```
/*********** Shared_ptr ***********/

// 1. When a pointer outlives pointee: danling pointer
// 2. When a pointee outlives all its pointers: resource leak
//
// Smart Pointers: Make sure the lifetime of a pointer and pointee match.

class Dog {
    string m_name;
  public:
      void bark() { cout << "Dog " << m_name << " rules!" << endl; }
      Dog(string name) { cout << "Dog is created: " << name << endl;
m_name = name; }
      Dog() { cout << "Nameless dog created." << endl; m_name =
"nameless"; }
     ~Dog() { cout << "dog is destroyed: " << m_name << endl; }
        //void enter(DogHouse* h) { h->setDog(shared_from_this()); }  //
Dont's call shared_from_this() in constructor
};

class DogHouse {
    shared_ptr<Dog> m_pD;
public:
    void setDog(shared_ptr<Dog> p) { m_pD = p; cout << "Dog entered
house." << endl;}
};

int main ()
{
    shared_ptr<Dog> pD(new Dog("Gunner"));
    shared_ptr<Dog> pD = make_shared<Dog>(new Dog("Gunner")); // faster
and safer

    pD->bark();

    (*pD).bark();

    //DogHouse h;
//    DogHouse* ph = new DogHouse();
//    ph->setDog(pD);
//    delete ph;


    //auto pD2 = make_shared<Dog>( Dog("Smokey") ); // Don't use shared
pointer for object on stack.
//    auto pD2 = make_shared<Dog>( *(new Dog("Smokey")) );
//    pD2->bark();
//
//    Dog* p = new Dog();
//    shared_ptr<int> p1(p);
//    shared_ptr<int> p2(p);  // Erroneous

    shared_ptr<Dog> pD3;
    pD3.reset(new Dog("Tank"));
```

```
    pD3.reset();  // Dog destroyed. Same effect as: pD3 = nullptr;
//
    //pD3.reset(pD.get());  // crashes

    /********** Custom Deleter ************/
    shared_ptr<Dog> pD4( new Dog("Victor"),
                         [](Dog* p) {cout << "deleting a dog.\n"; delete
p;}
                         );
                         // default deleter is operator delete.

    //shared_ptr<Dog> pDD(new Dog[3]);
    shared_ptr<Dog> pDD(new Dog[3], [](Dog* p) {delete[] p;} );


/********** weak_ptr *********************/
class Dog {
      //shared_ptr<Dog> m_pFriend;
      weak_ptr<Dog> m_pFriend;
  public:
      string m_name;
      void bark() { cout << "Dog " << m_name << " rules!" << endl; }
      Dog(string name) { cout << "Dog is created: " << name << endl;
m_name = name; }
     ~Dog() { cout << "dog is destroyed: " << m_name << endl; }
     void makeFriend(shared_ptr<Dog> f) { m_pFriend = f; }
     void showFriend() { //cout << "My friend is: " << m_pFriend.lock()-
>m_name << endl;
                         if (!m_pFriend.expired()) cout << "My friend is:
" << m_pFriend.lock()->m_name << endl;
                         cout << " He is owned by " <<
m_pFriend.use_count() << " pointers." << endl; }
};

int main ()
{
    shared_ptr<Dog> pD(new Dog("Gunner"));
    shared_ptr<Dog> pD2(new Dog("Smokey"));
    pD->makeFriend(pD2);
    pD2->makeFriend(pD);

    pD->showFriend();
}


/********** unique_ptr *********************/

// Unique Pointers: exclusive owenership

class Dog {
      //Bone* pB;
      unique_ptr<Bone> pB;  // This prevents memory leak even constructor
fails.
  public:
```

```
      string m_name;
      void bark() { cout << "Dog " << m_name << " rules!" << endl; }
      Dog() { pB = new Bone(); cout << "Nameless dog created." << endl;
m_name = "nameless"; }
      Dog(string name) { cout << "Dog is created: " << name << endl;
m_name = name; }
     ~Dog() { delete pB; cout << "dog is destroyed: " << m_name << endl;
}
};

void test() {

    //Dog* pD = new Dog("Gunner");
    unique_ptr<Dog> pD(new Dog("Gunner"));

    pD->bark();
    /* pD does a bunch of different things*/

    //Dog* p = pD.release();
    pD = nullptr;
    //pD.reset(new Dog("Smokey"));

    if (!pD) {
        cout << "pD is empty.\n";
    }

    //delete pD;
}

void f(unique_ptr<Dog> p) {
    p->bark();
}

unique_ptr<Dog> getDog() {
    unique_ptr<Dog> p(new Dog("Smokey"));
    return p;
}

void test2() {
    unique_ptr<Dog> pD(new Dog("Gunner"));
    unique_ptr<Dog> pD2(new Dog("Smokey"));
    pD2 = move(pD);
    // 1. Smokey is destroyed
    // 2. pD becomes empty.
    // 3. pD2 owns Gunner.

    pD2->bark();
//    f(move(pD));
//    if (!pD) {
//        cout << "pD is empty.\n";
//    }
//
//    unique_ptr<Dog> pD2 = getDog();
//    pD2->bark();
```

```
    unique_ptr<Dog[]> dogs(new Dog[3]);
    dogs[1].bark();
    //(*dogs).bark(); // * is not defined
}

void test3() {
    // prevent resource leak even when constructor fails
}


int main ()
{
    test2();
}
```