Creative Software Design

Polymorphism 2

Yunho Kim
yunhokim@hanyang.ac.kr
Dept. of Computer Science

Today's Topics

- Pure Virtual Function
- The Practical Power of Polymorphism
- Some Issues about Virtual Functions
- Abstract Class / Pure Abstract Class
- Type Casting Operators
- Virtual Inheritance & Multiple Inheritance
- Behind Virtual Functions

Review: Virtual Functions

- Virtual functions are keys to implement polymorphism in C++.
  - Declare polymorphic member functions to be ‘virtual’, and
  - Use the base class pointer to point an instance of the derived class, then
  - The function call from a base class pointer will execute the function overridden in the derived class.

CSStudent Example with Virtual Functions

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

class Person
{
public:
    virtual void talk()
    {
        cout << "I'm a person" << endl;
    }
};

class Student : public Person
{
public:
    virtual void talk()
    {
        cout << "I'm a student" << endl;
    }
    void study()
    {
        cout << "study" << endl;
    }
};
```

```cpp
class CSStudent : public Student
{
public:
    virtual void talk()
    {
        cout << "I'm a CS student" <<
endl;
    }
    void writeCode()
    {
        cout << "writeCode" << endl;
    }
};

int main()
{
    CSStudent csst;
    csst.talk(); //"I'm a CS student"

    Person& asPerson = csst;
    asPerson.talk(); //"I'm a CS student"

    return 0;
}
```

CSStudent Example w/o Virtual Functions

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

class Person
{
public:
    void talk()
    {
        cout << "I'm a person" << endl;
    }
};

class Student : public Person
{
public:
    void talk()
    {
        cout << "I'm a student" << endl;
    }
    void study()
    {
        cout << "study" << endl;
    }
};
```

```
class CSStudent : public Student
{
public:
    void talk()
    {
        cout << "I'm a CS student" <<
endl;
    }
    void writeCode()
    {
        cout << "writeCode" << endl;
    }
};

int main()
{
    CSStudent csst;
    csst.talk(); //"I'm a CS student"

    Person& asPerson = csst;
    asPerson.talk(); //"I'm a person"

    return 0;
}
```

Pure Virtual Function

- What if you cannot define the base class' member function? (no 'default' behavior)

```
#include <vector>
#include <iostream>
using namespace std;
class Shape {
public:
  virtual void Draw() const {
    // Nothing to do here
  }
};

class Rectangle : public Shape {
public:
  virtual void Draw() const {
    cout << "rect" << endl;
  }
};

class Triangle : public Shape {
  // What if we forget to override
  // Draw() here?
};
```

```
int main() {
  vector<Shape*> v;
  v.push_back(new Rectangle);
  v.push_back(new Triangle);

  for (size_t i = 0; i < v.size(); ++i) {
    v[i]->Draw();
  }
  for (size_t i = 0; i < v.size(); ++i) {
    delete v[i];
  }
  return 0;
}
```

Pure Virtual Function

- In such cases, use *pure virtual functions*
  - Just declare it ending with '= 0'

```
class Shape {
public:
  // Pure virtual Draw function.
  virtual void Draw() const = 0;
};
```

Pure Virtual Function

- A class with pure virtual functions cannot be instantiated.
  – A pure virtual function typically has no definition.
- Its subclass should override pure virtual functions to be instantiated. Or you'll see a compile error.

```cpp
#include <vector>
#include <iostream>
using namespace std;
class Shape {
public:
  virtual void Draw() const = 0;
};

class Rectangle : public Shape {
public:
  virtual void Draw() const {
    cout << "rect" << endl;
  }
};

class Triangle : public Shape {
  // What if we forget to override
  // Draw() here? => Error!
};
```

```cpp
int main() {
  vector<Shape*> v;
  v.push_back(new Rectangle);
  v.push_back(new Triangle);

  for (size_t i = 0; i < v.size(); ++i) {
    v[i]->Draw();
  }
  for (size_t i = 0; i < v.size(); ++i) {
    delete v[i];
  }
  return 0;
}
```

Pure Virtual Function

- Just provides *“interface to do something”* in a base class.
  - "What to do"

- *“How to do it”* is implemented in the definition of each overridden virtual function in derived classes.

- FYI, a pure virtual function (C++ term) is often called an *abstract method* in other programming languages (Java, Python, ...).

The Practical Power of (Subtype) Polymorphism

- When coding *type-specific details*, it allows you to avoid using if...else or switch statements which are often error-prone.

- With polymorphism...
  - It’s easier to add a new type (just adding a new subclass without touching the existing class code).
  - Each type-specific implementations are isolated from each other (in different classes).
  - It does not allow an exceptional case with an unexpected type.
  - It removes duplicate if...else or switch statements.

```
class Animal
{
public:
    AnimalType type;

    virtual string talk() {
        switch(type) {
        case CAT: return "Meow!";
        case DOG: return "Woof!";
        case DUCK: return "Quack!";
        case PIG: return "Oink!";
        default:
            assert(0);
            return string();
        }
    }

    virtual int getNumLegs() {
        switch(type) {
        case CAT: return 4;
        case DOG: return 4;
        case DUCK: return 2;
        case PIG: return 4;
        default:
            assert(0);
            return -1;
        }
    }

    virtual void walk() {
        switch(type) {
        case CAT:
            ...
            break;
        case DOG:
            ...
            break;
        case DUCK:
            ...
```

```
class Animal
{
public:
    virtual string talk() = 0;
    virtual int getNumLegs() = 0;
    virtual void walk() = 0;
};

class Cat : public Animal
{
public:
    virtual string talk() { return "Meow!"; }
    virtual int getNumLegs() { return 4; }
    virtual void walk() {...}
};

class Dog : public Animal
{
public:
    virtual string talk() { return "Woof!"; }
    virtual int getNumLegs() { return 4; }
    virtual void walk() {...}
};

class Duck : public Animal
{
public:
    virtual string talk() { return "Quack!"; }
    virtual int getNumLegs() { return 2; }
    virtual void walk() {...}
};

class Pig : public Animal
{
public:
    virtual string talk() { return "Oink!"; }
    virtual int getNumLegs() { return 4; }
    virtual void walk() {...}
};
```

Quiz #1

- What is the expected output? (including compile/runtime error)

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

class Base
{
public:
    virtual void show() = 0;
};

int main(void)
{
    Base b;
    Base *bp;
    return 0;
}
```

Some Issues with Virtual Functions

- You may have heard that virtual functions have some disadvantages.
  - More memory: an object of a class with virtual functions has an additional member, a *vptr*
  - Slower speed: pointer indirection to call functions, limited possibilities to be inlined or optimized

Some Issues with Virtual Functions

- But, when coding *type-specific details*, these issues are too tiny to matter.

- Because replacing virtual function calls with if...else or switch
  - has disadvantages described in “The Practical Power of (Subtype) Polymorphism” page.
  - and might be even slower.

Some Issues with Virtual Functions

- But if your classes are not designed to be inherited,
- Actually, there is no reason to use virtual functions.
  - It's better to avoid (slightly) more memory and (slightly) slower speed in this case.
- From C++11, you can use a keyword ‘final’ to prevent a class from being a super class
  - The keyword also can be used for a member function to prevent overriding

Abstract Class

- An *abstract class* is a class that **cannot be instantiated**
  - a.k.a. *abstract base class*
  - A class that can be instantiated is called *concrete class*
- In C++, a class **with one or more pure virtual functions** is an abstract class.
  - Its subclass must implement all the pure virtual functions to be instantiated (or itself become an abstract class)

```
class Shape {
public:
  virtual void Draw() const = 0;
};

int main() {
  Shape shape; // error! cannot be instantiated!
  return 0;
}
```

Constructors in Abstract Classes

- Do we need to define a constructor for an abstract class? An abstract class will never be instantiated!

- Yes! You should still create a constructor to initialize its members, since they will be inherited by its subclass.

```
class Animal
{
private:
    string name;
public:
    Animal(const string& name_):name(name_) {}
    virtual string talk() = 0;
    virtual int getNumLegs() = 0;
    virtual void walk() = 0;
};

class Cat : public Animal
{
public:
    Cat(const string& name_):Animal(name_) {}
    virtual string talk() { return "Meow!"; }
    virtual int getNumLegs() = { return 4; }
    virtual void walk() {...};
};

class Dog : public Animal
{
public:
    Dog(const string& name_):Animal(name_) {}
    virtual string talk() { return "Woof!"; }
    virtual int getNumLegs() = { return 4; }
    virtual void walk() {...};
};
```

Destructors in Abstract Classes

- Then do we need to define a destructor for an abstract class?

- Yes! An abstract class SHOULD have a virtual destructor even if it does nothing.

Destructors in Abstract Classes

- An abstract class SHOULD have a virtual destructor even if it does nothing.

- Recall that:
- A destructor of a *base* class **should be** `virtual` if
  - its descendant class instance is **deleted by the base class pointer.**

  (..or)
  - any of member function is **virtual** (which means it's a polymorphic base class).

- An abstract class
  - has at least one pure **virtual function.**
  - is designed to be used as "base class reference(or pointer)".

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

class Shape
{
public:
    Shape() {}
    virtual ~Shape() {}
    virtual void draw() = 0;
};

class Rectangle : public Shape
{
private:
    int* width;
    int* height;
public:
    Rectangle()
    {
        width = new int;
        height = new int;
    }
    virtual ~Rectangle()
    {
        delete width;
        delete height;
    }
    virtual void draw()
    { ... }
};
```

```cpp
int main()
{
    Shape* shape1 = new Rectangle;
    shape1->draw();
    delete shape1;

    return 0;
}
```

Pure Abstract Class

- A class that has **only pure virtual functions**
  - No member variables or non-pure-virtual functions (except destructor)
  - Defines an **interface** to a service - “What does the class do”, “How it should be used”
  - “How to do it” should be implemented in derived concrete classes
- In general, a pure abstract class is used to define an interface and is intended to be inherited by concrete classes.

```
class Shape {
public:
  virtual ~Shape() {}
  virtual void Draw() const = 0;
  virtual int GetArea() const = 0;
  virtual void MoveTo(int x, int y) = 0;
};

void DrawShapes(const vector<Shape*>& v) {
  for (int i = 0; i < v.size(); ++i) v[i]->Draw();
}
```

Quiz #2

- Why do we need to define a constructor and destructor for an abstract class which cannot be instantiated?

Type Casting Operators in C

- C-style casting operator: `(T)var` (or `T(var)`)

- Problems:
  - Programmer's intention is not clear
  - No type checking (unsafe)
  - Not easy to search (C/C++ code has a very large number of parentheses!)

Type Casting Operators in C++

- C++ casting operators
  - `static_cast<T>(var)`
  - `dynamic_cast<T>(ptr)`
  - `const_cast<T>(ptr)`
  - `reinterpret_cast<T>(ptr)`

- Each operator is designed to be used for specific purpose.

static_cast

- `static_cast<T>` performs type checking at *compile time*.
  - If T is a pointer or reference type:
    - Safe for upcast (derived -> base)
    - Unsafe for downcast (base -> derived)
      - It’s the programmer's responsibility to make sure that *base class pointer* is actually pointing to a specified *derived class object.*
  - If T is a primitive type:

```
int i = static_cast<int>(2.0);
```

can be used for casting between primitive types

static_cast

```
class B {};

class D : public B
{
public:
    int member_D;
    void test_D() { member_D=10; }
};

class X {};

int main() {
   B b; D d; char ch; int i=65;
   B* pb = &b; D* pd = &d;

   D* pd2 = static_cast<D*>(pb);   // Unsafe. If you access pd2's members not
                                   // in B, you get a run time error.
   pd2->test_D();                 // Runtime error!

   B* pb2 = static_cast<B*>(pd);    // Safe, D always contains all of B.

   X* px = static_cast<X*>(pd);   // Compile error!

   ch = static_cast<char>(i);    // int to char
}
```

dynamic_cast

- `dynamic_cast<T>` performs type checking at *run time*.
  - Safe for downcast
    - If *base class pointer* is **not** pointing to a specified *derived class object*, `dynamic_cast` of base to derived pointer returns null pointer (0).
  - Note that `dynamic_cast` can only downcast polymorphic types (base class should have at least one virtual function).

dynamic_cast

```
#include <iostream>

class B
{
public:
    virtual ~B() {}
};

class D : public B
{
public:
    void test_D() { std::cout << "test_D()" << std::endl; }
};

int main() {
   B b; D d;

   B* pb = &b;
   //B* pb = &d;

   D* pd2 = dynamic_cast<D*>(pb);
   if(pd2)
       pd2->test_D();
}
```

const_cast, reinterpret_cast

- `const_cast<T*>` removes 'const' from `const T* ptr`
- `reinterpret_cast` is just like C-style cast; avoid using it.

```
class B {};
class X {};

int main() {
    B b;
    B* pb = &b;

    const B* cpb = pb;
    B* pb2 = const_cast<B*>(cpb);

    X* px = reinterpret_cast<X*>(pb);
}
```

Notes for C++ Casting Operators

- Hard to type! (too many characters!)
- Actually, they are *ugly by design.*
  - "Maybe, *because static_cast* is so *ugly and* so relatively hard *to type,* you're more likely to think twice before using one? That would be good, because casts really are mostly avoidable in modern C++.

  \- Bjarne Stroustrup (C++ creator)
  http://www.stroustrup.com/bs_faq2.html#static-cast
- Avoid casting as far as possible. Prefer polymorphism.

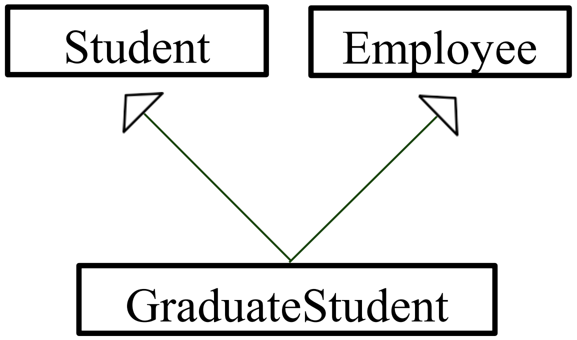
Quiz #3

- What is the expected output? (including compile/runtime error)

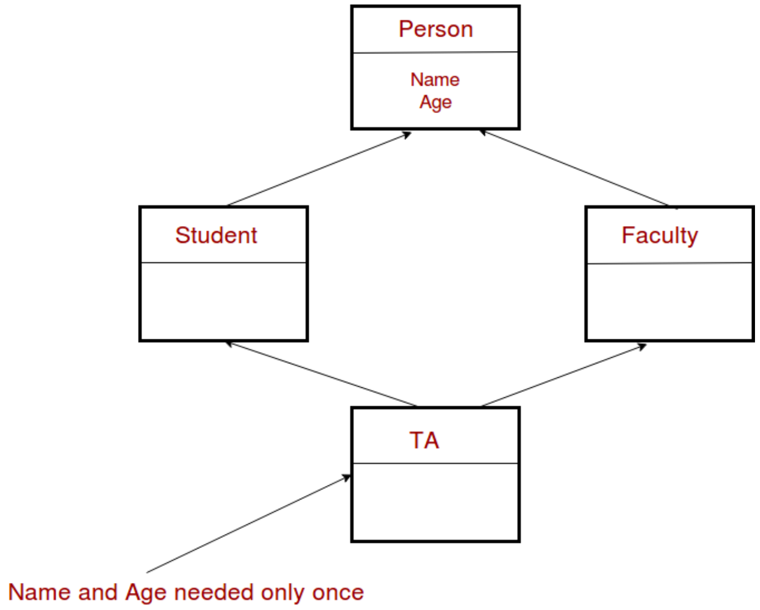
```
#include <iostream>
using namespace std;

class A
{
public:
    virtual int getA(){ return 1; }
};

class B : public A
{
public:
    virtual int getA() { return 2; }
    int getB() { return 3; }
};

int main()
{
    A a;
    B b;
    B* b1 = static_cast<B*>(&a);

    cout << "b.getA(): " << b.getA() << endl;
    cout << "b.getB(): " << b.getB() << endl;
    cout << "b1->getA(): " << b1->getA() << endl;
    cout << "b1->getB(): " << b1->getB() << endl;
}
```

Recall: Multiple Inheritance

- Inheriting from two or more base classes.
  - The derived class has all the members of base classes



- Issues
  - Ambiguity
    - What happens if base classes has same- named members?
  - The diamond problem
    - What happens if parent classes are derived from the same grandparent class?



The Diamond Problem Example

```
// Person class.

class Person {
  int age;
  string name;
 public:
  Person(const string& name);
  const string &getName() const {
    return name;
  }
…
};

// Student class.

class Student : public Person {
 public:
  Student(const string& name);
…
};
```

```
// Faculty class

class Faculty : public Person {
 public:
  Faculty(const string& name, int salary);
…
};

// TA class

class TA : public Student, public Faculty
{
 public:
  Employee(const string& name, int salary);
…
};

int main(){
  TA ta;
  cout << ta.getName(); // Compile error
…}
```

Recall Multiple Inheritance

- Actually, you can avoid these problem by using `virtual` inheritance in C++.

- General advice: Avoid using multiple inheritance as much as possible.
  - It is commonly believed that multiple inheritance tends to mass things up.
  - That's why Java forbids multiple inheritance.

- Note that multiple inheritance from *interfaces* (pure abstract classes in C++) can be very helpful.
  - Java only allows multiple inheritance from *interfaces* (“implements” multiple interfaces in Java)

The Diamond Problem Example

```
// Person class.

class Person {
  int age;
  string name;
 public:
  Person(const string& name);
  const string &getName() const {
    return name;
  }
…
};

// Student class.

class Student : virtual public Person {
 public:
  Student(const string& name);
…
};
```

```
// Faculty class

class Faculty : virtual public Person {
 public:
  Faculty(const string& name, int salary);
…
};

// TA class

class TA : public Student, public Faculty
{
 public:
  Employee(const string& name, int salary);
…
};

int main(){
  TA ta;
  cout << ta.getName(); // OK
…}
```

Behind Virtual Functions

- How do virtual functions work internally in C++?
- → It depends on complier implementation. The C++ standard only specifies the behavior of virtual functions.

- But most compilers use *virtual method table* (a.k.a. ***vtable***) mechanism.

Memory Layout of C++ Object

```
class Shape
{
public:
    Shape();
    double getArea();
    double getPerimeter();
private:
    Vector2D position;
    Color outline, fill;
};
```

```
int main()
{
    Shape s1;
    Shape* s2 = new Shape;
    delete s2;
    return 0;
}
```

s1

fill
outline
position

***s2**

fill
outline
position

Memory Layout of C++ Object

```
class Shape
{
public:
    Shape();
    double getArea();
    double getPerimeter();
private:
    Vector2D position;
    Color outline, fill;
};
```

```
int main()
{
    Shape s1;
    Shape* s2 = new Shape;
    double a = s2->getArea();
    delete s2;
    return 0;
}
```

s1

fill
outline
position

***s2**

fill
outline
position

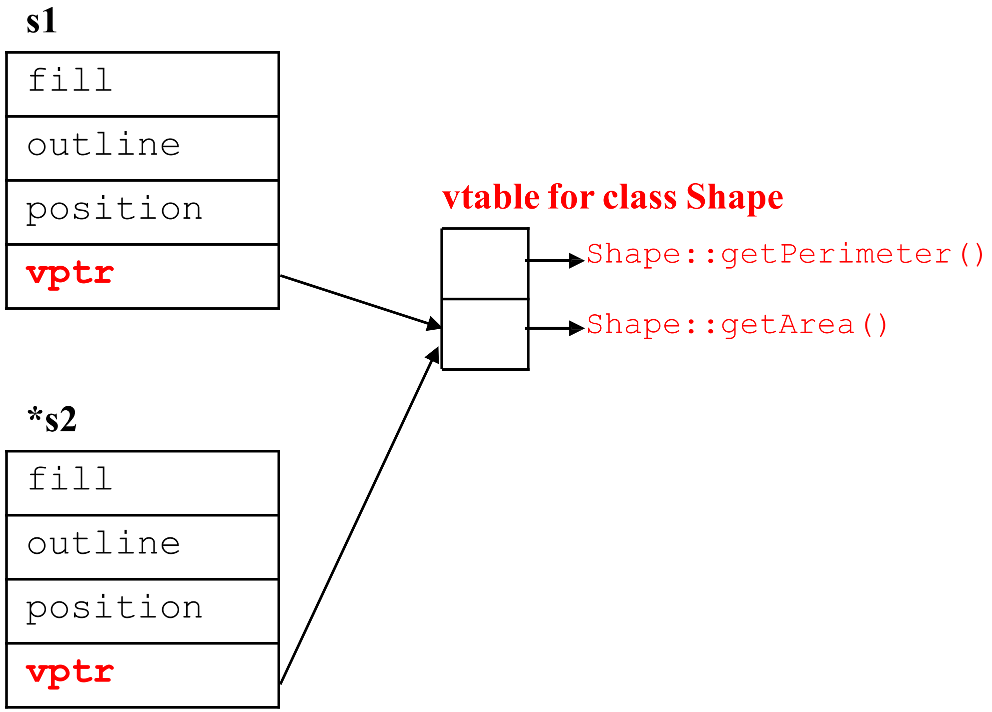
jumps to → `Shape::getArea()` (in code segment)

(Static binding)

Memory Layout of C++ Object

```
class Shape
{
public:
    Shape();
    virtual double getArea();
    virtual double getPerimeter();
private:
    Vector2D position;
    Color outline, fill;
};
```

```
int main()
{
    Shape s1;
    Shape* s2 = new Shape;
    delete s2;
    return 0;
}
```

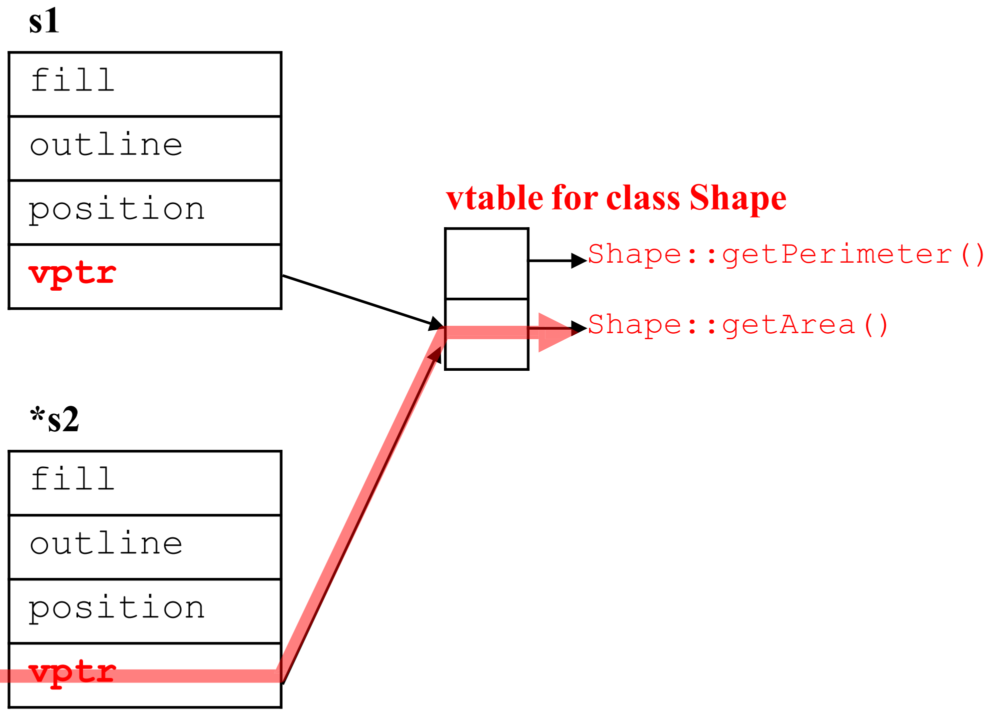


- ***vtable*** is a lookup table that contains the addresses of the object's dynamically bound virtual functions.
- ***vtable*** is created only for **classes with at least one virtual function.**
- ***vptr*** is created as a "hidden" member of **each instance of these classes** and initialized to point to the ***vtable*** **of the actual type of the instance.**

Memory Layout of C++ Object

```
class Shape
{
public:
    Shape();
    virtual double getArea();
    virtual double getPerimeter();
private:
    Vector2D position;
    Color outline, fill;
};
```

```
int main()
{
    Shape s1;
    Shape* s2 = new Shape;
    double a = s2->getArea();
    delete s2;
    return 0;
}
```



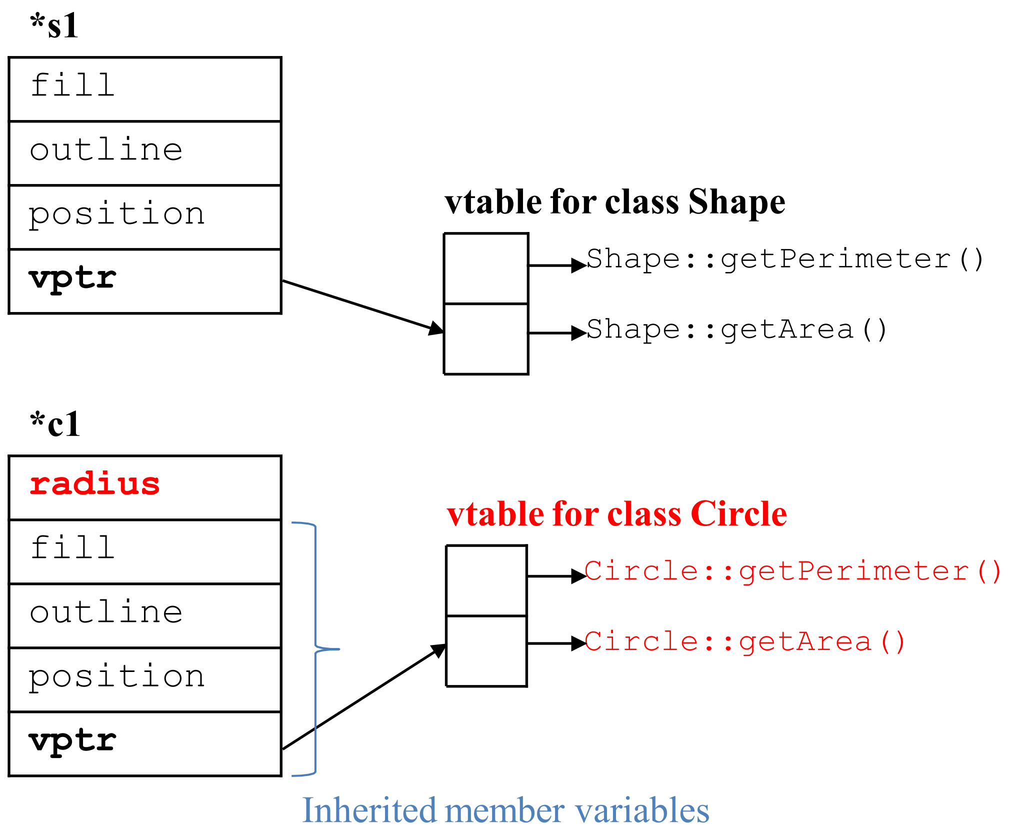
jumps to

(Dynamic binding)

Memory Layout of C++ Object

```
class Shape
{
public:
    Shape();
    virtual double getArea();
    virtual double getPerimeter();
private:
    Vector2D position;
    Color outline, fill;
};

class Circle: public Shape
{
public:
    Circle(double r);
    virtual double getArea();
    virtual double getPerimeter();
private:
    double radius;
};
```

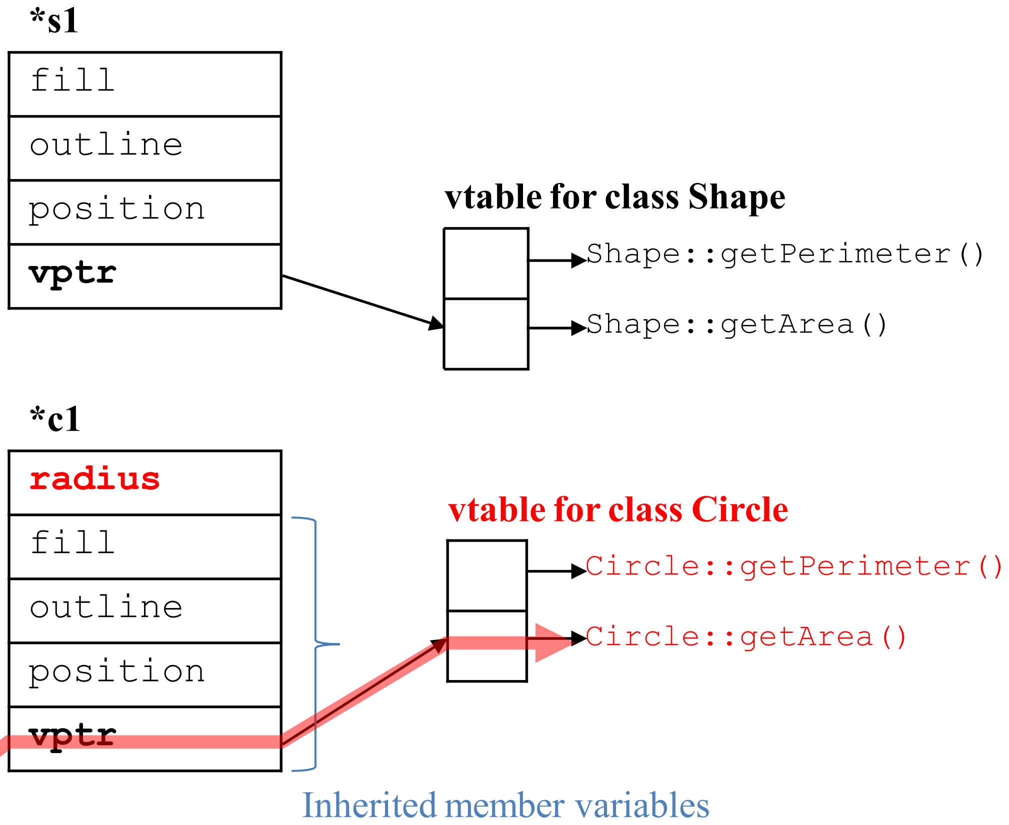
```
int main()
{
    Shape* s1 = new Shape;
    Shape* c1 = new Circle;
    return 0;
}
```



Memory Layout of C++ Object

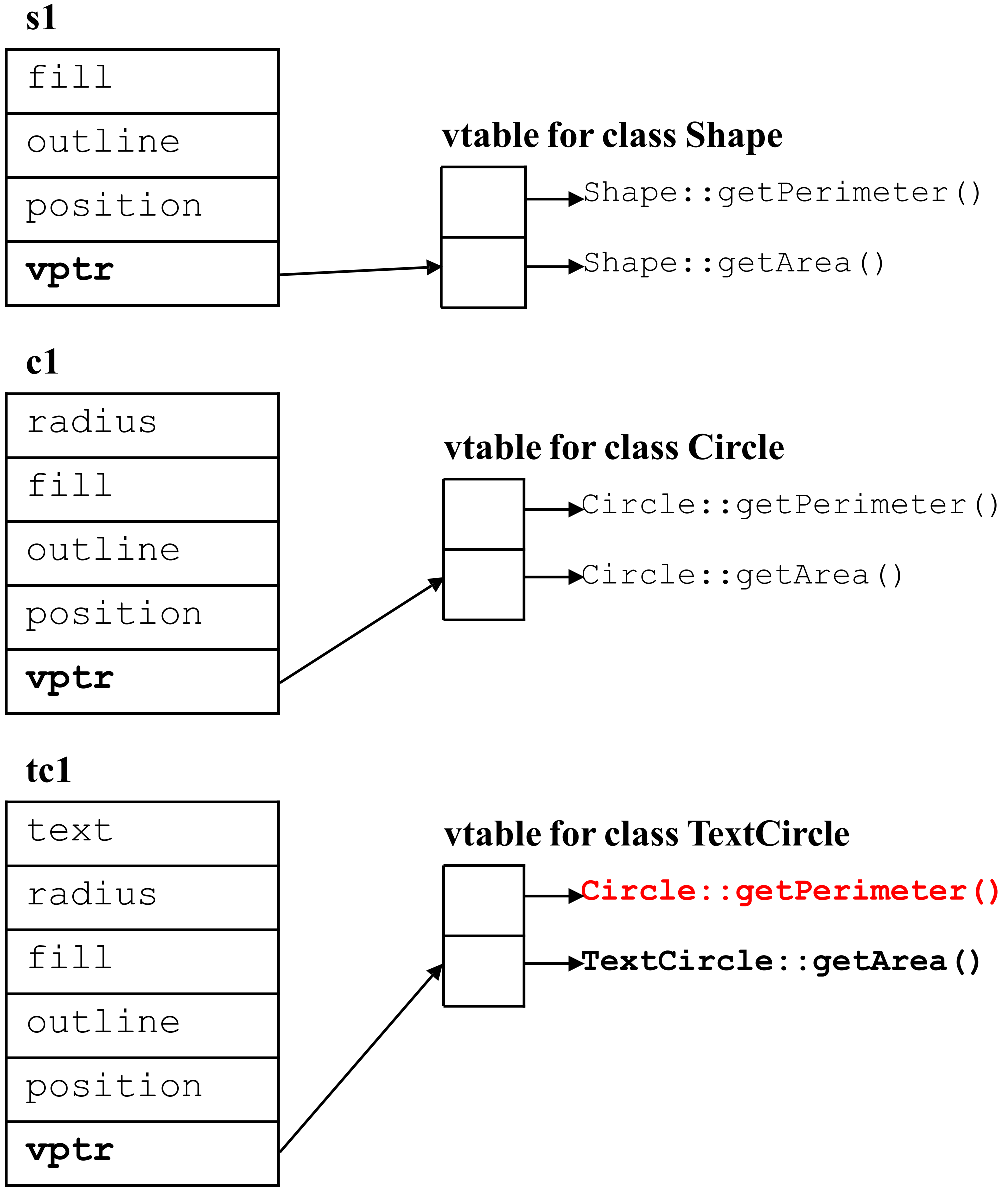
```
class Shape
{
public:
    Shape();
    virtual double getArea();
    virtual double getPerimeter();
private:
    Vector2D position;
    Color outline, fill;
};

class Circle: public Shape
{
public:
    Circle(double r);
    virtual double getArea();
    virtual double getPerimeter();
private:
    double radius;
};
```

```
int main()
{
    Shape* s1 = new Shape;
    Shape* c1 = new Circle;
    c1->getArea();
    return 0;
```



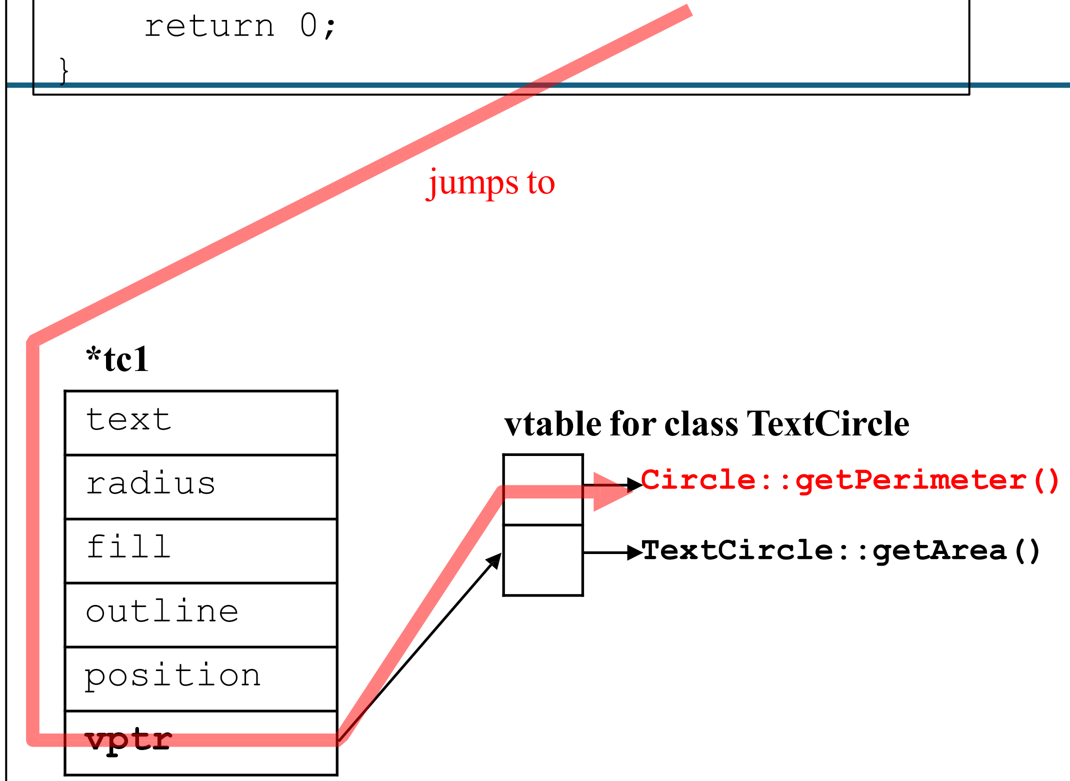
```
class Shape
{
public:
    Shape();
    virtual double getArea();
    virtual double getPerimeter();
private:
    Vector2D position;
    Color outline, fill;
};

class Circle: public Shape
{
public:
    Circle(double r);
    virtual double getArea();
    virtual double getPerimeter();
private:
    double radius;
};

class TextCircle: public Circle
{
public:
    TextCircle(string s);
    virtual double getArea();
private:
    string text;
};
```

```
int main(){
    Shape s1; Circle c1; TextCircle tc1;
    return 0;
}
```



```
class Shape
{
public:
    Shape();
    virtual double getArea();
    virtual double getPerimeter();
private:
    Vector2D position;
    Color outline, fill;
};

class Circle: public Shape
{
public:
    Circle(double r);
    virtual double getArea();
    virtual double getPerimeter();
private:
    double radius;
};

class TextCircle: public Circle
{
public:
    TextCircle(string s);
    virtual double getArea();
private:
    string text;
};
```

```
int main(){
    Shape* tc1 = new TextCircle;
    double p = tc1->getPerimeter();
    return 0;
}
```



Behind Virtual Functions

- ***vtable*** is created only for **classes with at least one virtual function (a.k.a. *polymorphic classes*)**, generally at compile time.
  - It is a lookup table that contains the addresses of the object’s dynamically bound virtual functions.

- ***vptr*** is created & initialized at runtime, when a *polymorphic class* instance is constructed.
  - created as a “hidden” member of the instance.
  - initialized to point to the *vtable* of the actual type of the instance.
  - The actual name of vtpr depends on the compiler: __vtpr, __vfptr, ...

Behind Virtual Functions

- Compiling non-virtual function calls:
  – Compiler generates code to call (jump to the address of) the non-virtual function.

- Compiling virtual function calls:
  – Compiler generates code to go through *vptr* to find *vtable* and call a certain entry of the *vtable* (the index for each function is known at compile time).
  – Which *vtable* is pointed by *vptr* is determined at run time (when an object is constructed).