LAB4_INHERITANCE: Virtual Functions and Abstract Classes

Due date: **2025-04-07 23:59**.

Lab 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.

Download The Provided Files

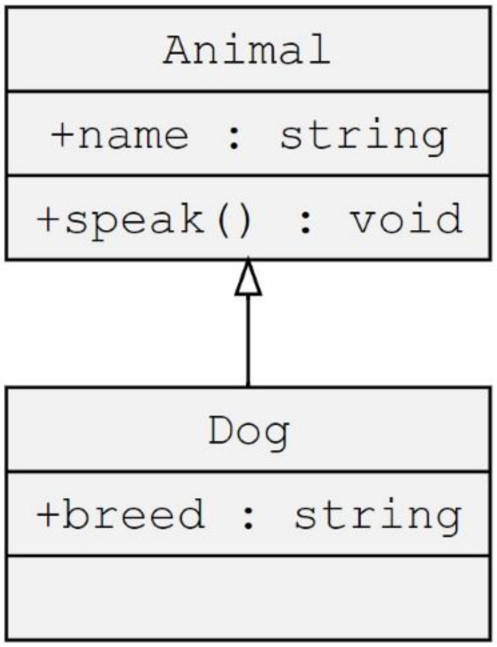
Just like you already did it in previous labs, start with downloading and unzipping the provided files from here lab4_inheritance.zip .

There will be many files in your **lab4_inheritance** directory, but you will only need to modify the following files

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

Background on Class Inheritance

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



The code would look something like the following:

```
class Animal {
   public:
      std::string name;

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

class Dog : public Animal {
   public:
      std::string breed;

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

};

void Dog::speak() const {
    std::cout << "Woof Woof" << std::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 an** `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.

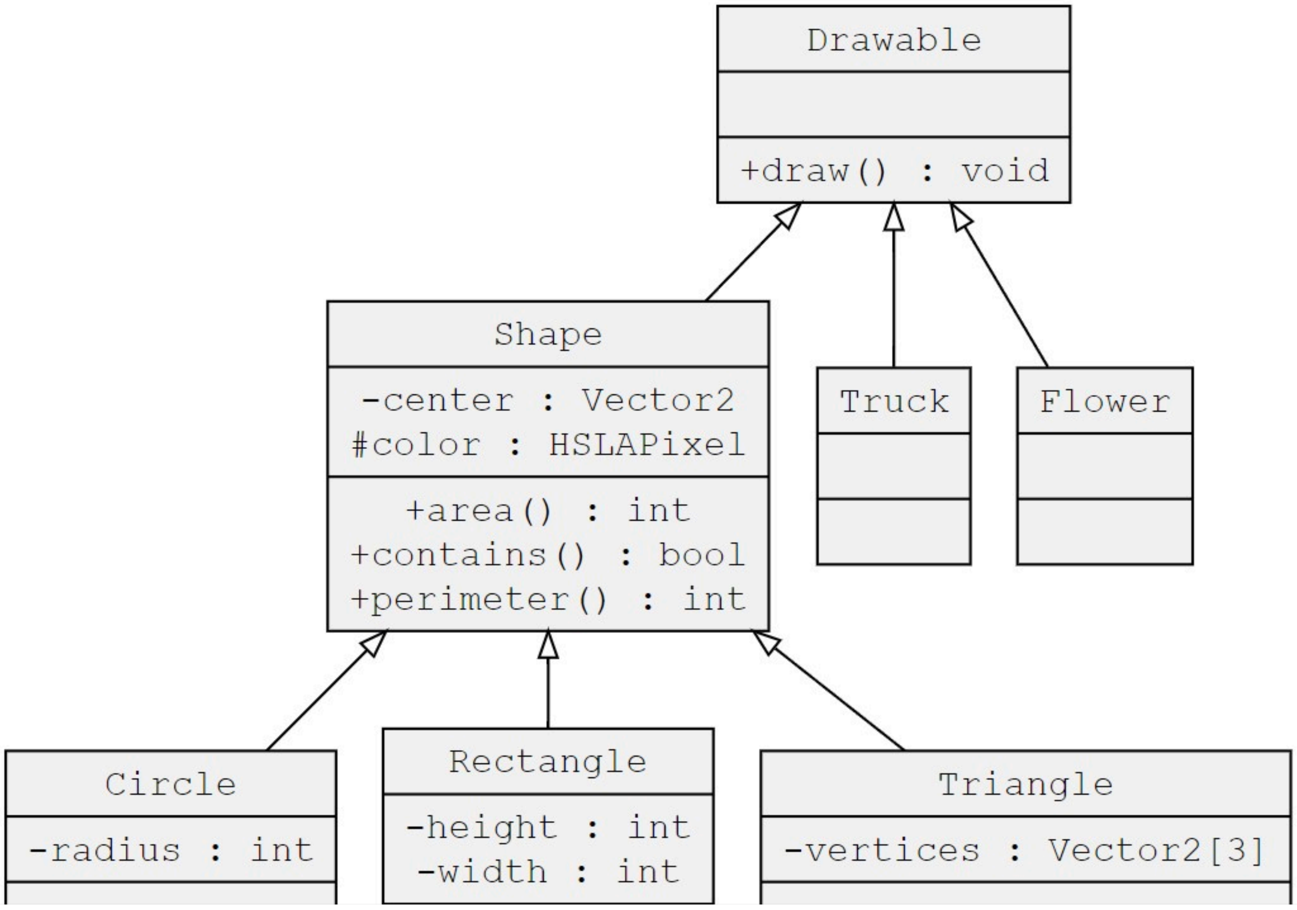
Here is an example of how we could use a `Dog` object:

```
std::shared_ptr<Dog> d = std::make_shared<Dog>();
```

```
/* 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() will print
              // "Woof Woof"

/* Additionally we can treat our Dog only like an Animal like this */
std::shared_ptr<Animal> a = d;

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

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

/* Additionally, if we try to have our Animal pointer point back to a Dog
 * pointer this will cause a problem because an Animal Is NOT A Dog. */
std::shared_ptr<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
```

Class Hierarchy

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:

```
std::unique_ptr<Drawable> triangle  = std::make_unique<Triangle>(....);
std::unique_ptr<Drawable> circle    = std::make_unique<Circle>(...);
std::unique_ptr<Drawable> rectangle = std::make_unique<Rectangle>(....);
std::unique_ptr<Drawable> truck     = std::make_unique<Truck>(...);
std::unique_ptr<Drawable> flower    = std::make_unique<Flower>(....);

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

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

Fully Working Code in main.cpp

Look at **main.cpp** for a working example executable. **main.cpp** gets compiled and linked into an executable named **lab_inheritance**. Follow the instructions below to build, run, and view the output. Note that unlike most labs, this main executable is not what will be tested, but rather it is an example for you to look at to help understand how to fix the other parts.

The **Makefile** provided for this lab will create several useful executables when you run `make`. Two of them are **lab_inheritance** and **lab_inheritance-asan**. There are other executables that correspond to the broken parts of the code, but you can also compile them separately later. So when you want to test a specific executable, you can run either the executable itself, or the **-asan** version of it to detect any memory errors. For example, you could run:

```
./lab_inheritance
```

You could also run it as:

```
./lab_inheritance-asan
```

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

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

The output of running `./lab_inheritance` is **out.png**. You can see that the code in **main.cpp** has no bugs, as the output matches the provided **soln_out.png** file. However, this is not the case for other executables in this lab, you will quickly see that they are not working the way they should. Your objective for this lab is to go through the five **test_*** executables and fix the code to work correctly by modifying how the classes in the hierarchy declare and implement their methods.

A Note on the Output .png Files

exercises, compare your output **.png** to the correct solution **soln_*.png** while also making sure the console output is correct and there are no memory leaks.

Exercise 1: Fix the Virtual Methods

Start with building and running **test_virtual**:

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

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!

Checklist:

- 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**.

Submit Your Solution

0.00 / 1

0 answers
Frequently Asked Questions

Upload your **shape.cpp** and **shape.h** here. Make sure the area and perimeter match and that there are no memory leaks.

Allowed filenames: shape.h, shape.cpp

Submit your files here: Choose Files No file chosen

Send answer

Exercise 2: Fix the Destructor

Please build and run **test_destructor**:

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

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**!

Checklist:

- Investigate and fix the code so that 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**.

0.00 / 1

0 answers
Frequently Asked Questions

Fix the Destructor

Make sure `test_destructor` compiles and runs without memory leaks.

Allowed filenames: drawable.h, shape.h
Submit your files here: Choose Files No file chosen
Send answer

Exercise 3: Fix the Constructor

Please build and run **test_constructor**:

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

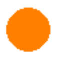
When you run **test_constructor** you will see the following output:

```
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!

Checklist:

- 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:



0.00 / 1

0 answers
Frequently Asked Questions

Fix the Constructor

Ensure that the terminal output is correct and that the produced **test_constructor.png** looks correct.

Allowed filenames: circle.cpp
Submit your files here: Choose Files No file chosen

Exercise 4: Fix the Pure Virtual Method

Please build and run **test_pure_virtual**.

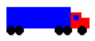
```
make test_pure_virtual # make test_pure_virtual
./test_pure_virtual # run test_pure_virtual
```

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

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

Checklist:

- 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:



0.00 / 1

0 answers
Frequently Asked Questions

Fix the Pure Virtual Method

Ensure that `test_pure_virtual` compiles and runs without issues, and that the output image **test_pure_virtual.png** matches the solution exactly. Pay attention to the order in which the different elements are drawn such that overlapping shapes are drawn as intended.

Allowed filenames: truck.h, truck.cpp

Submit your files here: Choose Files No file chosen

Send answer

Exercise 5: Fix the Slicing

Please build and run **test_slicing** with:

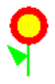
```
make test_slicing    # make test_slicing
./test_slicing       # run test_slicing
```

After you run **test_slicing** open up its output **test_slicing.png**. You will see that a `Flower` has NOT been drawn

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.

Checklist:

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



Fix the Slicing

Make sure that `test_slicing` compiles and runs without issues and that the produced **test_slicing.png** looks correct!

0.00 / 1

0 answers
Frequently Asked Questions

Allowed filenames: flower.h, flower.cpp
Submit your files here: Choose Files No file chosen
Send answer

Acknowledgments

We would like to express our gratitude to prof. Cinda Heeren and the student staff of the UIUC CS Data Structures course for creating and sharing the programming exercise materials that have served as the basis for the labs used in this course.

Edited by: Elmeri Uotila and Anna LaValle

Give feedback on this content

Comments about the task?
Enter your answer here
Send feedback