Polymorphism
Object-Orientated Programming (OOP) concept that a single object may take on the type of any of its base types.
- A Planet may polymorph itself to a Sphere
- A Sphere cannot polymorph to be a Planet (base types only)

Virtual
- The `virtual` keyword allows us to override the behavior of a class by its derived type.

<table>
<thead>
<tr>
<th>Sphere.cpp</th>
<th>Planet.cpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere::print_1()</td>
<td></td>
</tr>
<tr>
<td>Sphere::print_2()</td>
<td></td>
</tr>
<tr>
<td>virtual Sphere::print_3()</td>
<td></td>
</tr>
<tr>
<td>virtual Sphere::print_4()</td>
<td></td>
</tr>
<tr>
<td>virtual Sphere::print_5() = 0;</td>
<td></td>
</tr>
<tr>
<td>// No print_1() defined // in Planet</td>
<td></td>
</tr>
<tr>
<td>Planet::print_2()</td>
<td></td>
</tr>
<tr>
<td>// No print_3() defined // in Planet</td>
<td></td>
</tr>
<tr>
<td>Planet::print_4()</td>
<td></td>
</tr>
<tr>
<td>Planet::print_5()</td>
<td></td>
</tr>
</tbody>
</table>

/* This virtual function has no implementation in the .cpp file. The presence of a pure virtual function makes the Sphere class an abstract class and we expect derived classes to implement these functions. */

// .h
virtual Sphere::print_5() = 0;

Q: Suppose we have print_6() where:
- print_6() is non-virtual in Sphere
- print_6() is virtual in Planet
...what happens?

Given a Planet polymorphed as a Sphere, the type of the object is a Sphere. Therefore, Sphere::print_6() will be examined. As it is non-virtual, "Sphere" will be printed and Planet:: will never be accessed.
Why?
Suppose you’re managing an animal shelter that adopts cats and dogs.

In order to write generic code, we would like to write code for our animal shelter in terms of Animals. By writing an Animal base class and derived classes Dog, Cat, Fox, etc, we can have specific instances of dogs and cats while writing all helper functions in terms of Animals.

Option 1 – No Inheritance  
Requires one function for every type of animal :

```cpp
animalShelter.cpp
1 Cat & AnimalShelter::adopt() { ... }  
2 Dog & AnimalShelter::adopt() { ... }  
3 ...
```

Option 2 – Inheritance  
Only a single function is needed to handle all Animals *yay!*

```cpp
animalShelter.cpp
1 Animal & AnimalShelter::adopt() { ... }
```

Pure Virtual Methods
In Sphere, print_5() is a pure virtual method:

```cpp
Sphere.h
1 virtual Sphere::print_5() = 0;
```

A pure virtual method does not have a definition and makes the class and abstract class.

Abstract Class:
1. [Requirement]: A class is an abstract class if it contains one or more pure virtual functions.
2. [Syntax]: No additional syntax is needed (eg: you do not need to write “abstract class”). Once you have a single pure virtual function, the class is abstract!
3. [As a result]: Once a pure virtual function exists, an instance of that class can no longer be created. The only instances of an abstract class must be polymorphed from another class (and must be either a reference or a pointer).

Abstract Class Animal
In our animal shelter, Animal is an abstract class if it contains one or more pure virtual function. Several pure virtual methods includes:

```cpp
virtual std::string speak() const = 0; // What does an animal say?  
// ...only specific animals speak.
std::string Dog::speak() const { return "woof"; }
std::string Cat::speak() const { return "meow"; }
std::string Fox::speak() const { return "???"; }
```

Virtual Destructor
In the following code, Sphere and Planet both have a destructor:

```cpp
Sphere.h | Planet.h
---|---
```
```cpp
class Sphere {  
    public:  
        ~Sphere();  
    ...  
};
```
```cpp
class Planet : public Sphere {  
    public:  
        ~Planet();  
    ...  
};
```

If a Planet has been polymorphed into a Sphere, the Planet destructor will never be called due to ~Sphere() being non-virtual! If Planet allocates any resources, these resources will never be freed because ~Planet() will never get called.

Therefore, base classes must always have a virtual destructor:

```cpp
Sphere.h | Planet.h
---|---
```
```cpp
class Sphere {  
    public:  
        virtual ~Sphere();  
    ...  
};
```
```cpp
class Planet : public Sphere {  
    public:  
        ~Planet();  
    ...  
};
```

Q: What about constructors – do they need to be virtual?
An object is always constructed as a certain type and then becomes another type through polymorphism. You may polymorph the object on the very first line of code, but the object is still constructed as the type you created. Therefore, the correct constructor is always called without the use of virtual.
Assignment Operator
I didn’t cover a few details of the assignment operator -- let’s do that:

1. [Default Assignment Operator]
C++ generate a default assignment operator for simple classes:
   - No non-static const variables
   - No reference variables

2. [Self-Assignment]
   - Programmers are never perfect and are never optimal.
   Consider the following:

   ```cpp
class Sphere {
    private:
    float radius;

    public:
    Sphere(float r) : radius(r) {}
    // other member functions...

    Sphere& operator=(const Sphere &other) {
        if (&other != this) {
            _destroy();  // destroy self
        }
        _copy(other);  // copy other to self
        return *this;  // return self
    }
};
```

   - A trivial implementation may:

   ```cpp
class Sphere {
    private:
    float radius;

    public:
    Sphere(float r) : radius(r) {}
    // other member functions...

    Sphere& operator=(const Sphere &other) {
        _destroy();  // destroy self
        _copy(other);  // copy other to self
        return *this;  // return self
    }
};
```

**Discussion:** When you’re copying yourself to yourself, destroying self also destroys other. Therefore, it’s critical to avoid this logic if are copying ourselves:

```cpp
class Sphere {
    private:
    float radius;

    public:
    Sphere(float r) : radius(r) {}
    // other member functions...

    Sphere& operator=(const Sphere &other) {
        if (&other != this) {
            _destroy();  // destroy self
        }
        _copy(other);  // copy other to self
        return *this;  // return self
    }
};
```