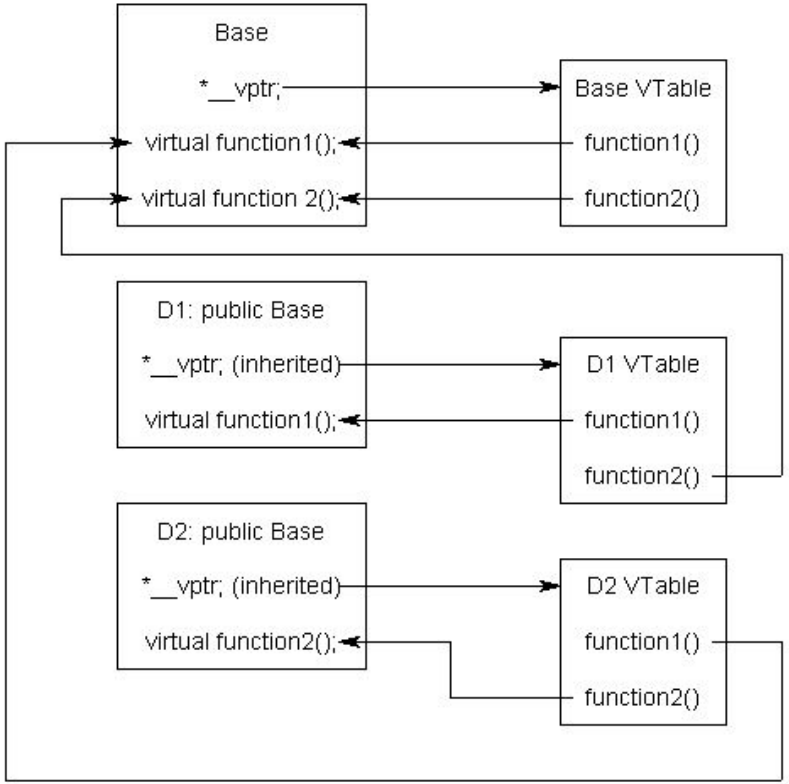
  E.g. If we can call a method fn() on an object o of a class c, we say that the object o is binded with the method fn().
  - This happens at compile time and is known as **static** or **compile - time binding**.
  - The calls to the virtual member functions are resolved during run-time. This mechanism is known as **dynamic binding**.
- The most prominent reason why a virtual function will be used is to have a **different functionality** in the derived class.

Polymorphism

Virtual Functions and Binding

UCLA Samueli Computer Science

- Whenever a program has a virtual function declared, a **v-table** is constructed for the class. The v-table consists of **addresses to the virtual functions for classes that contain one or more virtual functions**.
- This vtable pointer or _vptr, is a hidden pointer added by the compiler to the base class and this pointer is pointing to the virtual table of that particular class. This _vptr is inherited to all the derived classes.



Polymorphism

Virtual Functions: Examples

UCLA Samueli Computer Science

```
class Shape {
  public:
    virtual double getArea()
    { return (0); }
    ...
  private:
    ...
};
```

```
class Square: public Shape {
  public:
    Square(int side){ m_side=side; }
    virtual double getArea()
    { return (m_side*m_side); }
    ...
  private:
    int m_side;
};
```

```
class Circle: public Shape {
  public:
    Circle(int rad){ m_rad=rad; }
    virtual double getArea()
    { return (3.14*m_rad*m_rad);}
    ...
  private:
    int m_rad;
};
```

```
int main() {
  Shape *s = new Square();
  Shape *c = new Circle();
  s->getArea();
  c->getArea();
}
```

When you use the `virtual` keyword, C++ **figures out what class is being referred** and **calls the right function.**

How?

Polymorphism

Virtual Functions: Examples

UCLA Samueli Computer Science

```
class Shape {
  public:
    virtual double getArea()
    { return (0); }
    ...
  private:
    ...
};
```

```
class Square: public Shape {
  public:
    Square(int side){ m_side=side; }
    virtual double getArea()
    { return (m_side*m_side); }
    ...
  private:
    int m_side;
};
```

```
class Circle: public Shape {
  public:
    Circle(int rad){ m_rad=rad; }
    virtual double getArea()
    { return (3.14*m_rad*m_rad);}
    ...
  private:
    int m_rad;
};
```

```
int main() {
  Shape *s = new Square();
  Shape *c = new Circle();
  s->getArea();
  c->getArea();
}
```

In the above example, the pointer is of type base (Shape) but it points to the derived class (Square and Circle) object. The method `getArea()` is virtual in nature.

Hence in order to resolve the virtual method call, the context of the pointer is considered, i.e., the `getArea()` method of the derived class (Square and Circle) is called and not that of the base (Shape).

If the method was non virtual in nature, the `getArea()` method of the base class would have been called.

Polymorphism

Virtual Functions: Examples

UCLA Samueli Computer Science

```
class Shape {
  public:
    virtual double getArea()
    { return (0); }
    ...
  private:
    ...
};
```

```
class Square: public Shape {
  public:
    Square(int side){ m_side=side; }
    virtual double getArea()
    { return (m_side*m_side); }
    ...
  private:
    int m_side;
};
```

```
class Circle: public Shape {
  public:
    Circle(int rad){ m_rad=rad; }
    virtual double getArea()
    { return (3.14*m_rad*m_rad);}
    ...
  private:
    int m_rad;
};
```

```
int main() {
  Shape *s = new Square();
  Shape *c = new Circle();
  s->getArea();
  c->getArea();
}
```

I will not forget to add `virtual` in front of my destructors when I use inheritance/polymorphism.

→ *What is the problem if not?*

Polymorphism

Virtual Functions: Destructor

UCLA Samueli Computer Science

```
#include <iostream.h>
class base
{
public:
   ~base()
   {

   }
};

class derived : public base
{
public:
   ~derived()
   {

   }

};

void main()
{

   base *ptr = new derived();
   // some code
   delete ptr;
}
```

It is to be remembered that destructors are called in the reverse order of inheritance.

If a base class pointer points to a derived class object and we some time later use the delete operator to delete the object, then **the derived class destructor is not called.**

Polymorphism

Virtual Functions: Destructor

UCLA Samueli Computer Science

```
#include <iostream.h>
class base
{
public:
   virtual ~base()
   {

   }
};

class derived : public base
{
public:
   ~derived()
   {

   }

};

void main()
{

   base *ptr = new derived();
   // some code
   delete ptr;
}
```

As the pointer is of type base, the base class destructor would be called but the derived class destructor would not be called at all. **The result is memory leak.**

In order to avoid this, we have to make the destructor **virtual** in the base class.

Polymorphism

Virtual Functions? Constructor

UCLA Samueli Computer Science

- A constructor cannot be virtual because at the time when the constructor is invoked the virtual table would not be available in the memory.
  Hence we cannot have a virtual constructor.

Polymorphism

Pure Virtual Functions & Abstract Base Class

UCLA Samueli Computer Science

- Sometimes we have no idea what to implement in base functions. For example, without knowing what the animal is, **it is difficult to implement the speak() function.**
- Solution: Pure virtual functions
- Note:
  - Declare pure virtual functions in the base class. (=0!)
  - Considered as dummy function.
  - The derived class **MUST** implement all the pure virtual functions of its base class.
- If a class has at least one pure virtual function, it is called *abstract base class*.

```
class Shape {
  public:
    virtual double getArea()
    { return 0; }
    ...
  private:
    ...
};
```

Never actually used!

```
class Animal
{
  public:
    Animal();
    virtual ~Animal();
    int getAge() const;
    virtual void speak() const = 0;
  private:
    int m_age;
};
```

Group Exercises: Worksheet

UCLA Samueli Computer Science

- Exercise problems from **Worksheet #5** (see “LA worksheet” tab in CS32 website). Answers will be posted next week.
- Questions for today:
  - “I’m Gene” (Code Fix)
  - Cat Class (Implement Constructors)
  - Dog Class (Output)

Group Exercises: Worksheet #1

UCLA Samueli Computer Science

```
#include <iostream>
using namespace std;
class LivingThing {
   public:
     void intro() {
      cout << "I'm a living thing" << endl; }
};
class Person : public LivingThing {
   public:
     void intro() {
      cout << "I'm a person" << endl; }
};
class UniversityAdministrator : public Person {
   public:
     void intro() {
       cout << "I'm a university administrator" << endl; }
};
class Chancellor : public UniversityAdministrator {
   public:
     void intro() { cout << "I'm Gene" << endl; }
};
```

```
int main() {
   LivingThing* thing = new Chancellor();
   thing->intro();
}
```

- What does main() output?
- How do we make it output "I'm Gene"?

Group Exercises: Worksheet #1

UCLA Samueli Computer Science

```
#include <iostream>
using namespace std;
class LivingThing {
   public:
     virtual void intro() {
      cout << "I'm a living thing" << endl; }
};
class Person : public LivingThing {
   public:
     virtual void intro() {
      cout << "I'm a person" << endl; }
};
class UniversityAdministrator : public Person {
   public:
     void intro() {
       cout << "I'm a university administrator" << endl; }
};
class Chancellor : public UniversityAdministrator {
   public:
     virtual void intro() { cout << "I'm Gene" << endl; }
};
```

```
int main() {
   LivingThing* thing = new Chancellor();
   thing->intro();
}
```

Current output: "I'm a living thing"

How to change output:

- Remember that the **virtual** keyword allows us to change the behavior of base class functions

Group Exercises: Worksheet #2

UCLA Samueli Computer Science

```
class Animal {
  public:
      Animal(string name);
  private:
      string m_name;
};
class Cat : public Animal {
  public:
      Cat(string name, int amountOfYarn);
  private:
      int m_amountOfYarn;
};
class Himalayan : public Cat {
  public:
      Himalayan(string name, int amountOfYarn);
};
class Siamese: public Cat {
  public:
      Siamese(string name, int amountOfYarn, string toyName);
  private:
      string m_toyName;
};
```

Implement all of the constructors

Group Exercises: Worksheet #2

UCLA Samueli Computer Science

```
Animal::Animal(string name)
  : m_name(name){ }

Cat::Cat(string name, int amountOfYarn)
  : Animal(name), m_amountOfYarn(amountOfYarn) { }

Himalayan::Himalayan(string name, int amountOfYarn)
  : Cat(name, amountOfYarn) { }

Siamese::Siamese(string name, int amountOfYarn, string
toyName)
  : Cat(name, amountOfYarn), m_toyName(toyName) { }
```

- Must use an initializer list since the base class does not have a default constructor

Group Exercises: Worksheet #3-4

UCLA Samueli Computer Science

```
#include <iostream>
using namespace std;

class Pet {
  public:
      Pet() { cout << "Pet" << endl; }
      ~Pet() { cout << "~Pet" << endl; }
  };

class Dog : public Pet {
  public:
      Dog() { cout << "Woof" << endl; }
      ~Dog() { cout << "Dog ran away!" << endl; }
  private:
      Pet buddy;
};

int main() {
  Pet* milo = new Dog;
  delete milo;
}
```

- Are there any problems with the code?
- What will the main function print out?

Group Exercises: Worksheet #3-4

UCLA Samueli Computer Science

```
#include <iostream>
using namespace std;

class Pet {
  public:
      Pet() { cout << "Pet" << endl; }
      virtual ~Pet() { cout << "~Pet" << endl; }
  };

class Dog : public Pet {
  public:
      Dog() { cout << "Woof" << endl; }
      virtual ~Dog() { cout << "Dog ran away!" << endl; }
  private:
      Pet buddy;
};

int main() {
  Pet* milo = new Dog;
  delete milo;
}
```

- Always use a virtual destructor!
- Remember order of construction
  - Base Class
  - Member Variables
  - Constructor Body
- Remember order of destruction
  - Destructor Body
  - Member Variables
  - Base Class

Output:

```
Pet
Pet
Woof
Dog ran away!
~Pet
~Pet
```