Creative Software Programming

10 – Polymorphism 2

Today's Topics

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

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

```
#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;
  }
};
```

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

  void WriteCode() {
    cout << "write_code" << endl;
  }
};

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

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

  return 0;
}
```

CSStudent Example with 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 << "write_code" << endl;
  }
};

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

  Person& as_person = csst;
  as_person.Talk();  // "I'm a person"

  return 0;
}
```

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

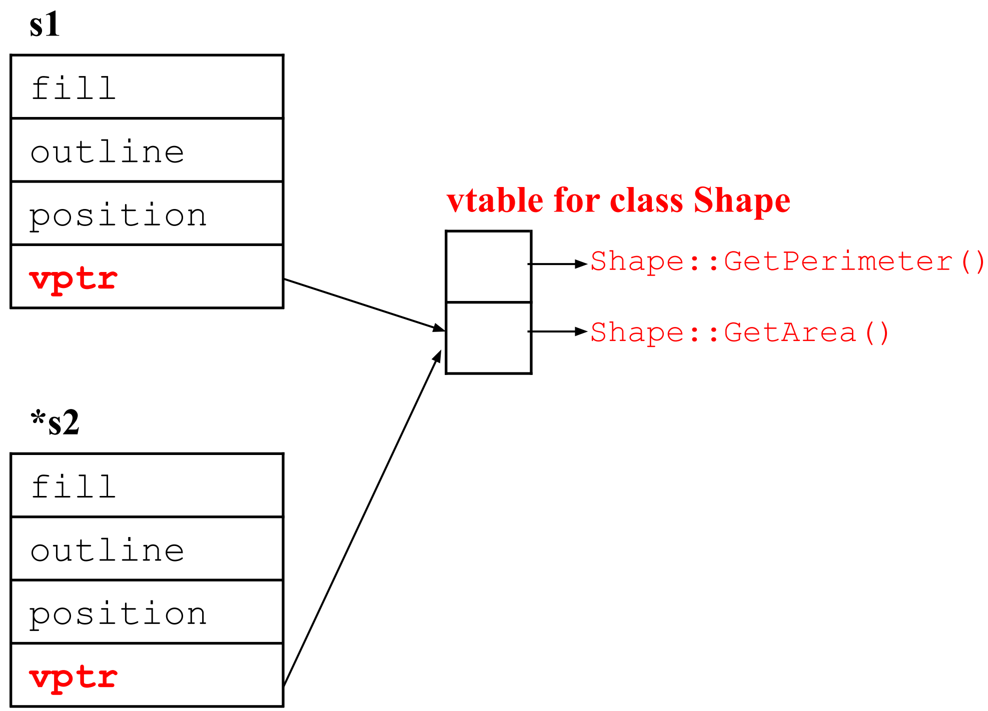
jumps to

Shape::GetArea() (in code segment)

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;
}
```

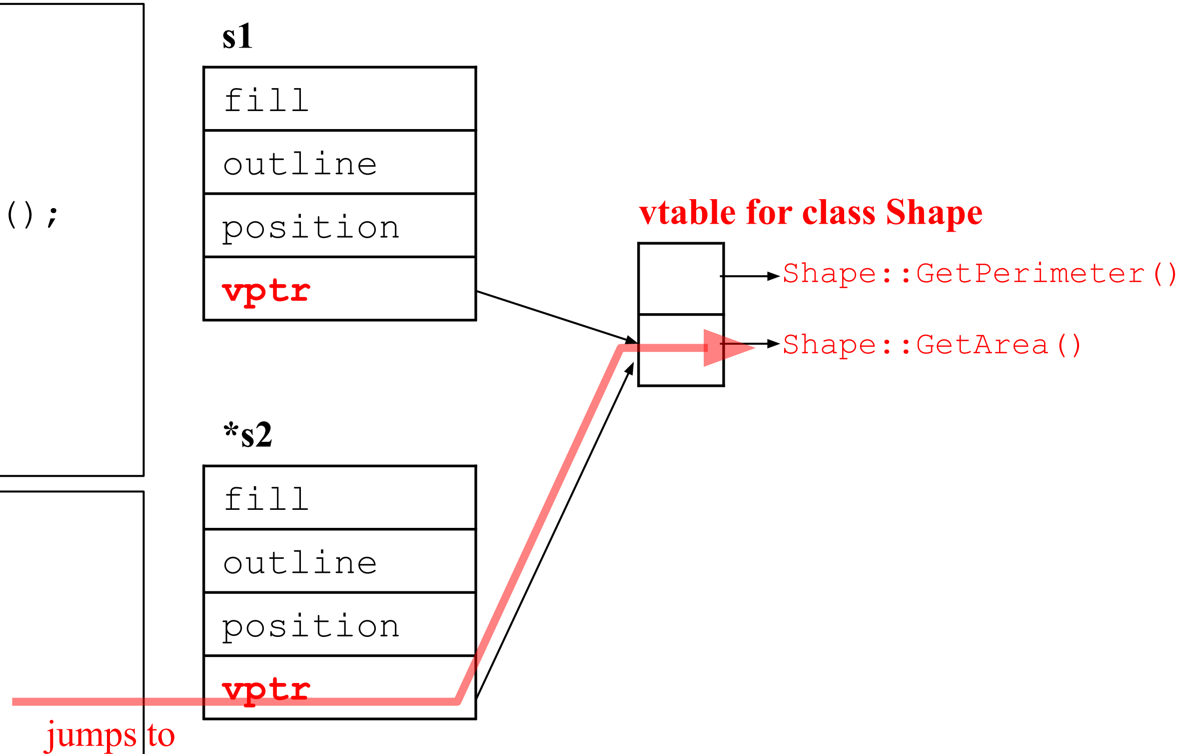


- ***vtable*** is created only for **classes with at least one virtual function.**
- It is a lookup table that contains the addresses of the object's dynamically bound virtual functions.
- ***vptr*** is created as a “hidden” member of **each instance of these classes** and initialized to point to the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

Memory Layout of C++ Object

```
class Shape {
 public:
  Shape();
  virtual double GetArea();
  virtual double GetPerimeter();

 private:
  Vector2D position;
  Color outline, fill;
};
```

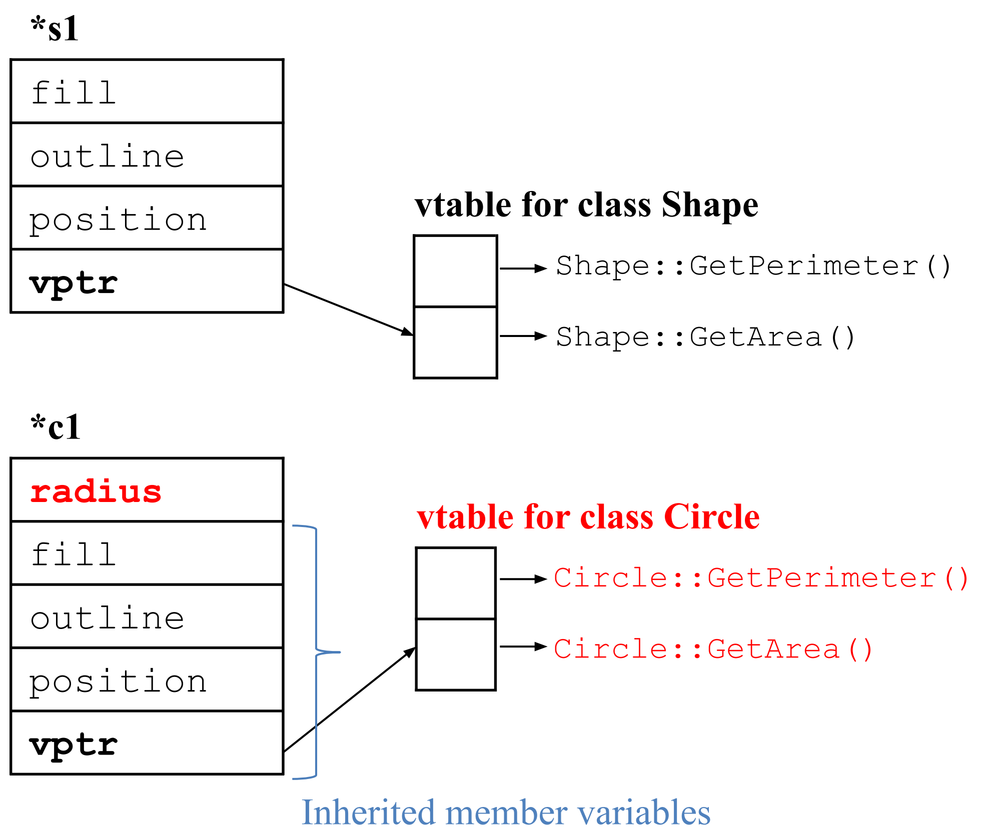
```
int main() {
  Shape s1;
  Shape* s2 = new Shape;

  double a = s2->GetArea();
  delete s2;
  return 0;
}
```



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;
};
```

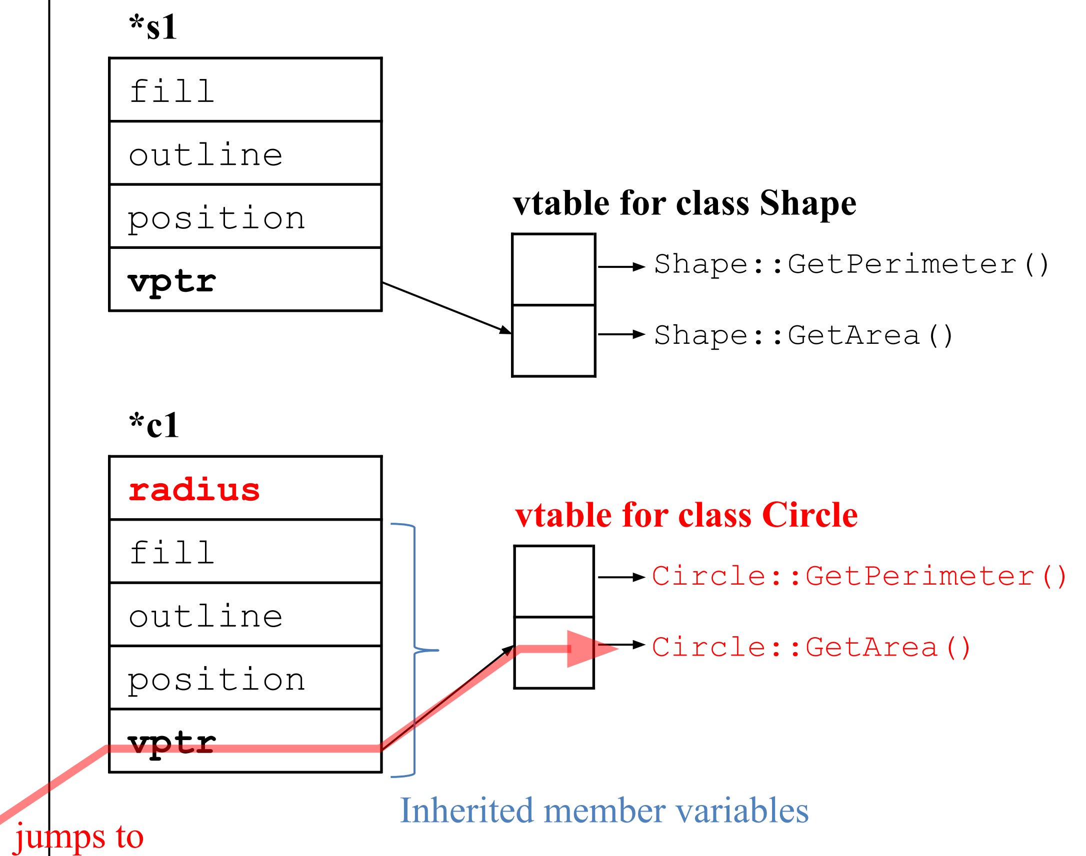
```
int main() {
  Shape* s1 = new Shape;
  Shape* c1 = new Circle;
  return 0;
}
```



Memory Layout of C++ Object

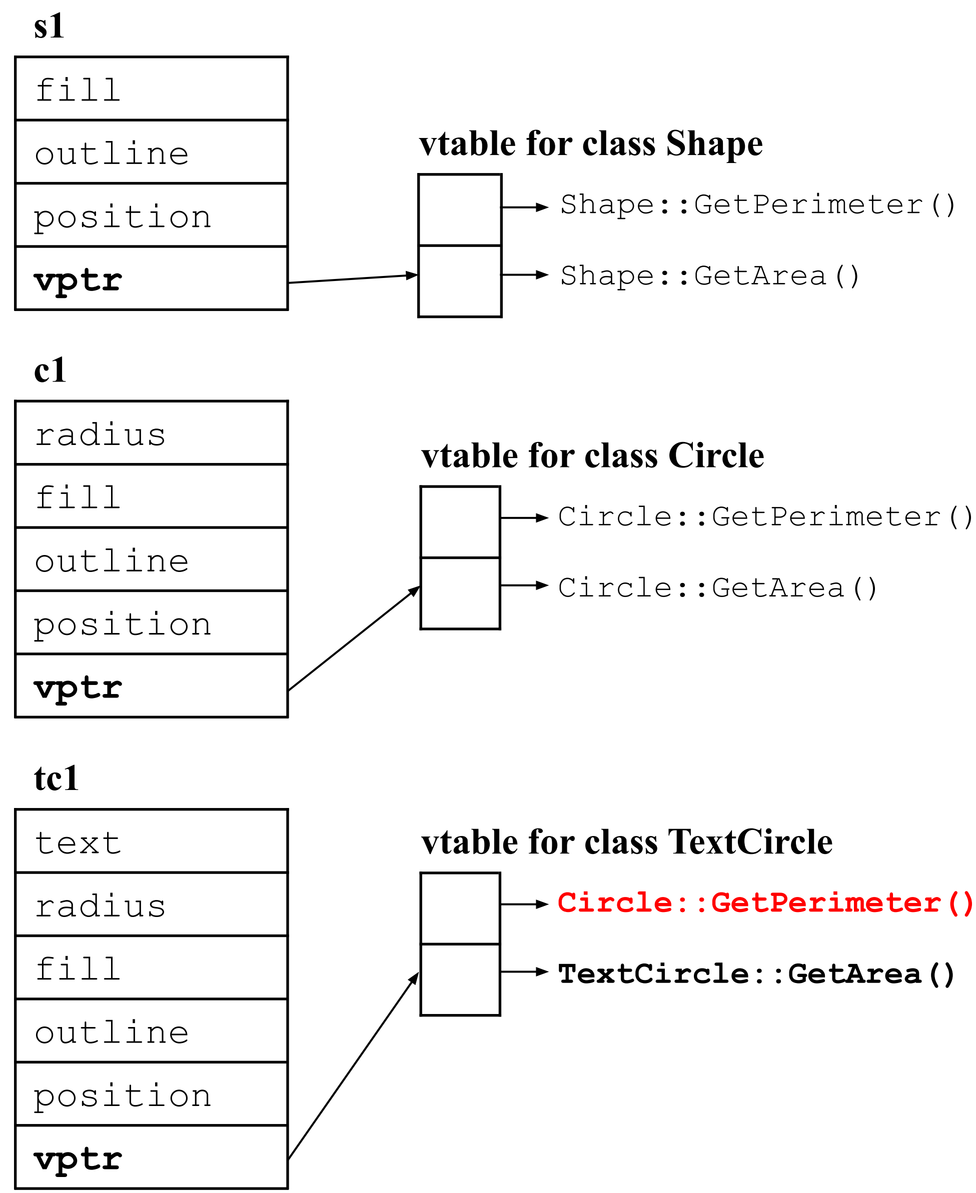
```
class Shape {
 public:
  Shape();
  virtual double GetArea();
  virtual double GetPerimeter();

 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;
}
```



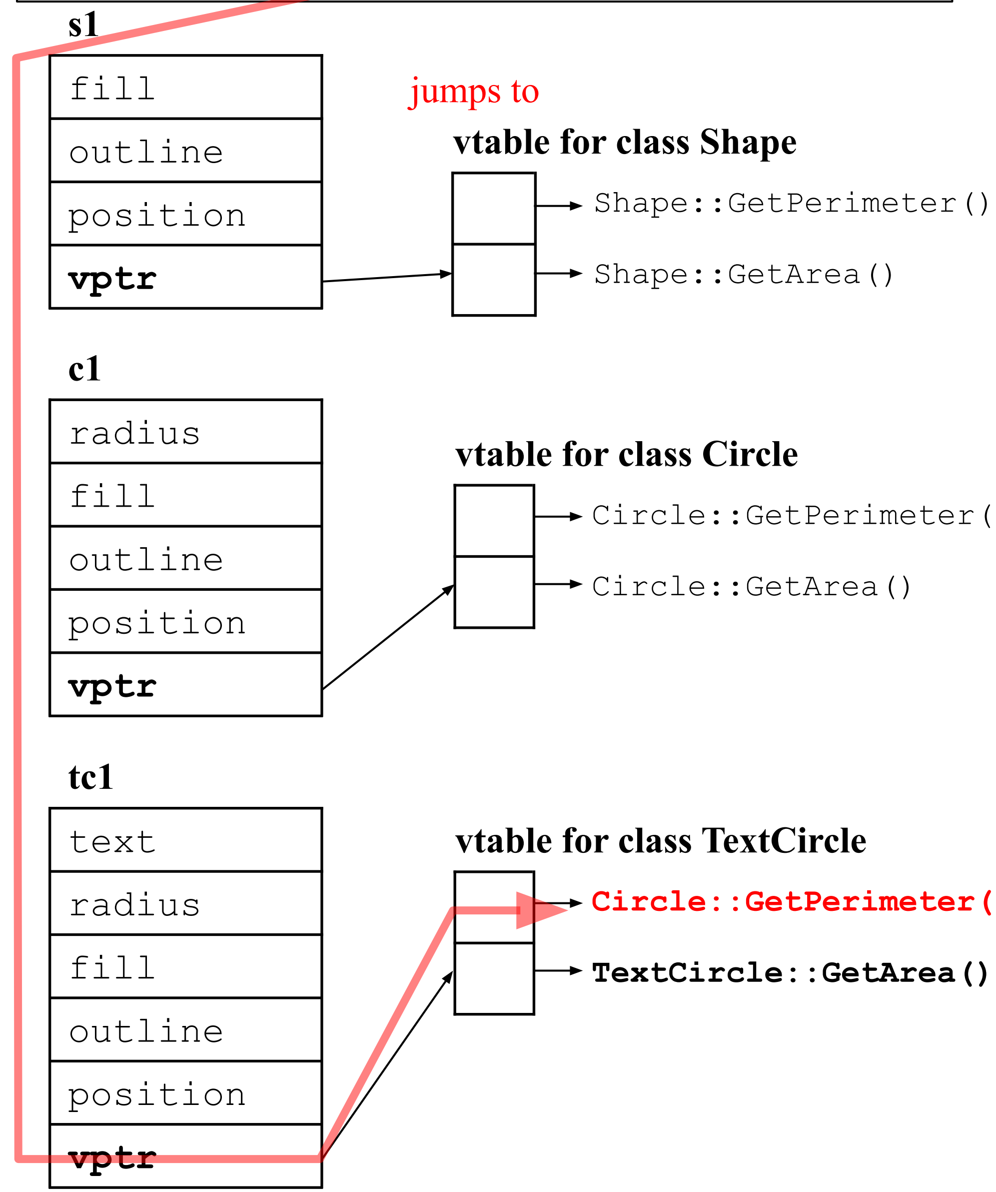
```
class Shape {
 public:
  Shape();
  virtual double GetArea();
  virtual double GetPerimeter();

 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;
}
```



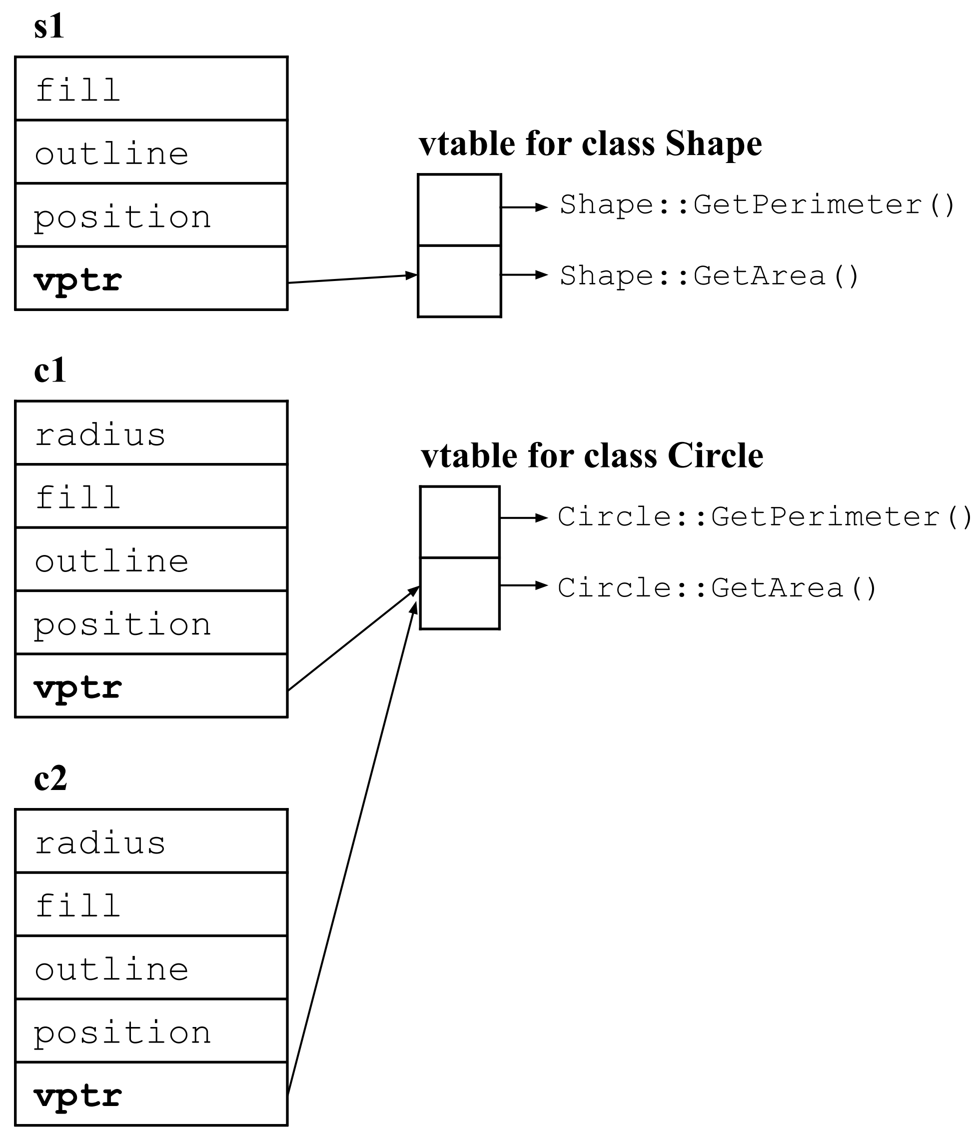
```
class Shape {
 public:
  Shape();
  virtual double GetArea();
  virtual double GetPerimeter();

 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 s1; Circle c1; TextCircle tc1;
  double p = tc1->GetPerimeter();
  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; Circle c2;
  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 *polymorphic class* instances are constructed.
  - created as a “hidden” member of the instances.
  - initialized to point to the *vtable* of the actual type of the instances.

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).

Quiz #1

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

class Person {
 public:
  virtual void Talk();
  virtual void GetName();
  void SayHi();
};

class Student : public Person {
 public:
  virtual void Talk();
  virtual void Study();
};

class CSStudent : public Student {
 public:
  virtual void Talk();
  virtual void WriteCode();
};
```

- List all the functions whose addresses are in the **vtable of the CSStudent class**.

Pure Virtual Function

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

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

struct Shape {
  virtual void Draw() const {
    // What should we do here?
  }
};

struct Rectangle : public Shape {
  virtual void Draw() const {
    cout << "rect" << endl; // Draw a
rectangle.
  }
};

struct 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’

```
#include <vector>
#include <iostream>
using namespace std;
struct Shape {
  // Pure virtual Draw function.
  virtual void Draw() const = 0;
};
```

Pure Virtual Function

- Pure virtual functions...
  - Cannot have definitions.
  - Should be overridden to be instantiated. Or you'll see a compile error.

```
#include <vector>
#include <iostream>
using namespace std;
struct Shape {
  // Pure virtual Draw function.
  virtual void Draw() const = 0;
};

struct Rectangle : public Shape {
  virtual void Draw() const {
    cout << "rect" << endl; // Draw a
rectangle.
  }
};

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

```
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.

- A pure virtual function (C++ term) is often called an *abstract method* in other programming languages (java, python, ...).

The Power of (Subtype) Polymorphism

- Allows you to avoid using if...else or switch statements to code *type-specific details*, 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;

  string Talk() {
    switch(type) {
    case CAT: return "Meow!";
    case DOG: return "Woof!";
    case DUCK: return "Quack!";
    default: assert(0); return string();
    }
  }
  int GetNumLegs() {
    switch(type) {
      case CAT: return 4;
      case DOG: return 4;
      case DUCK: return 2;
      default: assert(0); return -1;
    }
  }
  void Walk() {
    switch(type) {
      case CAT: ...
        break;
      case DOG: ...
        break;
      case DUCK: ...
        break;
      default: assert(0); break;
    }
  }
};
```

```
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() { ... }
};
```

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

- If you're going to code some type-specific details using virtual functions with inheritance (i.e. using polymorphism), these issues are too tiny to matter.

- Because replacing virtual function calls with if...else or switch
  - has disadvantages described in "The 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,
- It's better not use virtual functions to avoid using more memory and slower speed.

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 the all of the pure virtual functions to be instantiated (or itself become an abstract class)

```
struct Shape {
  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!

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?

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* size;

 public:
  Rectangle() {
    size = new int[2];
  }

  virtual ~Rectangle() {
    delete[] size;
  }

  virtual void Draw() {
   ...
  }
};
```

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

  return 0;
}
```

Pure Abstract Class (a.k.a. Interface Class)

- A class **only with 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.

```
struct Shape {
  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

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

class Animal {
 public:
  virtual string Talk() = 0;
  virtual int GetNumLegs() = 0;
};

class Cat : public Animal {
 public:
  virtual string Talk() { return "Meow!"; }
  virtual int GetNumLegs() { return 4; }
};

class Dog : public Animal {
 public:
  virtual int GetNumLegs() { return 4; }
};
```

```
int main() {
  vector<Animal*> animals;
  animals.push_back(new Cat);
  animals.push_back(new Dog);

  for (int i = 0; i < animals.size(); ++i) {
    cout << animals[i]->GetNumLegs() << endl;
  }
  return 0;
}
```

- What is the expected output of this program? (If a compile error is expected, just write down "error").

Type Casting Operators in C

- C-style casting operator: `(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` performs type checking at *compile time*.
  - 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*.
  - Can be used for casting between primitive types

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

static_cast

```
class B {};

class D : public B {
 public:
  int member_d;
  void TestD() { member_d = 10; }
};

class X {};

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

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

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

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

  char ch; int i = 65;
  ch = static_cast<char>(i);  // int to char
}
```

dynamic_cast

- `dynamic_cast` 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(const char* s) { std::cout << s << std::endl; }
};

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

  D* pd2 = dynamic_cast<D*>(pb);
  if (pd2) pd2->Test("b -> b -> d");

  pb = &d;
  pd2 = dynamic_cast<D*>(pb);
  if (pd2) pd2->Test("d -> b -> d");

  return -1;
}
```

const_cast, reinterpret_cast

- `const_cast` 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);
}
```

Quiz #3

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

class Animal {
 public:
  virtual string Talk() { return "..."; }
  virtual int GetNumLegs() { return 0; }
};

class Cat : public Animal {
 public:
  virtual string Talk() { return "Meow!"; }
  virtual int TetNumLegs() { return 4; }
};

int main() {
  Animal* pa = new Animal;
  Cat* pd = dynamic_cast<Cat*>(pa);
  if (pd) {
    cout << pd->GetNumLegs() << endl;
  } else {
    cout << pa->GetNumLegs() << endl;
  }
  return 0;
}
```

- What is the expected output of this program? (If a compile error is expected, just write down "error").

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.

Next Time

- Next lecture:
  - 11 – Copy Constructor, Operator Overloading