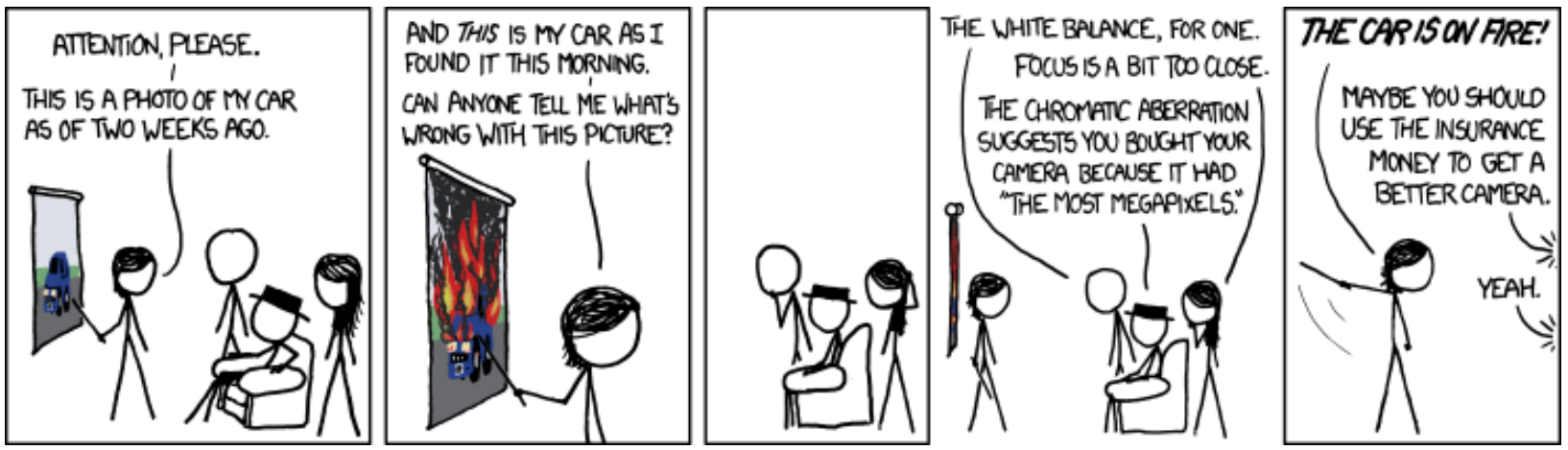
lab_inheritance Insidious Inheritance

Due: Sunday, September 20 at 11:59 PM

Assignment Description

In this lab you will get experience with some of the implementation issues and conceptual details of inheritance. Inheritance is a mechanism for increasing the reusability and reliability of C++ code. It is worth mentioning that inheritance is a characteristic of all object oriented programming languages. Our goal is to give you a glimpse of the functionality of inheritance, so that you can make informed design decisions in the future. Please read through the entire lab before you begin. The compilation notes at the bottom will tell you how to organize your development files.



Car Problems (XKCD #1014)

Checking out the code

After reading this lab specification, the first task is to check out the provided code from the class repository. To check out your files for this lab, type the following into the command console:

```
svn up
```
TERMINAL

This will create a new directory in your working directory called `lab_inheritance`.

The code for this activity resides in the `lab_inheritance/` directory. Get there by typing this in your working directory:

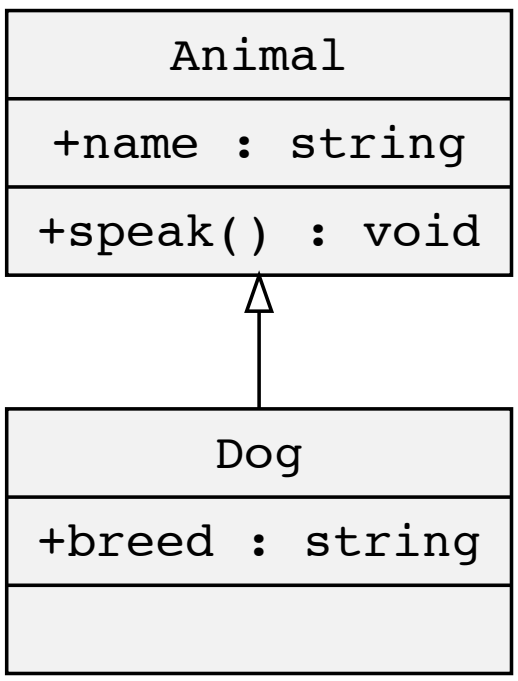
```
cd lab_inheritance/
```
TERMINAL

You will only need to modify the following files:

- `shape.{cpp,h}`
- `circle.{cpp,h}`
- `truck.{cpp,h}`
- `flower.{cpp,h}`
- `drawable.h`

Class Hierarchy

To help us understand class hierarchies better here is an example of a simple class hierarchy showing that a `Dog` **is a** `Animal`.



The code would look something like the following:

```C++
class Animal {
   public:
      string name;

      virtual void speak() = 0;
      /* The = 0 at the end of the method means that the method is a p
       * meaning that it does not have an implementation and it delega
       * of implementing the method to the classes that is derived fro
};

class Dog : public Animal {
   public:
      string breed;

      /* Dog inherits speak from Animal */
      void speak();

};

void Dog::speak() {
```

```
    cout << "Woof Woof" << endl;
}
```

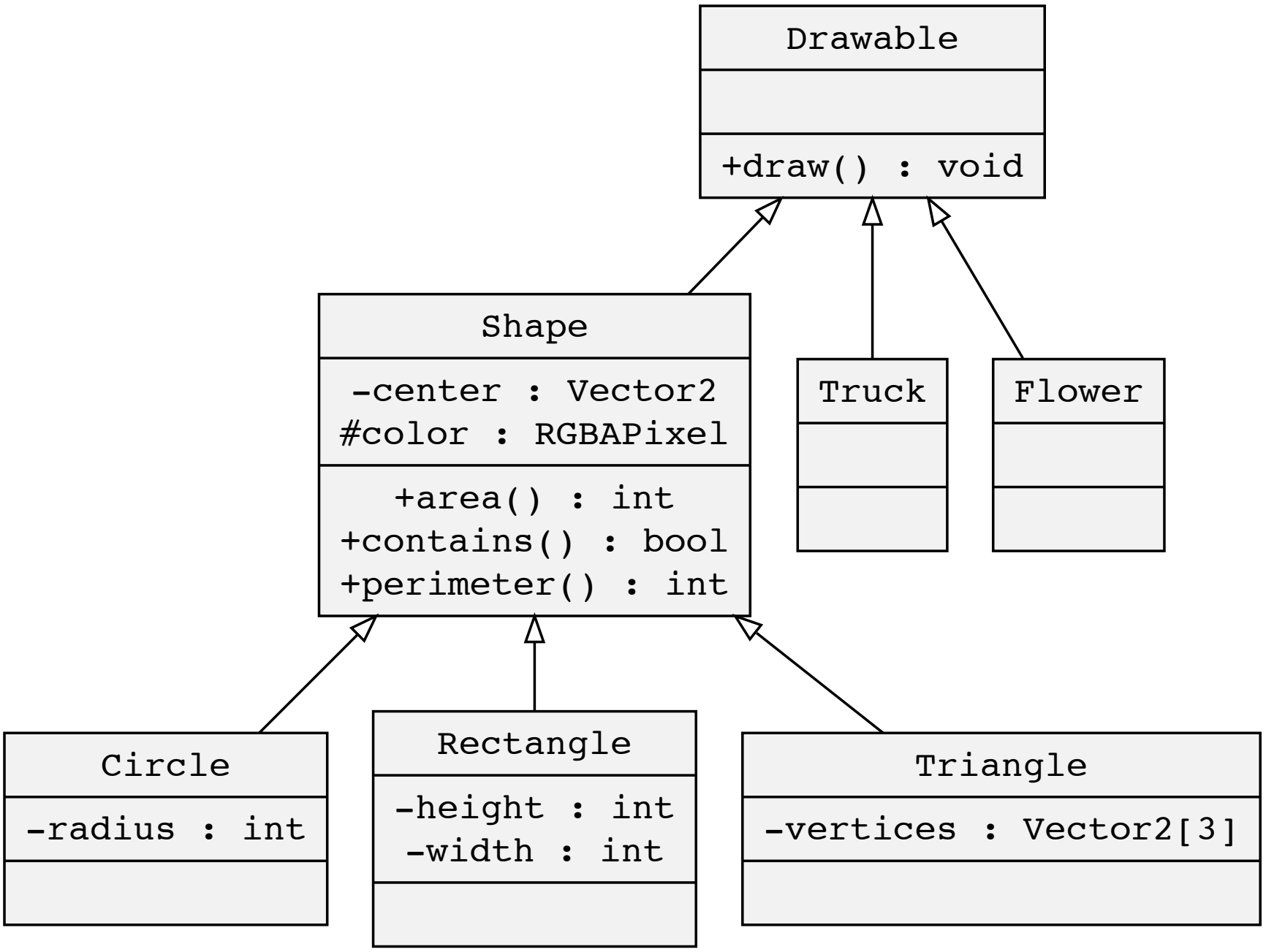
In this example `Animals` have a `name` and can `speak` but since `speak` is a pure virtual method we CANNOT construct an `Animal` by itself. That is `Animal` is an abstract class and it can only be constructed by one of its derived classes. For example, a `Dog` is a derived class of Animal. This means that a Dog **is a** `Animal`, and, therefore, it inherits a `name` and a `speak` method from `Animal`. However, since the `Animal`'s `speak` does not have an implementation, `Dog` **must** implement the `speak` method.

Here is an example of how we could use a `Dog` object:

C++

```
Dog* d = new Dog();

/* Like usual we can access all the public methods and member variable
 * Dog */
d->breed;

/* But now since a Dog is an Animal we can also do this too */
d->name;     // inherited from Animal
d->speak();  // inherited from Animal and since it is a Dog speak() wi
             // "Woof Woof"

/* Additionally we can treat our Dog only like an Animal like this */
Animal* a = d;

/* But now we can only do the following */
a->name;
a->speak();  // Still prints "Woof Woof" because speak is a virtual me

a->breed     // ERROR! This will NOT work since we perceive it as an A

/* Additionally, if we try to have our Animal pointer point back to a
 * pointer this will cause a problem because an Animal Is NOT A Dog. *
Dog* d2 = a;  // ERROR!  Animal Is NOT A Dog

/* Furthermore, since Animal is abstract and has a pure virtual method
 * we CANNOT construct one! */
Animal a2;  // ERROR! Animal is an abstract class
```

Now that we can understand a simple class hierarchy, let's look at a more complex one. Here is a diagram depicting the class hierarchy that is used in this lab. (Note: This diagram is missing some information, e.g. methods, member variables, etc.., for demonstration purposes)



This means everything is a `Drawable` and will have a `draw` method. Code like the following is perfectly acceptable:

```C++
Drawable* triangle  = new Triangle(....);
Drawable* circle    = new Circle(...);
Drawable* rectangle = new Rectangle(....);
Drawable* truck     = new Truck(...);
Drawable* flower    = new Flower(....);

/* Now the only thing we can use on triangle, circle, rectangle, truck
 * flower is draw but what gets drawn will change depending on what ty
 * pointer is actually pointing to. This is called polymorphism, the b
 * changes depending on the actual type of the object being pointed to

PNG canvas;
triangle->draw(&canvas);   // draws a Triangle even though triangle is
circle->draw(&canvas);     // draws a Circle even though circle is a D
rectangle->draw(&canvas);  // draws a Rectangle even though rectangle
truck->draw(&canvas);      // draws a Truck even though truck is a Dra
flower->draw(&canvas);     // draws a Flower even though flower is a D
```

Look at `main.cpp` for a working example executable in action. `main.cpp` gets compiled and linked into an executable named `lab_inheritance`. Follow the instructions below to build, run, and view the output:

The `Makefile` provided for this MP will create two useful executables when you run `make`. It will generate `lab_inheritance` and `lab_inheritance-asan`. So when you want to test a specific part of your MP you can use either of those. For example when you run

```
./lab_inheritance
```
TERMINAL

You could also run it as:

```
./lab_inheritance-asan
```
TERMINAL

if you want to test it with Address Sanitizer, and to check for memory leaks:

```
./lab_inheritance-asan
```
TERMINAL

Additionally you're free to run `Valgrind` on the normal executable:

```
valgrind --leak-check=full ./lab_inheritance
```
TERMINAL

This lab will use all of these objects in interesting ways but as you will quickly see they are not working the way the should. Your objective for this lab is to go through the 5 test executables and fix the code to work correctly by modifying how the classes in the hierarchy declare and implement their methods.

Exercise 1: Fix the Virtual Methods

Please build and run `test_virtual`:

```
make test_virtual test_virtual-asan # make test_virtual
./test_virtual-asan                 # run test_virtual with asan
valgrind ./test_virtual             # run test_virtual with valgrind
```
TERMINAL

As you will see when you run `test_virtual`, the output will say:

```
The Perimeters are NOT the same.
The Areas are NOT the same.
```

However, if you look closely at the code they should be the same because both of the pointers in `test_virtual.cpp` point to the same object!

Exercise

- Investigate and fix the code so that the areas and the perimeters are the same.
- To fix this problem you should only need to modify `shape.cpp` and/or `shape.h`.

Exercise 2: Fix the Destructor

Please build and run `test_destructor`:

```
make test_destructor test_destructor-asan    # make test_destructor
valgrind --leak-check=full ./test_destructor # run test_destructor in
./test_destructor-asan                       # test it with Address Sa
```
TERMINAL

When you run `test_destructor` in Valgrind or ASAN you will see that `test_destructor` is leaking memory. However, if you look closely, `Triangle` does have a valid destructor and it is being called in `test_destructor`!

Exercise

- Investigate and fix the code so that the there is no more memory leak inside of `test_destructor`.
- To fix this problem you should only need to modify `drawable.h` and `shape.h`.

Exercise 3: Fix the Constructor

Please build and run `test_constructor`:

```
make test_constructor # make test_constructor
./test_constructor    # run test_constructor
```
TERMINAL

When you run `test_constructor` you will see the following output:

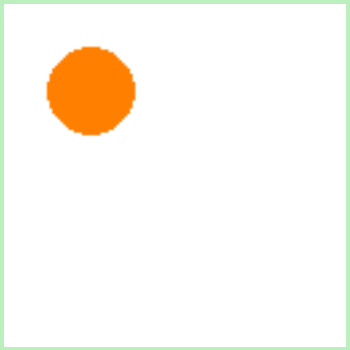
```
Circle's color is NOT correct!
Circle's center is NOT correct!
```

If you look closely, we are constructing a `Circle` with a valid center and color. However, when it is being drawn and when we ask for the `Circle`'s center and color they are not the same!

Exercise

- Investigate and fix the code so that the `Circle` is being constructed with the proper center and color.

- To fix this problem you should only need to modify `circle.cpp`
- The correct `test_constructor.png` should look like the following:



Exercise 4: Fix the Pure Virtual Method

Please build and run `test_pure_virtual`.

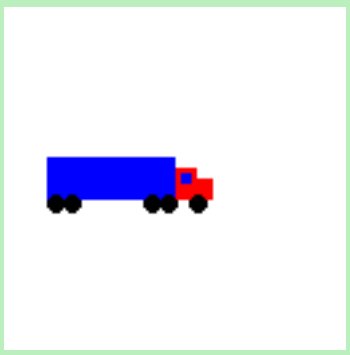
```
make test_pure_virtual # make test_pure_virtual
./test_pure_virtual    # run test_pure_virtual
```
TERMINAL

When you try to make `test_pure_virtual` you will see that it does not compile.

However, if you look at the `truck.{h,cpp}` it is a fully featured class! Why is it not compiling?

Exercise

- Investigate and fix the code so that `test_pure_virtual` compiles, runs, and outputs a `Truck`.
- To fix this problem you should only need to modify `truck.h` and `truck.cpp`.
- In order to have the `Truck` draw properly you will first need to have Exercise 3 completed.
- The correct `test_pure_virtual.png` should look like the following:



Exercise 5: Fix the Slicing

Please build and run `test_slicing` with:

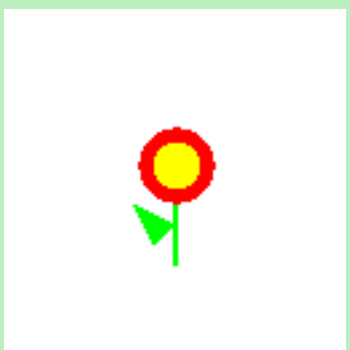
```
make test_slicing # make test_slicing
./test_slicing    # run test_slicing
```
TERMINAL

After you run `test_slicing` open up its output `test_slicing.png`. You will see that a `Flower` has NOT been drawn. For some reason just a bunch of X's has been drawn and a red circle.

If you look at `flower.h` and `flower.cpp`, we have all of the proper member variables set up. However, when we try to draw them they are drawn incorrectly.

Exercise

- Investigate and fix the code so that `test_slicing` outputs a `Flower`.
- To fix this problem you should only need to modify `flower.h` and `flower.cpp`.
- You must use polymorphism!
- The correct `test_slicing.png` output should look like the following:



Committing Your Code

To commit your code, simply run:

```
svn ci -m "lab_inheritance submission"
```
TERMINAL

Grading Information

The following files are used to grade this assignment:

- `shape.cpp`
- `shape.h`
- `circle.cpp`
- `circle.h`
- `truck.cpp`
- `truck.h`
- `flower.cpp`
- `flower.h`
- `drawable.h`