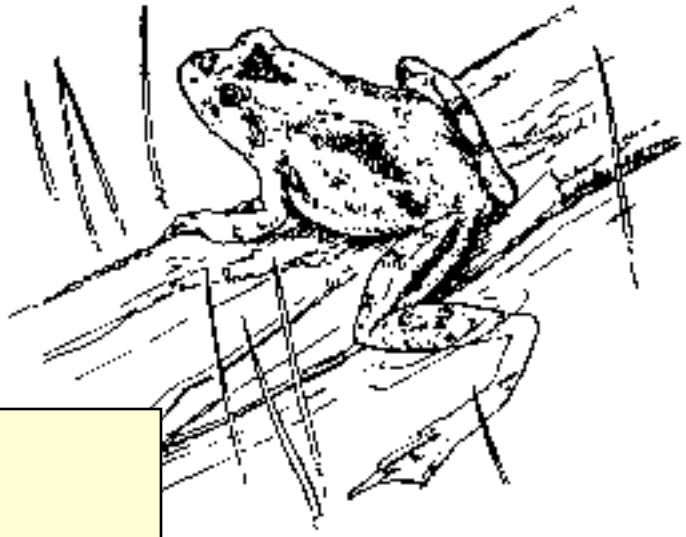
FRIEND
&
STATIC

Question



- Say we had a Node class:

```
class Node
{
   private:
      int data;
      Node* next;
};
```

- …but all the node manipulation was done in a separate class called **`LinkedList`**.
- Would **`LinkedList`** be able to access the next variable in an instance of **`Node`**?

Sometimes you just need a `friend`

- Sometimes you want external code (code that's not in the class) to be able to access private class variables!
- … but not anyone else.
- This can be done using the C++ **friend** keyword.

How It All Happens

- This is done by adding the “friend class ClassName” within the class:

```
class Node
{
    friend class LinkedList;

    private:
        int data;
        Node* next;
};
```

```
void LinkedList::doStuff()
{
     Node* ptr = head;

     // used to be illegal
     // now it's legal!
     ptr = ptr->next;
}
```

- Now LinkedList can access all variables in an instance of Node as if they were public.

Friend Declarations

```
class Node
{
   friend class LinkedList;

   private:  // ... etc
};
```

- Friend declarations can be put anywhere in the class – public section, private section, top, bottom, whatever
- The classes that you declare to be friends don't actually have to exist…
  - so watch for typos!
- A class can have lots of friends!

Friend Functions

```
class Node
{
   friend void breakStuff
();

   private:
      int data;
      Node* next;
};
```

```
void breakStuff()
{
    Node* ptr = head;

    // mwahahahahaha!!!
    delete ptr;
    ptr = NULL;
}
```

- Another option is to declare a single function to be a friend
- Here, the function `void breakStuff()` is allowed private access to Node
- How could we make a function in *another class* be a friend?

Friend Functions

```
class Node
{
   friend void breakStuff( int x, float q );

   private:
      int data;
      Node* next;
};
```

to make this work we have to put the entire function signature here!

- How would we make a member function of another class a friend? (Not the *entire* class – just a single member function)

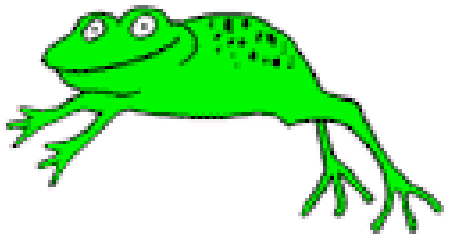
Friendliness

- Is **friend** a good idea or a bad idea?
- Does **friend** break the idea of encapsulation?
- When and why might you want to do this?

How to Remember This:

In C++, all your `friend`s can see your `private`s.

Static: Background



```
class Dog
{
   private:
      char name[50];
      int age;
};
```

```
int main()
{
   Dog gus;
   Dog pepper;
   Dog bitsy;
   Dog charlie;
   Dog toby;
   Dog checkers;
}
```

- Here we have 6 different instances of the class `Dog`.
- Each instance has its own set of member variables.
- So there are 6 different age variables – one per instance.

The Problem

- What if we wanted to keep a count of the number of instances of `Dog` in the entire program?
- Where would it make most sense to keep that counter?
  - A member variable in `Dog`?
  - A global variable?
- The ideal would be a counter that belongs to the *entire class* – not just a single instance of it.

Introducing static

- This can be done using the C++ keyword **static**.
- static variables are shared amongst all instances of the class – no one instance gets to “own” a static variable!
- Best way to think of a static: the lifetime of a global variable, but the access/scope of a class member variable (what’s the difference?)

Declaring Static Variables

```
class Dog
{
   private:
      char name[50];
      int age;

      // counter declaration
      static int counter;
};

// actual counter definition
int Dog::counter = 0;
```

- static variables are weird – we *declare* them inside the class, we *define* them outside the class
- Think of them like a function – the declaration is only a prototype!
- It still needs to be defined in the global scope (why?)

```
class Dog
{
   private:
      char name[50];
      int age;

      // counter declaration
      static int counter;
};

// actual counter definition
int Dog::counter = 0;
```

```
int main()
{
   Dog gus;
   Dog pepper;
   Dog bitsy;
   Dog charlie;
   Dog toby;
   Dog checkers;
}
```

So…

- There is a single `counter` variable in this program…
- To which of the 6 Dogs does `counter` "belong"?
- Which of them can access it?
- What do you think is the syntax for doing so? (inside the class)

Accessing Static Variables

- Static variables can be accessed exactly like regular variables!
- In addition, access specifiers (public, private, etc.) work the same way they do with non-static variables

```
class Dog
{
   public:
      Dog()
      {
         counter++;
      }

      ~Dog()
      {
         counter--;
      }

   private:
      char name[50];
      int age;

      // counter declaration
      static int counter;
};

// actual counter definition
int Dog::counter = 0;
```

Meanwhile, Outside the Class…

```
int main()
{
    Dog gus;
    Dog pepper;
    Dog bitsy;
    cout << bitsy.counter;

    Dog charlie;
    Dog toby;
    {
        Dog checkers;
        cout << gus.counter;
    }

    cout << toby.counter;
}
```

- As long as `counter` is public, it can be accessed outside the class as if it were a non-static variable

- What happens if there are no instances of `Dog` we can use to access `counter`?

Viva la variables estáticas!!

- Static member variables *always* exist – whether the class has ever been instantiated or not!
- We can access them using the scope resolution operator:

```
int main()
{
    cout << Dog::counter << endl;
}
```

- This works because `counter` belongs to the *class* `Dog` – not any one *instance* of `Dog`!

Static Methods

- Methods (member functions) can also be static!
- Static methods:
  - don't belong to any particular instance of the class
  - can be called even if there are *no* instances of the class!
  - can *not* access any non-static data in the class

Broke

- In this example program, `getCount()` tries to access both `age` and `counter`
- If we were to do this:

```
cout << Dog::getCount();
```

- … which instance of `age` would the function access?
- (This doesn't compile, by the way!)

```
class Dog
{
   public:
      static int getCount()

   private:
      char name[50];
      int age;

      // counter declaration
      static int counter;
};

int Dog::getCount()
{
   age++;
   return counter;
}

// actual counter definition
int Dog::counter = 0;
```

The Rules:

- Can static methods access:
  - Static member variables? - ***Yes!***
  - Non-static member variables? - ***No!***

- Can regular methods access:
  - Static member variables? - ***Yes!***
    - but there's only one copy to be shared amongst all instances of the class
  - Non-static member variables? - ***Yes!***
    - This is the normal case

Non-Class Static Variables

- Regular variables can be static too - not just class member variables
- Just like class static variables, regular static variables have:
  - **global** lifetime
  - **local** scope
- Static variables are only initialized *once*

```
void test()
{
   static int bob = 1;
   cout << bob++ << endl;
}

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