3 Object Slicing

Again a silly example:

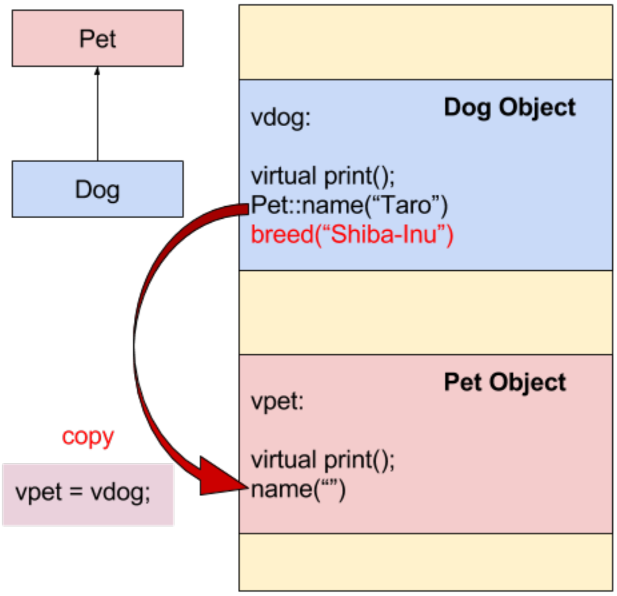
```
class Pet
{
public:
    virtual void print(){
        std::cout << "Name:" << name
                             << std::endl;
    }
    std::string name;
};
```

```
class Dog:public Pet
{
public:
    virtual void print(){
        std::cout << "Name:" << name << std::endl;
        std::cout << "Breed:" << breed << std::endl;
    }
    std::string breed;
};
```

C++ allows the following assignments:

```
int main(int argc,char *argv[]){
    Dog vdog;
    vdog.name = "Taro";
    vdog.breed = "Shiba-Inu";
    Pet vpet = vdog;

    vpet.print();
    return 0;
}
```



output

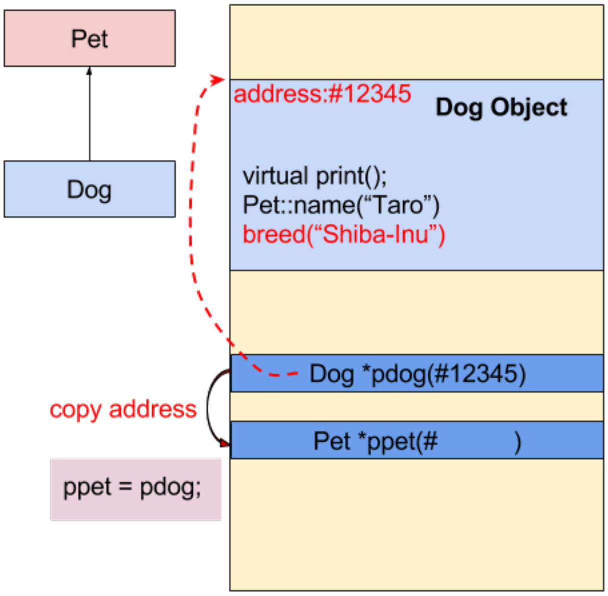
```
Name: Taro
```

- However, **vpet** will loose the **breed** member of **vdog** since an object of class **Pet** has no **breed** member
- This code would be **illegal: cout << vpet.breed;**
- This is the **slicing problem**
- It is legal to assign a derived class object into a base class variable
  - This slices off data in the derived class that is not also part of the base class
  - Member functions and member variables are lost
- It is possible in C++ to avoid the slicing problem
  - Using **pointers** to dynamic variables we can assign objects of a derived class to variables of a base class without loosing members of the derived class object

3.1 Dynamic Variables and Derived Classes

```
int main(int argc,char *argv[]){
    Dog *pdog = new Dog;
    pdog->name = "Taro";
    pdog->breed = "Shiba-Inu";
    Pet *ppet = pdog;

    ppet->print();
    return 0;
}
```



ppet->print( ); is legal and produces:

```
Name: Taro
Breed: Shiba-Inu
```

3.1.1 Use Virtual Functions

- The previous example:
  `ppet->print( );`
  worked because **print** was declared as a **virtual function**
- This code would still produce an **error**:
  `std::cout << "name: " << ppet->name << "breed: " << ppet->breed;`
- **ppet->breed** is still **illegal** because **ppet** is a pointer to a **Pet** object that has no **breed** member
- Function **print( )** was declared **virtual** by class **Pet**
- When the computer sees **ppet->print( )**, it checks the virtual table for classes **Pet** and **Dog** and finds that **ppet** points to an object of type **Dog**
- Because **ppet** points to a **Dog** object, code for **Dog::print( )** is used

To help make sense of object oriented programming with dynamic variables, remember these rules

1. If the domain type of the pointer **p_ancestor** is a base class for the domain type of pointer **p_descendant**, the following assignment of pointers is allowed
   `p_ancestor = p_descendant;`
   and no data members will be lost
2. Although all the fields of the **p_descendant** are there, virtual functions are required to access them