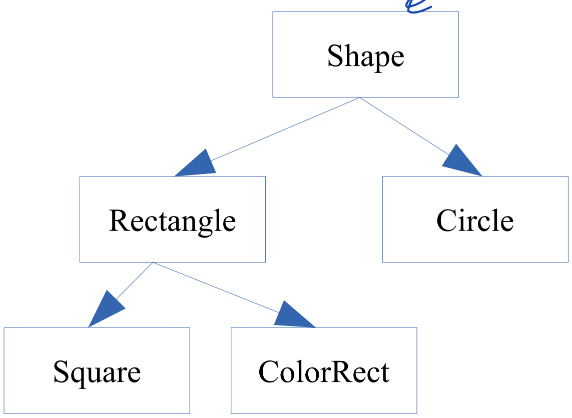
CS 315 – Programming Languages
Implementation of Dynamic Binding

Dynamic (late) binding is the fundamental technique behind polymorphism. The method calls made on a polymorphic reference are resolved at runtime. The appropriate method from the class of the object that is currently being pointed by a polymorphic reference is used.

A typical example:

Ellie



Assume Spahe is an abstract class that defines a getArea() method and the other classes override it as appropriate to provide implementations. Consider the following code:

```
double computeArea(ArrayList<Shape> shapes) {
     double sum = 0.0;
     for (Shape s : shapes)
          sum += s.getArea();
     return sum;
}
```

How is the call s.getArea() resolved? At compile time, the compiler should generate code for this call, but it does not know which class of object is being pointed by s. What is worse, at each iteration, it could be a different class of object. Yet, we need to generate code for a call that should work for all.

The idea is to generate an indirect call.:

- As part of each object, a *virtual table pointer* is kept.
- The virtual table pointer points to a table of function descriptors (aka the *virtual table*)
- The call is made by following the virtual table pointer, finding the appropriate function from the virtual table, and making a call to it.
- There is one virtual table per non-abstract class (not one per object).

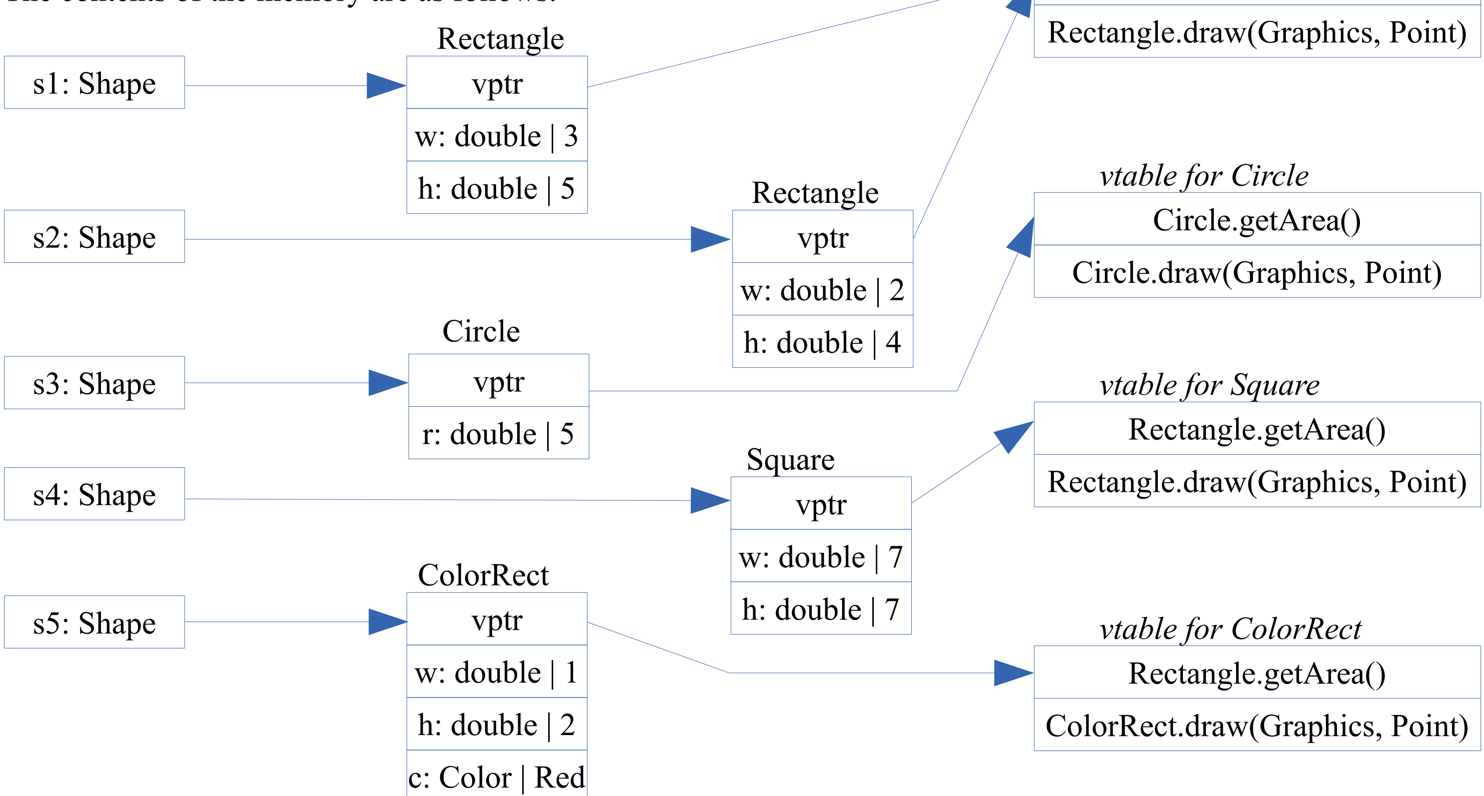
Consider the following example:

```
public abstract Shape {
      public abstract double getArea();
      public abstract void draw(Graphics g, Point p);
}
public class Circle extends Shape {
      protected double r;
      public Circle (double r) { this.r = r; }
      public double getArea() { return Math.PI * r * r; }
      public void draw(Graphics g, Point p) { … }
}
public class Rectangle extends Shape {
      protected double w;
      protected double h;
      public Rectangle(double w, double h) { this.w = w; this.h = h; }
      public double getArea() { return r * w; }
      public void draw(Graphics g, Point p) { … }
}
public class Square extends Rectangle {
      public Square(double s) { super(s, s); }
}
public class ColorRect extends Rectangle {
      protected Color c;
      public RoundRect(double w, double h, Color c)
            { super(w, h); this.c = c; }
      public void draw(Graphics g, Point p) { … }
}
```

And consider the following lines of code:

```
Shape s1 = new Rectangle(3, 5);
Shape s2 = new Rectangle(2, 4);
Shape s3 = new Circle(5);
Shape s4 = new Square(7);
Shape s5 = new ColorRect(1, 2, Color.Red);
```

The contents of the memory are as follows:



In C++, dynamic binding is optional. Member functions that are **virtual** use dynamic binding, whereas those that are not do not. By default, dynamic binding is not used.

A simple example illustrating this:

```
class A {
public:
      void foo() {}
      virtual void bar() {}
};
class B : public A {
public:
      void foo() {}
      void bar() {} // note: virtual keyword is optional when overriding
};

B b;
A* a = &b;
a->foo(); // calls A::foo (no dynamic binding, type of *a is used)
a->bar(); // calls B::bar (dynamic binding)
```

Now let us consider the following complete C++ example:

```
#include <vector>

class LivingBeing {
protected:
      int birtDate;
public:
      LivingBeing(int birthDate) : birthData(birthDate) {}
      virtual ~LivingBeing() {}
      int getBirthDate() const { return birthDate; }
      virtual void move(int dx, int dy) = 0; // abstract method
};

class Person : public LivingBeing {
public:
      enum Gender { Male, Female };
protected:
      Gender gender;
public:
      Person(int birthDate, Gender gender)
            : LivingBeing(birthDate), gender(gender) {}
      Gender getGender() const { return gender; }
      void move(int dx, int dy) { … }
};

class Musician : public Person {
public:
      enum Kind { Soloist, Guitarist, Flutist };
protected:
      Kind kind;
public:
      Musician(int birthDate, Gender gender, Kind kind)
            : Person(birthDate, gender), kind(kind) {}
      Kind getKind() const { return kind; }
      virtual void sing() { … }
};
```

```
using namespace std;

int main() {
  LivingBeing* lb1 = new Person(1923, Person::Male);
  LivingBeing* lb2 = new Person(1930, Person::Female);
  LivingBeing* lb3 = new Musician(1928, Person::Male, Musician::Flutist);

  vector<LivingBeing*> livings = { lb1, lb2, lb3 };
  for (LivingBeing* lb : livings)
      lb->move(3, -4);

  ...
}
```

Let us draw the contents of the memory assuming vector keeps an integer to store the size and an array to store data.