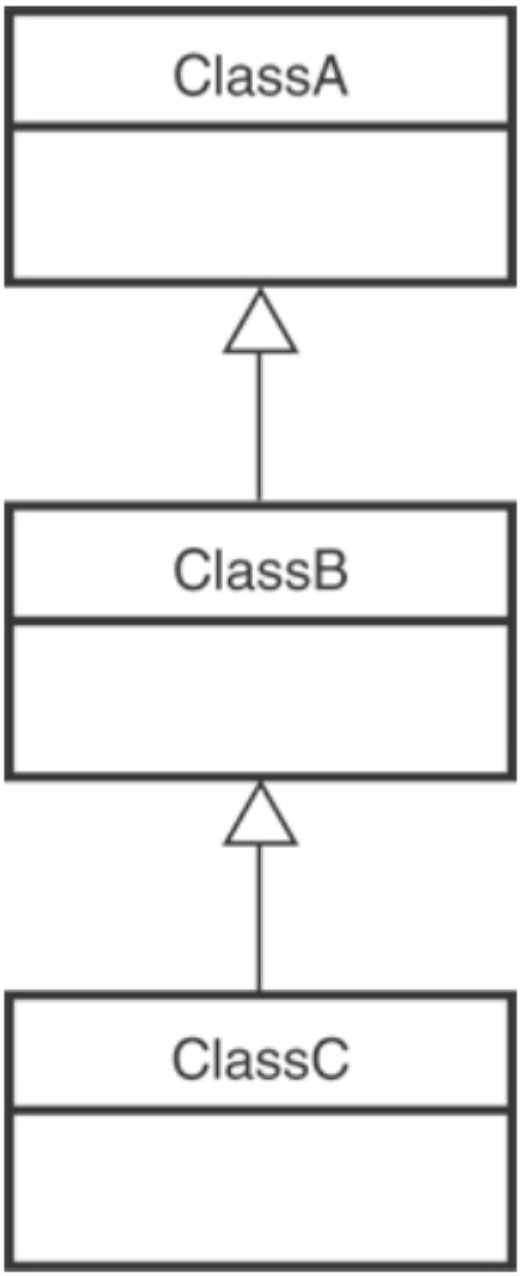
Polymorphism & Virtual Functions

This chapter introduces the concept of polymorphism and virtual functions, and their use in inheritance

We'll cover the following

Definitions

- A **virtual** function is a *member* function which is declared in the *base* class using the keyword `virtual` and is re-defined (*Overriden*) by the *derived* class.
- The term **Polymorphism** means the ability to take many forms. It occurs if there is a hierarchy of *classes* which are all related to each other by *inheritance*.



Class Hierarchy

> **Note:** In C++ what this means is that if we call a *member* function then it could cause a *different* function to be executed instead depending on what type of object invoked it.

Now, we'll look at an example using both these concepts to clarify your understanding.

Without Virtual Functions

```
#include <iostream>
using namespace std;

// Base class
class Shape {
   public:
      Shape(int l, int w){length = l; width=w;} //default constructor
      int get_Area(){cout << "This is call to parent class area"<<endl;}
  protected:
      int length,width;
};

// Derived class
class Square: public Shape {
   public:
      Square(int l=0, int w=0) : Shape(l,w) {} //declaring and initializing derived class construc
      int get_Area(){
        cout << "Square area: " << length*width << endl;
        return (length * width);
      }
};
// Derived class
class Rectangle: public Shape {
   public:
      Rectangle(int l=0, int w=0) : Shape(l,w) {} //declaring and initializing derived class const
      int get_Area(){ cout << "Rectangle area: " << length*width << endl;return (length * width);
};

int main(void) {
   Shape *s;
   Square sq(5,5); //making object of child class Sqaure
   Rectangle rec(4,5); //making object of child class Rectangle

   s = &sq;
   s->get_Area();
   s= &rec;
   s->get_Area();


   return 0;
}
```

In the above function,

- we store the address of each child class `Rectangle` and `Square` object in `s` and
- then we call the `get_Area()` function on it,
- ideally, it should have called the respective `get_Area()` functions of the child classes but
- instead it calls the `get_Area()` defined in the *base* class.
- This happens due **static linkage** which means the call to `get_Area()` is getting set only once by the compiler which is in the base class.

Using Virtual Functions

We can overcome **static linkage** problem by the use of `virtual` functions.

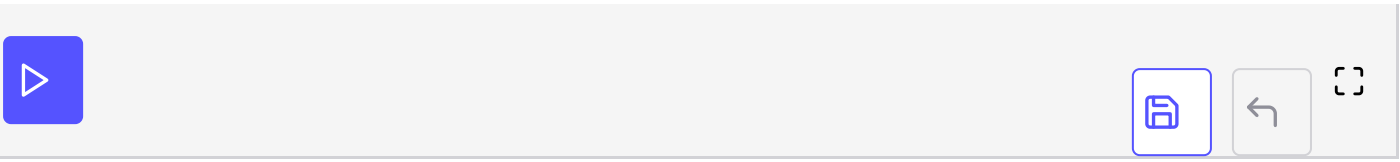
If we define a `virtual` function in the *base* class, with another version in a *derived* class, it will signal the compiler that we don't want **static linkage** for this function.

> **Note:** Using this allows for the selection of the function which we want to call at any given point to be based on the kind of object for which it is called.

Let's modify the code above by making use of this concept.

```
#include <iostream>
using namespace std;

// Base class
class Shape {
   public:
      Shape(int l, int w){length = l; width=w;} //default constructor
      //changing get_Area() to virtual type function
      virtual int get_Area(){cout << "This is call to parent class area"<<endl;}
  protected:
      int length,width;
};

// Derived class
class Square: public Shape {
   public:
      Square(int l=0, int w=0) : Shape(l,w) {} //declaring and initializing derived class construc
      int get_Area(){
        cout << "Square area: " << length*width << endl;
        return (length * width);
      }
};
// Dreived class
class Rectangle: public Shape {
   public:
      Rectangle(int l=0, int w=0) : Shape(l,w) {} //declaring and initializing derived class const
      int get_Area(){ cout << "Rectangle area: " << length*width << endl;return (length * width);
};

int main(void) {
   Shape *s;
   Square sq(5,5); //making object of child class Sqaure
   Rectangle rec(4,5); //making object of child class Rectangle

   s = &sq;
   s->get_Area();
   s= &rec;
   s->get_Area();


   return 0;
```

```
}
```



Now when *addresses* of objects of `Rectangle` and `Square` classes are stored in `*s` the respective `get_Area()` function is called since this time, the compiler looked at the contents of the *pointer* rather than its *type*.

This marks the end of the chapter on *classes and inheritance*. Next up we'll discuss *templates* in C++.