Class relationships:
---- ------------

Now that you have some experience with C++ classes, it's time to talk about relationships.

You may have wondered, "if my class can contain integers and floating point numbers, can it contain user-defined types?" And the answer to that is "yes".

Consider a class for "Animal". The class should contain the animals common name and the animals scientific name, among other things.

A string class is available for string manipulation. Unlike string constants, the string class supports string operations, like lexographical comparison, substring searches, & cetera.

It is logical for an Animal to take advantage of the string class. One reasonable choice is: let an Animal's common and scientific name be represented by instances of the string class.

Your definition of the Animal class might look like this.

```
#include <mstring.h>
class Animal {


private:

  string _commonName;      // the animal's common name
  string _scientificName;  // the animal's scientific name
.
.
.
}
```

This is called a has relationship between two classes.

* The member data `_commonName` and `_scientificName` are initialized when one of the constructors is called.

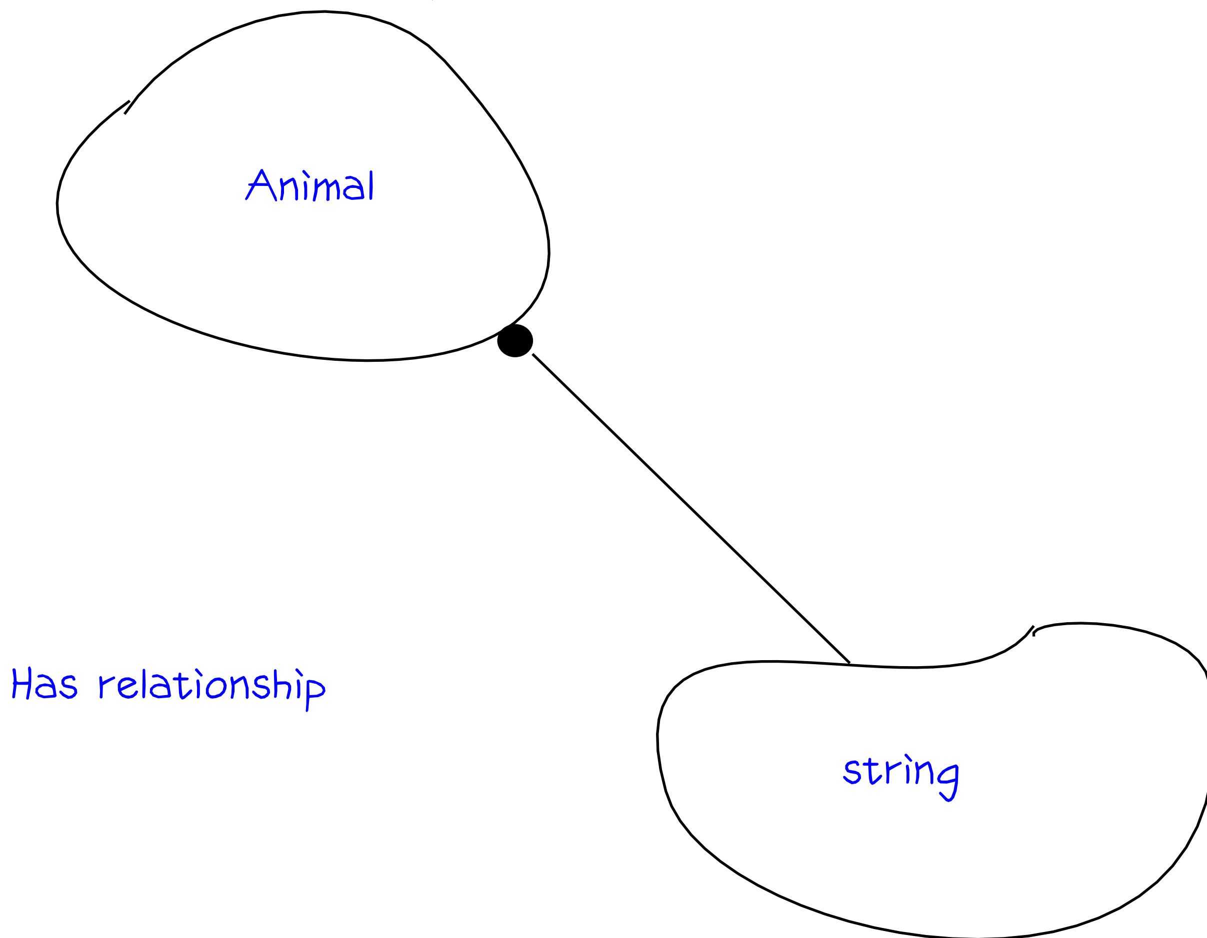
* The initialization syntax for member data can be used at this time: `variable(value)`, eg `_commonName("rabbit")`

* If the variables are not explicitly initialized as above, they will be initialized with their default constructor.

* ==> if there is not default constructor, they must be initialized as shown!

The member data is destroyed when the class that contains them is destroyed.

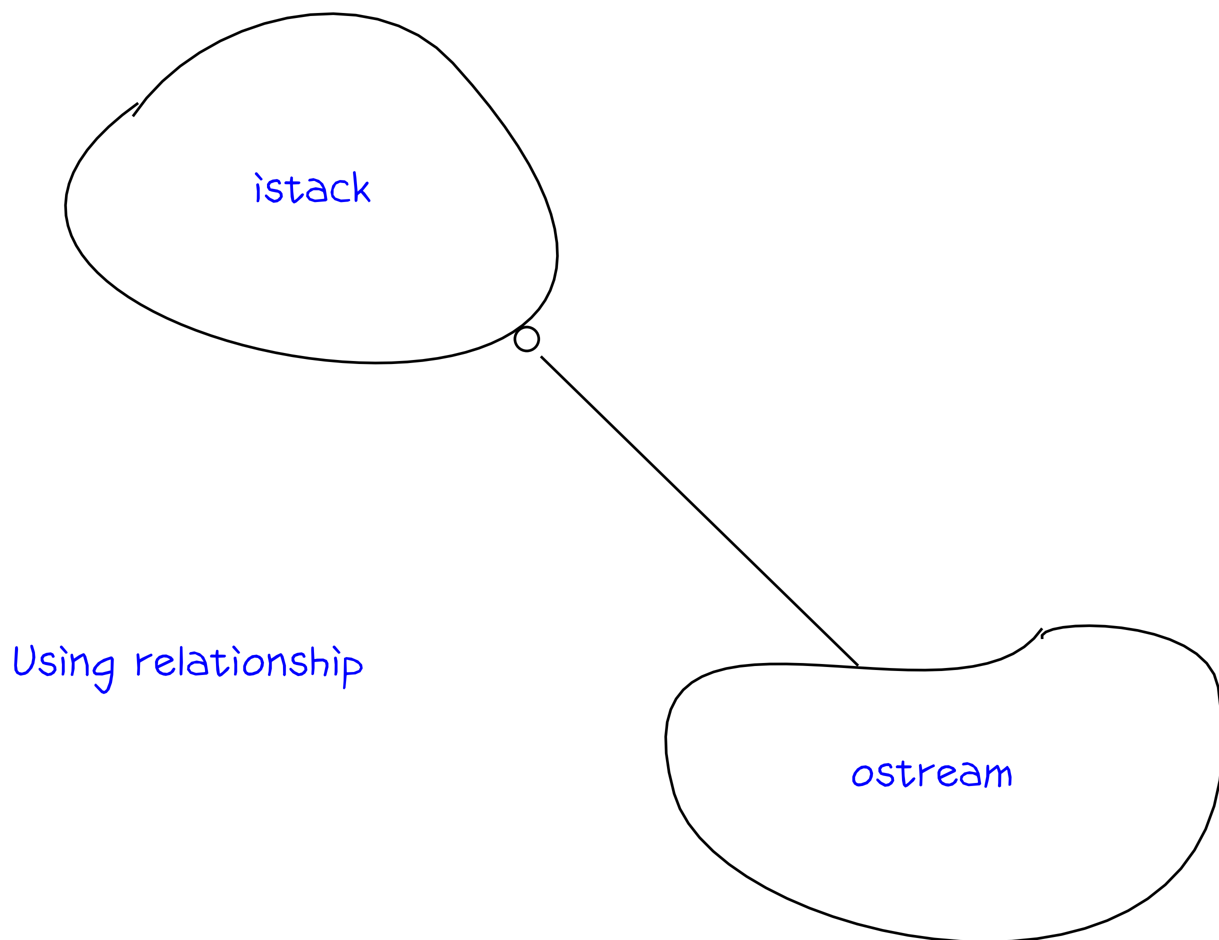
This relationship has a graphical representation:



An object might not "have" an object, but still "use" an object. This can happen is several ways. Your stack class presently "uses" the class ostream, because it uses cout, which is an ostream. The "using" relationship is weaker than the "has" relationship. It merely means that one class relies in some way upon the services offered by another class to carry out its own duties.

To use another class, it merely needs to include its header file, and link with that classes's object code.

The "using" relationship also has a graphical representation.



* Another way in which one class "uses" another class, is if one of its member functions takes a parameter which is an instance of another class.

* The cout object is an instance of the class ostream.

Exercise: rewrite the print method of your stack class so that it takes a single parameter of type ostream &:

```
class istack {

.
.
.
void print( ostream & myOutputStream);
.
.
}
```

Then change your test program(s) so they call the new print method rather than the old print method:

```
s.print(cout);
```

rather than:

```
s.print();
```

Your new print routine is fancier than your old one. To see why let's see how file I/O is done in C++.

File I/O is done using fstream objects. To access them you need to include a header file:

```
#include <fstream.h>
```

The constructor takes a string constant representing the name of the file as an argument. It actually opens the file if it can.

```
ofstream myFileStream ("filename");
```

The shift operator is defined for fstream objects, just like it is for ostream objects:

```
myFileStream << "Hello, File!" << endl;
```

When the destructor is called, the file is automatically closed.

Exercise: Try writing the helloFile program, to write a greeting onto a file.

Now, here comes a suprise.

Exercise: Try connecting your print method to an `ofstream` object rather than an `ostream` object; e.g, instead of:

```
s.print(cout);
```

change the statement to

```
s.print(myFileStream);
```

where myFileSteam is an *ofstream* object, as in the previous exercise.

But, how can this work??? Your print function takes an *ostream &* not an *fstream &* !!!

The answer is that there is yet another relationship between classes. This relationship is the tightest relationship that can exist between classes, and leads to some of the most powerful features of C++.

The relationship can best be described as an "is" relationship. An `ofstream` object is an `ostream` object.

Instances of the ofstream class have two data types: they have the `ofstream` type, and they have the `ostream` type.

When it is appropriate, you can tell C++ that one class inherits from another.

This says that one class ("Animal") represents a more general concept than another class ("Rabbit"),

That anything which is true ("Eats") of an object of one class ("Animal") is true of an object of another class ("Rabbit"), but not vice-versa.

If you need an animal, a rabbit will do, but if you need a rabbit, an animal will not do.

This is the proper way to understand the "is" relationship between classes. This relationship is called an inheritance relationship.

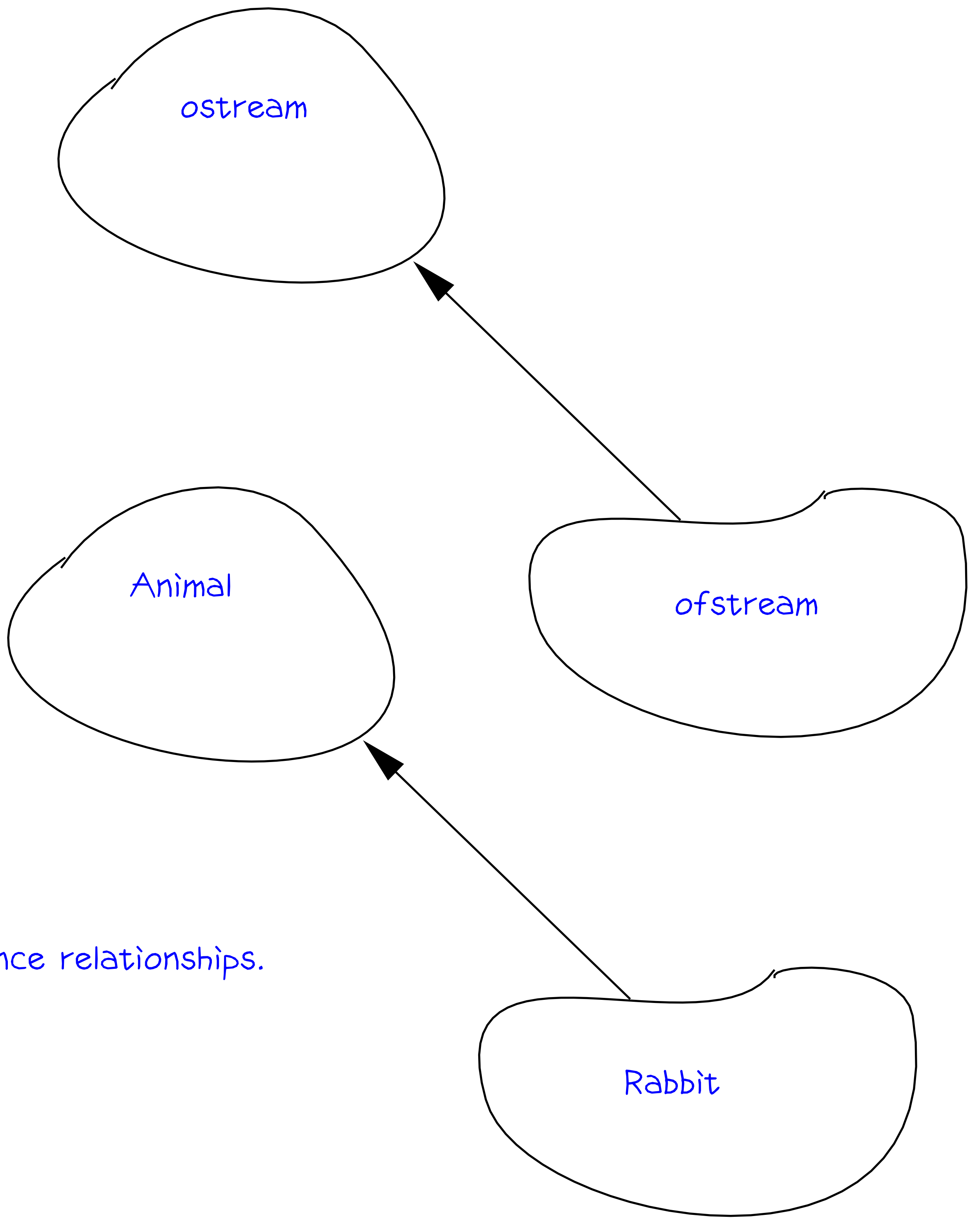
Lisikov principal: two classes should be related through inheritance if an "isa" relationship exists.

Inheritance is unfortunately very often misunderstood.

When one adheres to the principal, and uses is as a basis for class design, the rewards are immense.

The same function works on different classes of data.

It even works on classes of data that have not yet been written!



inheritance relationships.

Basic inheritance.

* Inheritance comes in three varieties: public, private, and protected.

* Forget about private and protected inheritance. We will talk about public inheritance only.

* A class that inherits from another

    -inherits its entire public interface.
    -is called a subclass or a derived class

* A class with a derived subclass is called a superclass.

* Methods that are defined in the superclass do not need to be redefined in the subclass. This saves time certainly.

* A mechanism exists to let subclasses override functions that are defined on superclasses. We defer that till later.

* Don't be tempted to use inheritance to save work.

Example: Suppose I have written a class that describes matrices. I now need a class to describe rotations. Should I choose to make a rotations a subclass of matrix? No. There are many methods that will normally be defined on a matrix that make no sense for a rotation: transpose, rank, numberOfRows, ... Sure it's a shortcut but it violates Lisikov AND makes it impossible to write meaningful code targeted to a type and all its subtypes.

OK, after all that hot air, here's how to make a subclass:

```
// Here is a class description for an Animal,
// which represents the highest level of
// generality
class Animal {

public:

 Animal();
 void wakeUp();
 void eat();
 void sleep();
 void haveTypicalDay();

private:
  .
};

// The class mammel represents a middle level
// of generality
class Mammal: public Animal {

public:

  void lactate();

private:
  .
};

// The class rabbit represents the lowest level
// of generality
class Rabbit: public Mammal {

public:

  void fornicate();

};

// A whale is also a mammal
class Whale: public Mammal {

public:

 void spout();

};
```