
Creative Software Design

10 – Polymorphism 2

Yoonsang Lee

Fall 2022

Today's Topics

- Behind Virtual Functions
- Pure Virtual Function
- The Practical 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 / reference to refer an instance of the derived class,
 - then the function call from a base class pointer / reference 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 << "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;
}
```

Behind Virtual Functions

- How do virtual functions work internally in C++?
- → It depends on compiler 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

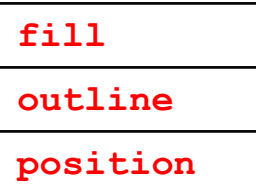
position

Memory Layout of C++ Object

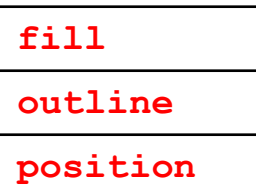
```
class Shape
{
public:
    Shape();
    double getArea();
    double getPerimeter();
private:
    Vector2D position;
    Color outline, fill;
};
```

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

s1



*s2



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

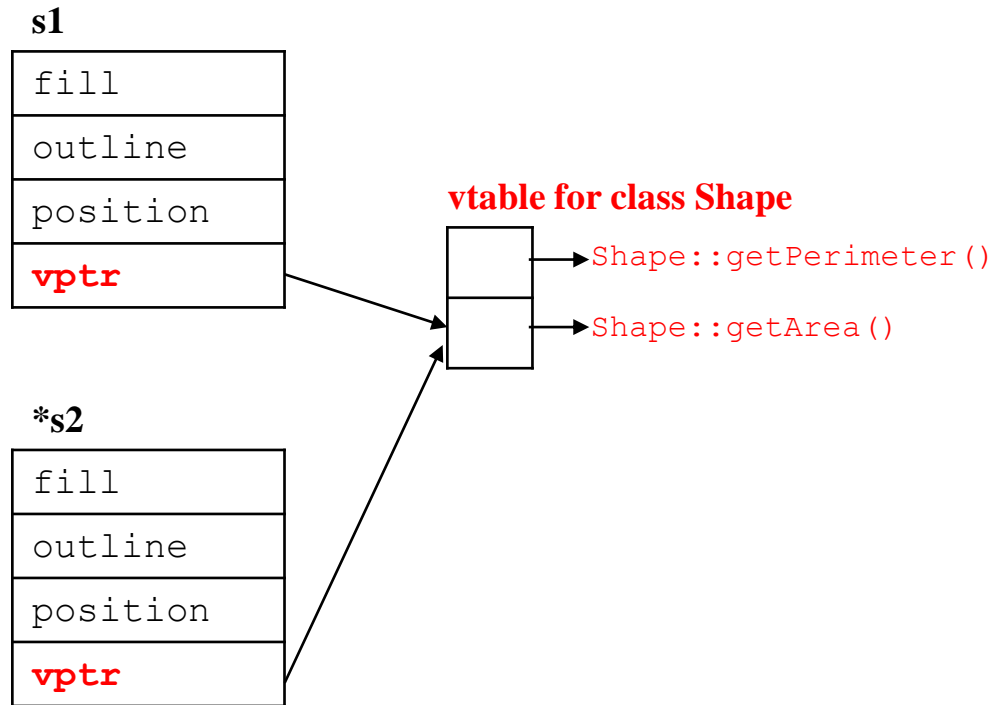
Static binding

- Compiler generates code to call (jump to the address of) Shape::getArea() directly.

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;
}
```

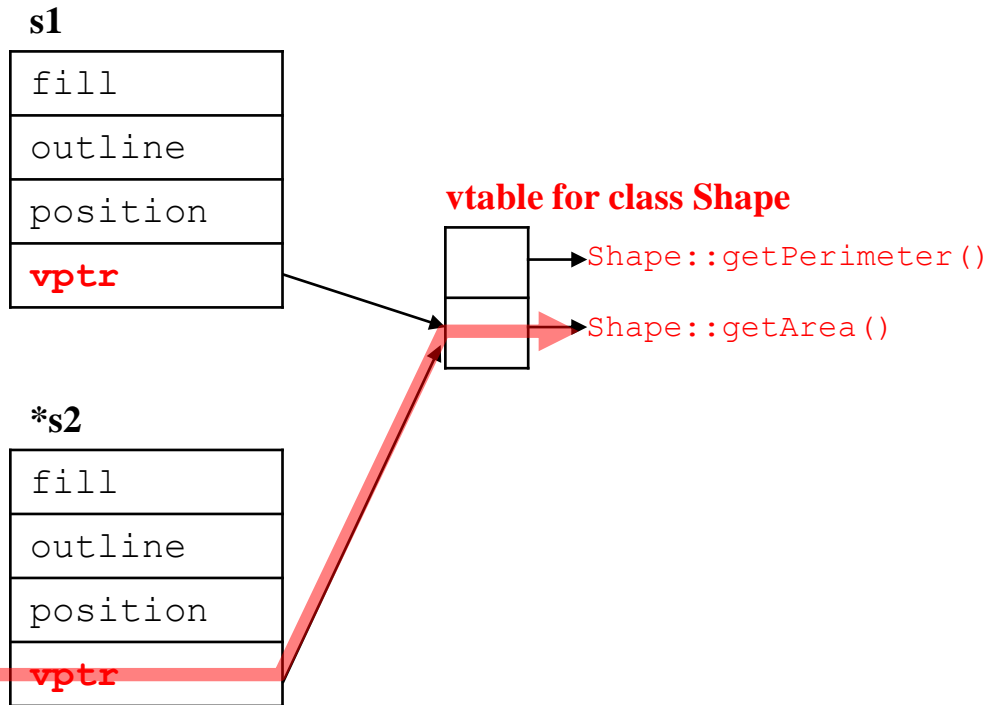


- ***vtable*** is a lookup table that contains the addresses of the object's dynamically bound virtual functions.
- ***vtable*** is created only for **classes with at least one virtual function**.
- ***vptr*** is created as a “hidden” member of **each instance of these classes** and initialized to point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



Dynamic binding

- Compiler generates code to call (jump to) the 'getArea' entry (index 0 in this example) of the vtable through the vptr.

Memory Layout of C++ Object

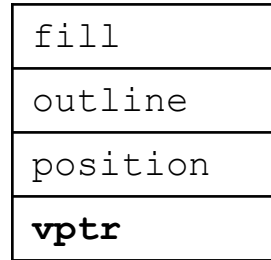
```
class Shape
{
public:
    Shape();
    virtual double getArea();
    virtual double getPerimeter();
private:
    Vector2D position;
    Color outline, fill;
};
```

```
class Circle: public Shape
```

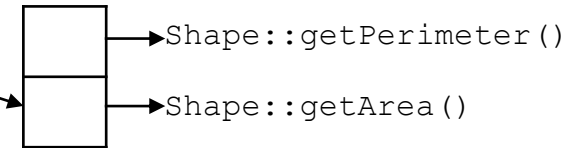
```
{
public:
    Circle(double r);
    virtual double getArea();
    double getPerimeter();
private:
    double radius;
};
```

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

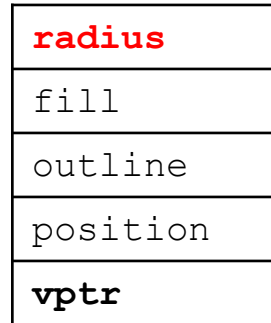
***s1**



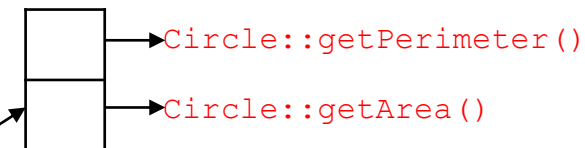
vtable for class Shape



***c1**



vtable for class Circle



Inherited member variables

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
```

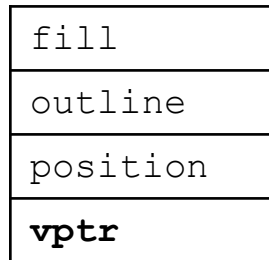
```
{
public:
    Circle(double r);
    virtual double getArea();
    double getPerimeter();
private:
    double radius;
};
```

```
int main()
```

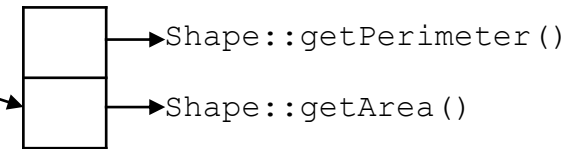
```
{
    Shape* s1 = new Shape;
    Shape* c1 = new Circle;
    c1->getArea();
    return 0;
```

jumps to

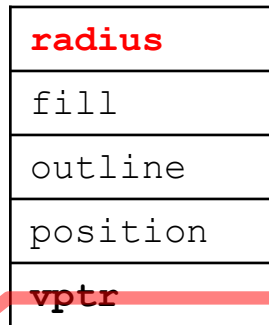
***s1**



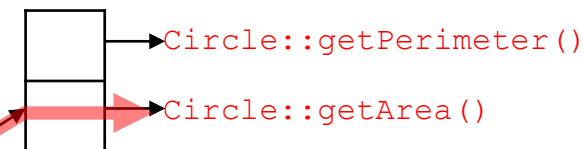
vtable for class Shape



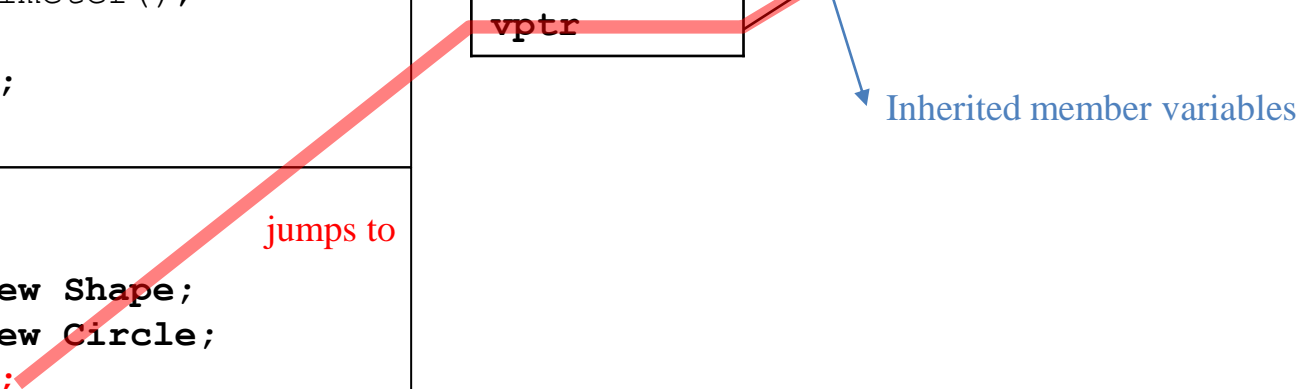
***c1**



vtable for class Circle



Inherited member variables



```

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();
    double getPerimeter();
private:
    double radius;
};

```

```

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

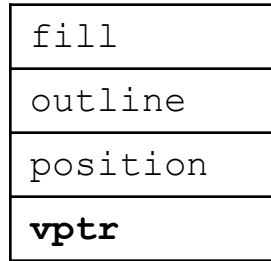
```

```

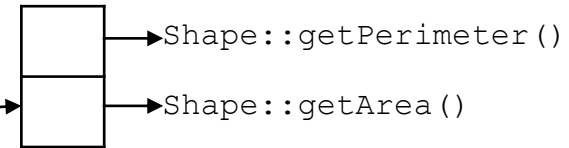
int main(){
    Shape s1; Circle c1; TextCircle tc1;
    return 0;
}

```

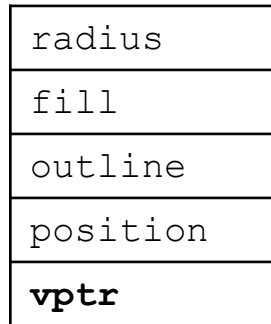
s1



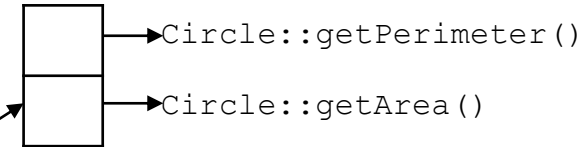
vtable for class Shape



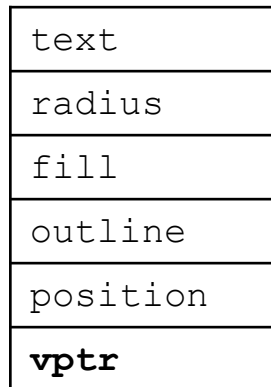
c1



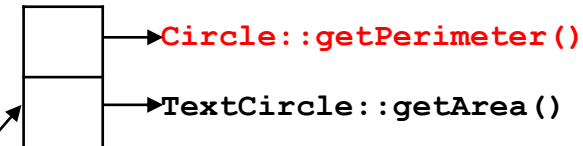
vtable for class Circle



tc1



vtable for class TextCircle

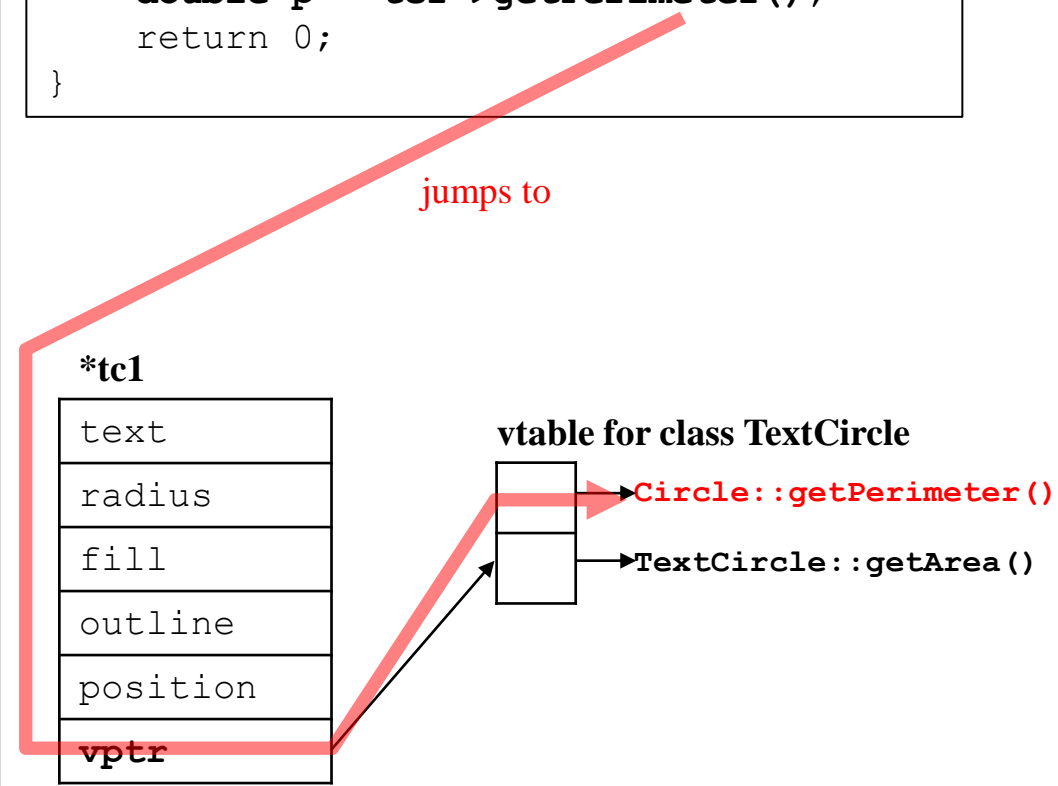


```
class Shape
{
public:
    Shape();
    virtual double getArea();
    virtual double getPerimeter();
private:
    Vector2D position;
    Color outline, fill;
};
```

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

```
class TextCircle: public Circle
{
public:
    TextCircle(string s);
    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 *vptr* depends on the compiler: `__vptr`, `__vfptr`, ...

Behind Virtual Functions

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

Quiz #1

- Go to <https://www.slido.com/>
- Join #csd-ys
- Click "Polls"

- Submit your answer in the following format:
 - **Student ID: Your answer**
 - e.g. **2017123456: 4)**

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

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() {
        // Nothing to do here
    }
};

class Rectangle : public Shape {
public:
    virtual void Draw() {
        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, you can use *pure virtual functions*.
 - Just declare a virtual function and end it with ‘= 0’

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

Pure Virtual Function

- A class with pure virtual functions cannot be instantiated.
- For its subclass to be instantiated, you must implement (override) all pure virtual functions.
 - or the subclass still has a pure virtual function and cannot be instantiated.

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

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

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

```
int main() {
    Shape s1; // => Error!

    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

- A pure virtual function in a base class specifies "**what to do**".
- Each overridden virtual function in derived classes describes "**how to do**".
- You can think a pure virtual function provides *interface* to do something.
- 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*, polymorphism 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() {...}
};

```

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,
- Then there is no reason to use virtual functions.
 - It's better to avoid using virtual functions not to have (slightly) more memory and (slightly) slower speed in this case.

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.
 - For its subclass to be instantiated, you must implement (override) all pure virtual functions.
 - or the subclass itself become an abstract class and cannot not be instantiated.

```
class Shape {
public:
    virtual void Draw() = 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 provide a constructor to initialize its member variables, since they will be inherited by its subclasses.

```
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**.
 - can be used as “base class reference(or pointer)”.

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

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

    return 0;
}
```

Pure Abstract 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.

```
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

- Go to <https://www.slido.com/>
- Join #csd-ys
- Click "Polls"

- Submit your answer in the following format:
 - **Student ID: Your answer**
 - e.g. 2017123456: 4)

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

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<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 the 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 the specified *derived class object*, `dynamic_cast` of base to derived pointer returns **null pointer** (0).
 - Note that `dynamic_cast` can only downcast polymorphic types.
 - The 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 'constness' 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

- Go to <https://www.slido.com/>
- Join #**csd-ys**
- Click "Polls"

- Submit your answer in the following format:
 - **Student ID: Your answer**
 - e.g. **2017123456: 4)**

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

Notes for C++ Casting Operators

- Hard to type! (too many characters!)
- Actually, C++ casting operators 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

- Labs for this lecture:
 - Lab1 (next Tue): Assignment 10-1
 - Lab2 (next Thur): Assignment 10-2
- Next lecture:
 - 11 – Copy Constructor, Operator Overloading