Polymorphism

OOP in C++

1. Pointers to Base Class

Lets revisit the example from *Friendship and Inheritance*. Instead of directly referencing the base class, we can use a pointer that points to the base class. This is known as polymorphism.

```cpp
#include <iostream>

class Polygon {
protected:
  int width, length;
public:
  void setValues(int a, int b){
    width = a; length = b;
  }
};

class Rectangle: public Polygon {
public:
  Rectangle(int a, int b): Polygon(a, b) {} //use Polygon constructor
  int area() {
    return width * length; //access protected members
  }
};

class Triangle: public Polygon {
public:
  Triangle(int a, init b) Polygon(a, b) {}
  int area() {
    return width * length / 2;
  }
};

int main() {
  Rectangle rect;
  Triangle tri;
  Polygon *poly1 = &rect;
  Polygon *poly2 = &tri;
  poly1->setValues(4, 5);
  poly2->setValues(4, 5);
  std::cout << rect.area() << std::endl;
  std::cout << tri.area() << std::endl;
  return 0;
}
```

Output:

```
20
10
```

We create two pointers to `Polygon`, `poly1` and `poly2` and assign them to the addresses of `rect` and `tri`. This is possible since `Rectangle` and `Triangle` are both derived from `Polygon`. It is important to note that this inheritance can only go in one direction; while `Rectangle` and `Triangle` have access to all non-private members of `Polygon`, `Polygon` cannot access any members of the derived classes. Thus, a statement such as `poly1->area()` would not work, since `area()` is not defined in `Polygon`.

2. Virtual Members

A virtual member is a a member function that can be redefined in a derived class. The syntax for defining a virtual function is the precede the declaration with the `virtual` keyword. As well, while it is not enforced, it is good practice to use the `override` keyword when overriding a virtual function. This helps prevent typos, as the compiler will raise an error if no matching `virtual` function is found in the base class.

```cpp
#include <iostream>

class Polygon {
protected:
  int width, length;
public:
  void setValues(int a, int b) {
    width = a; height = b;
  }
  virtual int area() {
    return 0;
  }
};

class Rectangle: public Polygon {
public:
  int area() override {
    return width * length;
  }
};

class Triangle: public Polygon {
public:
  int area() override {
    return width * length / 2;
  }
};

int main() {
  Rectangle rect;
  Triangle tri;
  Polygon poly;
  Polygon *polyPtr1 = &rect;
  Polygon *polyPtr2 = &tri;
  Polygon *polyPtr3 = &poly;
  polyPtr1->setValues(4, 5);
  polyPtr2->SetValues(4, 5);
  polyPtr3->SetValues(4, 5);
  std::cout << polyPtr1->area() << std::endl;
  std::cout << (*polyPtr2).area() << std::endl;
  std::cout << polyPtr3->area() << std::endl;
}
```

Output:

```
20
10
0
```

In this example, all three classes share a common member, `area`. In the base class, it was declared as `virtual`, therefore it can be redefined in the derived classes. Without this keyword, all three calls to `area` would just return `0`, since we would get `Polygon::area()`.

2.1. How does this even work??

Polymorphism is a bit like magic. We have three different implementations of the same function, `area()`, and they all have different behaviour. How does the compiler know which `area` to call?

At compile time, the compiler first performs a check to see if `Polygon` has a method called `area`. It doesn't choose which implementation to call, but instead places a "marker", saying to call the virtual function `area()` for the specified object through a *vtable*. During runtime, each class with a virtual function gets a vtable, which is a table of function pointers. Then, each object will get a vptr, pointing to its corresponding vtable. For example, `rect` has a vptr to Rectangle's vtable, containing `Rectangle::area()`. So, when we call `PolyPtr1->area()`, `PolyPtr1` points to a rectangle object, which points to a rectangle vtable, which stores the implementation of `Rectangle::area()`. This is known as *dynamic dispatch*: the decision of which function to call is not done at compile time, but rather dynamically at runtime.

3. Abstract Base Class

An abstract base class is a class that can only be used as a base class. In other words, we are unable to instantiate the class directly; instead, it has to be through derived classes. Due to this, we can create virtual functions that have no concrete implementation, known as a pure virtual function. This is done by replacing their definition with a `=0`. They are oftentimes thought of as *interfaces*; they dont have any concrete behaviour, but instead provide "frameworks" for other classes.

```
#include <iostream>
#include <string>

//Abstract base class
class Animal {
```

```cpp
protected:
  int height;
public:
  void setHeight(int a) {
    height = a;
  }
  virtual void sound(void) = 0; //pure virtual function, thus Animal cannot be
   ↪  instantiated directly.
};

class Dog: public Animal {
public:
  virtual void sound(void) override {
    std::cout << "Height: " << height << std::endl;
    std::cout << "A dog barks" << std::endl;
  }
};

class Person: public Animal {
public:
  virtual void sound(void) override {
    std::cout << "Height: " << height << std::endl;
    std::cout << "A person talks" << std::endl;
  }
};

int main() {
  Dog dog;
  Person person;
  Animal *ptr1 = &dog;
  Animal *ptr2 = &person;
  ptr1->setHeight(1);
  ptr2->setHeight(2);

  ptr1->sound();
  (*ptr2).sound();
  return 0;
}
```

Output:

```
Height: 1
A dog barks

Height: 2
A person talks
```

We are also able to implement functions within a base class that use pure virtual functions, even if there is no concrete implementation.

```cpp
#include <iostream>

class Polygon {
```

```cpp
protected:
  int width, length;
public:
  void setValues(int a, int b) { width = a; length = b; }
  virtual int area() = 0;
  void printArea() {
    std::cout << area() << std::endl;
  }
};

class Rectangle {
public:
  int area() {
    return width * length;
  }
};

class Triangle {
public:
  int area() {
    return width * length / 2;
  }
};

int main() {
  Rectangle rect;
  Triangle tri;
  Polygon *ptr1 = &rect;
  Polygon *ptr2 = &tri;
  ptr1->set_values(4, 5);
  ptr2->set_values(4, 5);
  ptr1->printArea();
  ptr2->printArea();
  return 0;
}
```

Output:

```
20
10
```