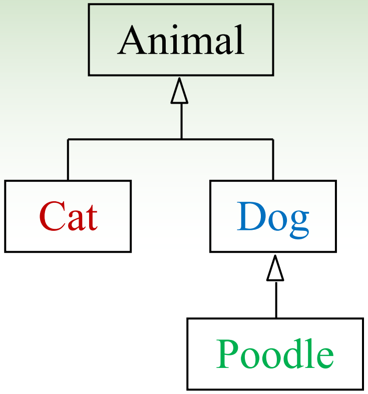
Polymorphism and Virtual Functions

Week 7

Type Compatibility in Inheritance Hierarchies

- Classes in a program may be part of an inheritance hierarchy
- Classes lower in the hierarchy are special cases of those above




```
class Animal {
public:
    void talk() {
     cout << "Animal sound ...\n";
    }
    void walk() {
      cout << "Animal walk\n";
    }
    void sleep() {
       cout << "Animal Sleep\n";
    }
};
```

```
class Cat:public Animal {
public:
  void talk() {
     cout << "cat hiss..\n";
  }
  void walk() {
        cout << "cat walk..\n";
  }
};
```

```
class Dog:public Animal {
public:
   void talk() {
       cout << "dog bark..\n";
   }
   void walk() {
       cout << "dog walk..\n";
    }
};
```

```
class Poodle:public Dog {
public:
    void talk() {
       cout << "poodle bark..\n";
    }
    void walk() {
       cout << "poodle walk..\n";
    }
};
```

Type Compatibility in Inheritance

- A pointer to a derived class can be assigned to a pointer to a base class.
- A base class pointer can point to derived class objects

```
auto main() -> int
{
    Cat *cptr = new Cat;

    Animal *aptr1 = cptr;

    Animal *aptr2 = new Dog;

}
```

Type Compatibility in Inheritance

- Assigning a base class pointer to a derived class pointer requires a cast
- The base class pointer must already point to a derived class object for this to work correctly.

```
auto main() -> int
{
    Animal *aptr = new Cat;

    Cat *cptr;
    cptr = static_cast<Cat *> (aptr);
}
```

Using Type Casts with Base Class Pointers

- C++ uses the declared type of a pointer to determine access to the members of the pointed-to object
- If an object of a derived class is pointed to by a base class pointer, all members of the derived class may not be accessible
- Type cast the base class pointer to the derived class (via **`static_cast`**) in order to access members that are specific to the derived class

Using Type Casts with Base Class Pointers

```
auto main() -> int
{
   Animal *aptr = new Dog;
   aptr->talk();
   aptr->walk();

   Cat *cptr = new Cat;
   cptr->talk();
   cptr->walk();

   Dog *dptr = static_cast<Dog *> (aptr);
   dptr->talk();
   dptr->walk();
}
```

```
Animal sound ...
Animal walk
cat hiss..
cat walk..
dog bark..
dog walk..
```


Casting in C++

- C++ is a strong-typed language.
- There are 6 different types of casts:
  - **C-style casts**,
  - **functional casts**,
  - **static casts**,
  - const casts,
  - dynamic casts, and
  - reinterpret casts.

```
int main () {
  int a=115, b=104, c=97, d=114, e=97,f=102;

  cout << a << b << c << d << e << f << endl;
  cout << char(a) << char(b) << char(c)
       << char(d) << char(e) << char(f)
       << endl;

  return 0;
}
```

```
1151049711497102
sharaf
```

```
int main () {
  float x = 3.5;
  int   y = x;
  cout << y << endl;

  int a = 10, b = 20;
  float c;
  c = a / b;
  cout << c << endl;

  c = float (a/b);
  cout << c << endl;
  c = (float)(a/b);
  cout << c << endl;

  c = float(a)/b;
  cout << c << endl;
  c = (float)a/b;
  cout << c << endl;
  c = static_cast<float>(a)/b;
  cout << c << endl;

  return 0;
}
```

```
3
0
0
0
0.5
0.5
0.5
```


Upcasting and downcasting

- **Upcasting** is a process of treating a pointer or a reference of derived class object as a base class pointer and performed automatically.
- **Downcasting** is converting base class pointer (or reference) to derived class pointer and must be explicitly done by programmer.

```
class Animal {
public:
    void talk() {
     cout << "Animal sound ...\n";
    }
    void walk() {
      cout << "Animal walk\n";
    }
    void sleep() {
       cout << "Animal Sleep\n";
    }
};
```

```
class Cat:public Animal {
public:
  void talk() {
     cout << "cat hiss..\n";
  }
  void walk() {
        cout << "cat walk..\n";
  }
  void run() {
      cout << "cat run..\n";
  }
};
```

```
class Dog:public Animal {
public:
   void talk() {
       cout << "dog bark..\n";
   }
   void walk() {
        cout << "dog walk..\n";
    }
    void run() {
        cout << "Dog run..\n"
    }
};
```

```
class Poodle:public Dog {
public:
    void talk() {
       cout << "poodle bark..\n";
    }
    void walk() {
       cout << "poodle walk..\n";
    }
};
```


Upcasting and downcasting

Upcasting

```
auto main() -> int
{
    Cat c;
    Animal *aptr = &c;
    aptr->talk();
    aptr->walk();

    // aptr->run();

}
```

downcasting

```
auto main() -> int
{
    Cat c;
    Animal *aptr = &c;

    // aptr->run();
    // Cat *cptr = aptr;
    Cat *cptr = static_cast<Cat *>(aptr);
    cptr->talk();
    cptr->walk();

    cptr->run();

}
```

dynamic_cast

- Given a pointer or a reference to an object of type B, dynamic_cast attempts to convert that object to a pointer or a reference to an object of type D if D is a sub class of B.
- dynamic_cast is an operator that converts safely one type to another type.
- dynamic_cast is used at **run-time** to find out correct down-cast.
  - The base class must have at least one virtual method
  - If the cast is successful, it returns a value of that type
  - If the cast fails and the type is a pointer, it returns a nullptr of that type.

```
class Animal{
public:
    virtual void talk()  {    cout << "Animal sound...\n"; }
    virtual void walk() {   cout << "Animal walk\n"; }
    virtual void sleep() {  cout << "Animal Sleep\n"; }
};
```

```
Animal *getAnimal ( ) {
    if ( rand()%2 == 0 )   return new Cat;
    else  return new Dog;
}
```

```
auto main() -> int
{
    srand(time(NULL));
    for (int i=0;i<5;i++)
    {
        cout << "........\n" << endl;
        Animal *aptr = getAnimal();
        cout << typeid(*aptr).name() << endl;

        Cat *cptr = dynamic_cast<Cat *> (aptr);

        if (cptr != nullptr)
        {
            cout  << "It is a cat \n";
            cptr->talk();
            cptr->walk();
        }
        else
            cout  << "It is a dog \n";
    }
}
```

```
........
3Dog
It is a dog
........
3Cat
It is a cat
cat hiss..
cat walk..
........
3Dog
It is a dog
........
3Dog
It is a dog
........
3Dog
It is a dog
```

static_cast

- static_cast is aimed at replacing **explicit casts**
- static_cast can be used to cast from one primitive type to another (such as int to char)

```
auto main() -> int
{
    Animal *aptr = new Cat;

    Cat *cptr = static_cast<Cat *> (aptr);

    cptr->talk();
    cptr->walk();
}
```

Must be a cat pointer

Must point to a cat object

reinterpret_cast (1)

- Converts between types by reinterpreting the underlying bit pattern.

```
class Apple{
public:
  void eatApple(){
    cout << "eating Apple" << endl;
  }
};

class Orange {
public:
  void eatOrange(){
    cout << "eating Orange" << endl;
  }
};
```

```
int main()
{
  Orange *o = new Orange();
  Apple *a = new Apple();

  Orange  *orange = reinterpret_cast<Orange*>(a);
  orange->eatOrange();

  return 0;
}
```

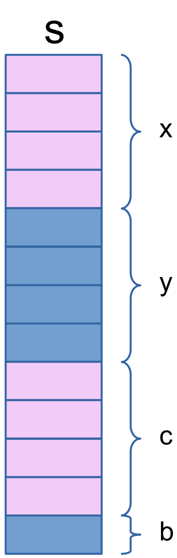
```
eating Orange
```

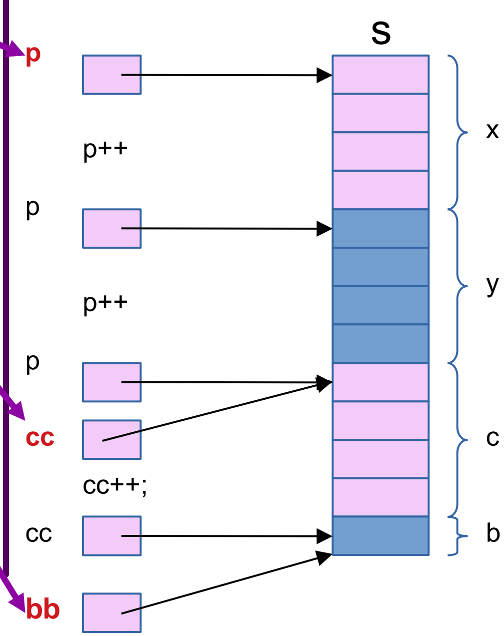

reinterpret_cast (2)

```
struct MyStruct{
  int x;
  int y;
  float c;
  bool b;
};
```

```
int main()
{
  MyStruct s {10,20,3.14,true};
  cout << s.x << endl
       << s.y << endl
       << s.c << endl
       << s.b << endl;

  int *p = reinterpret_cast<int*> (&s);
  cout << *p << endl;
  p++;
  cout << *p << endl;
  p++;
  float *cc = reinterpret_cast<float*>(p);
  cout << *cc << endl;
  cc++;
  bool *bb = reinterpret_cast<bool*> (cc);
  cout << *bb << endl;
  return 0;
}
```

```
10
20
3.14
1
10
20
3.14
1
```





const_cast

- Used to change the const or volatile qualifiers of pointers or references.

const_cast<T>(v)

- T must be a pointer, reference, or pointer to member type.

```
int main()
{
  const int constInt = 10;
  const int *constPtr = &constInt;

  int *d1 = const_cast<int*>(constPtr);
  *d1 = 15;

  int varInt = 20;
  const int *varPtr = &varInt;
  int *d2 = const_cast<int*> (varPtr);
  *d2 = 30;

  return 0;
}
```

Pointers, Inheritance, and Overridden Functions

If

- a derived class overrides a member function in the base class, and
- a base class pointer points to a derived class object,

then

the compiler determines the version of the function to use **by the type of the pointer**, **not by the type of the object**.

Polymorphism and Virtual Member Functions

- **Polymorphic code**: Code that behaves differently when it acts on objects of different types
- **Virtual Member Function**: The C++ mechanism for achieving polymorphism

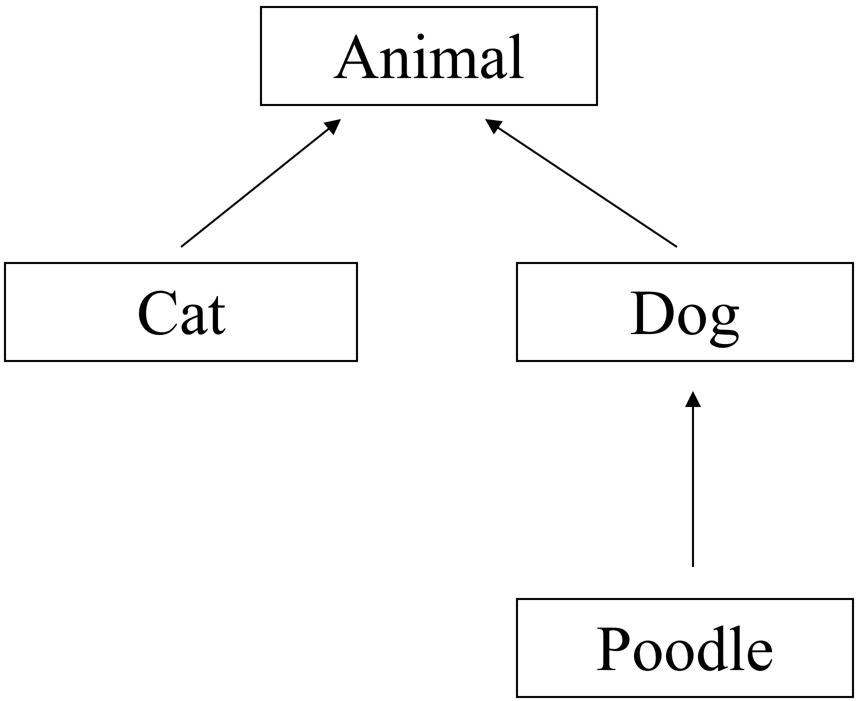
```
class Animal {
public:
    virtual void talk(){
     cout << "Animal sound ...\n";
    }
    virtual void walk(){
      cout << "Animal walk\n";
    }
    virtual void sleep(){
       cout << "Animal Sleep\n";
    }
};
```

```
auto main() -> int
{
    vector<Animal *> v;
    v.push_back(new Cat);
    v.push_back(new Cat);
    v.push_back(new Dog);
    v.push_back(new Cat);
    v.push_back(new Dog);

    for (auto x:v)
        x->talk();

    for (int i=0;i<v.size();i++)
        v[i]->walk();

}
```

Polymorphism

Consider the Animal, Cat, Dog hierarchy where each class has its own version of the member function id( )

```
class Animal {
 public:
   void id() {
       cout << "animal";
    }
};

class Cat : public Animal {
 public:
   void id() {
      cout << "cat";
   }
};

class Dog : public Animal {
 public:
   void id() {
      cout << "dog";
   }
};
```



Polymorphism

- Consider the collection of different Animal objects

- Prints: **`animal animal animal`**, **`animal`**, ignoring the more specific versions of **`id()`** in **`Dog`** and **`Cat`**

```
auto main() -> int
{
    Animal *arr[] = { new Animal, new Dog, new Dog, new Cat};

    for (int i=0;i<4;i++)
        arr[i]->id();
}
```

- The preceding code is not polymorphic: it behaves the same way even though **`Animal`**, **`Dog`** and **`Cat`** have different types and different **`id()`** member functions

- Polymorphic code would have printed "**`animal dog dog cat`**" instead of "**`animal animal animal animal`**"

Polymorphism

- The code is not polymorphic because in the expression

  `arr[i]->id();`

  the compiler sees only the type of the pointer `arr[i]`, which is pointer to `Animal`

- Compiler does not see type of actual object pointed to, which may be `Animal`, or `Dog`, or `Cat`

Virtual Functions

- Declaring a function **`virtual`** will make the compiler check the type of each object to see if it defines a more specific version of the virtual function

```
class Animal {
 public:
  virtual void id() {
      cout << "animal";
   }
};

class Cat : public Animal {
 public:
  void id() {
     cout << "cat";
  }
};

class Dog : public Animal {
 public:
  void id() {
     cout << "dog";
  }
};
```

```
auto main() -> int
{
    Animal *arr[] ={ new Animal, new Dog,
                     new Dog, new Cat};

    for (int i=0;i<4;i++)
        arr[i]->id();
}
```

```
animal
dog
dog
cat
```

Virtual Functions

- It is also possible to use virtual with the functions in the derived classes.
- Base class function must be virtual.

```
class Animal {
 public:
   virtual void id() {
       cout << "animal";
    }
};

class Cat : public Animal {
 public:
   virtual void id() {
      cout << "cat";
   }
};

class Dog : public Animal {
 public:
   virtual void id() {
      cout << "dog";
   }
};
```


Function Binding

- In `arr[i]->id()`, Compiler must choose which version of `id()` to use.
- There are different versions in the `Animal`, `Dog`, and `Cat` classes
- Function binding is the process of determining which function definition to use for a particular function call
- The alternatives are *static* and *dynamic* binding

- Static binding chooses the function in the class of the base class pointer, ignoring any versions in the class of the object actually pointed to Static binding is done at compile time

- Dynamic Binding determines the function to be invoked at execution time
- Can look at the actual class of the object pointed to and choose the most specific version of the function
- Dynamic binding is used to bind virtual functions

Virtual Tables

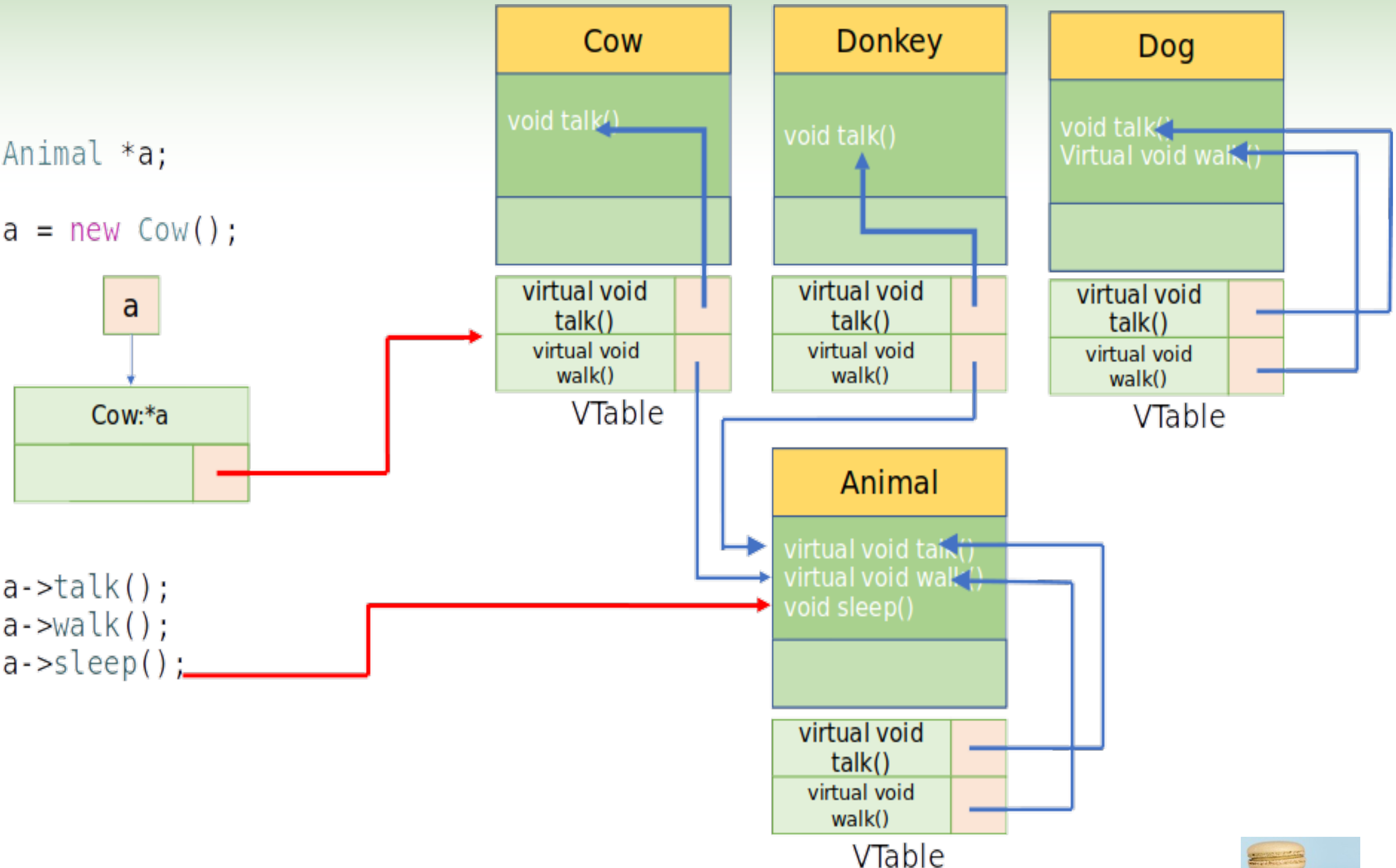
```
class Animal {
public:
    virtual void talk(){
        cout << "moo..heha..woof\n";
    }
    virtual void walk() {
        cout << "Animal walk\n";
    }
    void sleep() {
        cout << "Animal Sleep\n";
    }
};
```

```
int main( ) {
    srand((unsigned int)time(0));
    Animal *a;
    switch(rand()%3){
        case 0: a = new Cow();  break;
        case 1: a = new Donkey(); break;
        case 2: a = new Dog();
  }
    a->talk();
    a->walk();
    a->sleep();
    return 0;
}
```

```
class Cow:public Animal {
public:
    void talk() {
        cout << ”Moo…\n";
    }
};


class Donkey:public Animal {
public:
    void talk() {
        cout << ”bray…\n";
    }
};


class Dog:public Animal {
public:
    void talk(){
        cout << ”bark…\n";
    }

    void walk(){
        cout << "Dog walk..\n";
    }
};
```

Virtual tables

```
Animal *a;

a = new Cow();
```



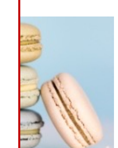

```
a->talk();
a->walk();
a->sleep();
```

Overriding vs. Overloading

- Recall that Overloaded functions have the same name but different signature (parameter list).
- If a base class has a **virtual member function** and a derived class has an overloading member function, use the **override** key word at the end of the function header or prototype to indicate that it should be treated as an overriding function.

```
class Animal {
public:
     virtual void talk() {
      cout << "moo..heha..woof\n";
    }
    virtual void walk() {
      cout << "Animal walk\n";
    }
    virtual void sleep() {
      cout << "Animal Sleep\n";
    }
};
```

```
class Cow:public Animal {
public:
    void talk() override {
       cout << "Moo…\n";
    }
};
class Donkey:public Animal {
public:
   void talk() override {
      cout << "bray…\n";
   }
};
class Dog:public Animal {
public:
    void talk() override {
       cout << "bark…\n";
    }
    void walk(){
       cout << "Dog walk..\n";
    }
};
```



The `final` Key Word and Overriding

- The key word **final** can be used at the end of the header or prototype of a function in an inheritance hierarchy if you want to ensure that no classes below this one override this function.
- If an attempt is made to override this function in a derived class, a compiler error will occur.

```
class Animal {
public:
    virtual void talk() {
      cout << "moo..heha..woof\n";
    }
    virtual void walk() {
      cout << "Animal walk\n";
    }
    virtual void sleep() final
    {
      cout << "Animal Sleep\n";
    }
};
```

```
class Cow:public Animal {
public:
    void talk() override {
       cout << "Moo…\n";
    }
    void sleep() {
       cout << "Cow Sleep\n";
    }
};
```

Generates a compiler error

Abstract Base Classes and Pure Virtual Functions

- An abstract class is a class that contains no objects that are not members of subclasses (derived classes)
- For example, in real life, Animal is an abstract class: there are no animals that are not dogs, or cats, or lions…
- Abstract classes are an organizational tool. They are useful in organizing inheritance hierarchies
- Abstract classes can be used to specify an interface that must be implemented by all subclasses

Abstract Functions

- The member functions specified in an abstract class do not have to be implemented
- The implementation is left to the subclasses
- In C++, an abstract class is a class with at least one abstract member function

Pure Virtual Functions

- In C++, a member function of a class is declared to be an abstract function by making it virtual and replacing its body with `= 0;`

```
class Animal{
  public:
   virtual void id()=0;
};
```

- A virtual function with its body omitted and replaced with `=0` is called a pure virtual function, or an abstract function

Abstract Classes

- An abstract class can not be instantiated
- An abstract class can only be inherited from; that is, you can derive classes from it
- Classes derived from abstract classes must override all pure virtual functions with concrete member functions before they can be instantiated.

```
class Animal {
public:
    virtual void talk()=0;
    virtual void sleep() {
      cout << "Animal Sleep\n";
    }
};

class Cow:public Animal {
public:
    void talk() override {
       cout << "Moo…\n";
    }
    void sleep() override {
        cout << "Cow Sleep\n";
    }
};
```

```
class Donkey:public Animal
{
public:
   void talk() override {
      cout << "bray…\n";
   }
};

class Dog:public Animal {
public:
    void talk() override {
       cout << "bark…\n";
    }
};
```

```
int main( ) {
   srand((unsigned int)time(0));

   //Animal an;
   //Animal ptr = new Animal;

   Animal *a;
   switch(rand()%3) {
      case 0: a = new Cow();
              break;
      case 1: a = new Donkey();
              break;
      case 2: a = new Dog();
   }
   cout << typeid(*a).name()
           << endl;
   a->talk();
   a->sleep();
   return 0;
}
```

3Dog
bark…
Animal Sleep

Composition vs. Inheritance

- Inheritance models an 'is a' relation between classes. An object of a derived class 'is a(n)' object of the base class
- Example:
  - an **`UnderGrad`** is a **`Student`**
  - a **`Mammal`** is an **`Animal`**
  - a **`Poodle`** is a **`Dog`**

Composition vs. Inheritance

- When defining a new class:
- Composition is appropriate when the new class needs to use an object of an existing class
- Inheritance is appropriate when
  - objects of the new class are a subset of the objects of the existing class, or
  - objects of the new class will be used in the same ways as the objects of the existing class

Composition

```
class Name {
    string first;
    string last;
public:
    Name(string f="X", string l="Y") {
        first = f;
        last = l;
    }
    void setFirst(string f){
        first = f;
    }
    void setLast(string l){
        last = l;
    }
    string getFirst(){
        return first;
    }
    string getLast(){
        return last;
    }
    operator string(){
        return first+" "+last;
    }
};
```

```
class Student {
    Name name;

public:
    Student () = default;

    Student (Name &name) {
        this->name = name;
    }

    operator string (){
        return (string)name;
    }
};
```

```
int main() {
    Name s1("John", "Watson");
    Student p(s1);
    cout << (string)p <<  endl ;
    return 0;
}
```

```
John Watson
```

Polymorphism and Virtual Functions

Week 7