Inheritance and Polymorphism

Sai-Keung Wong

National Yang Ming Chiao Tung University

Taiwan

Intended Learning Outcomes

- Use the base class to create derived classes
- Distinguish between the base class and derived class
- List the properties of the data fields with priviate, protected, and public attributes
- Describe the process of invoking constructors when an object is created
- Describe the process of invoking destructors when an objectis destroyed
- Define an abstract class and an abstract function
- Use "virtual" to declare a function of a class
- Distinguish between static casting and dynamic casting
- Define a template class

What to learn?

- Generic programming
- Base classes, derived classes
- Calling base class constructors
- Redefining and overloading functions
- Polymorphism
- Virtual functions
- Abstract classes
- Static casting and dynamic casting (static_cast<>, dynamic_cast)
- Upcasting and downcasting
- typeid
- template

Base Classes and Derived Classes

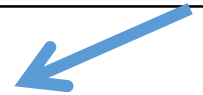
```
class Shape {
   public:
      Shape ( );
      double getArea( ) const;
   protected:
      double area; //inherited in the derived class
}; // the base class
```

Base Classes and Derived Classes

```
class Shape {
   public:
      Shape ( );
      double getArea( ) const;
   protected:
      double area; //inherited in the derived class
}; // the base class
```

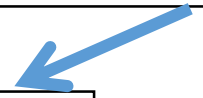
Base Classes and Derived Classes

```
class Shape {
   public:
      Shape ( );
      double getArea( ) const;
   protected:
      double area; //inherited in the derived class
}; // the base class
```



Base Classes and Derived Classes

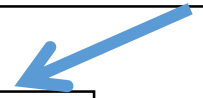
```
class Shape {
   public:
      Shape ( );
      double getArea( ) const;
   protected:
      double area; //inherited in the derived class
}; // the base class
```



```
class Circle: Shape {
   public:
      Circle( );
      double getArea( ) const;
      double getRadius() const { return radius;}
   protected:
      double radius;
}; // the derived class
```

Base Classes and Derived Classes

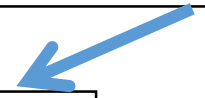
```
class Shape {
   public:
      Shape ( );
      double getArea( ) const;
   protected:
      double area; //inherited in the derived class
}; // the base class
```



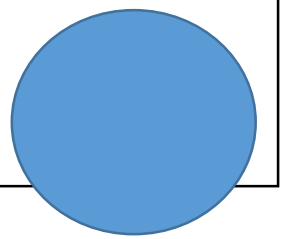
```
class Circle: Shape {
   public:
      Circle( );
      double getArea( ) const;
      double getRadius() const { return radius;}
   protected:
      double radius;
}; // the derived class
```

Base Classes and Derived Classes

```
class Shape {
   public:
      Shape ( );
      double getArea( ) const;
   protected:
      double area; //inherited in the derived class
}; // the base class
```

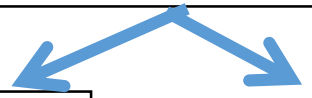


```
class Circle: Shape {
   public:
      Circle( );
      double getArea( ) const;
      double getRadius() const { return radius;}
   protected:
      double radius;
}; // the derived class
```

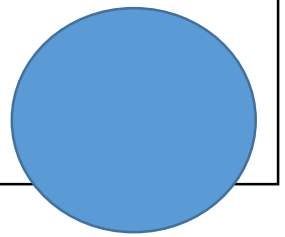


Base Classes and Derived Classes

```
class Shape {
   public:
      Shape ( );
      double getArea( ) const;
   protected:
      double area; //inherited in the derived class
}; // the base class
```



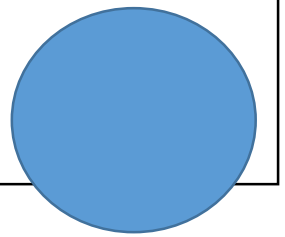
```
class Circle: Shape {
   public:
      Circle( );
      double getArea( ) const;
      double getRadius() const { return radius;}
   protected:
      double radius;
}; // the derived class
```



Base Classes and Derived Classes

```
class Shape {
   public:
      Shape ( );
      double getArea( ) const;
   protected:
      double area; //inherited in the derived class
}; // the base class
```

```
class Circle: Shape {
   public:
      Circle( );
      double getArea( ) const;
      double getRadius() const { return radius;}
   protected:
      double radius;
}; // the derived class
```

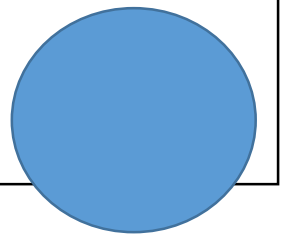


```
class Rectangle: Shape {
   public:
      Rectangle( );
      double getArea( ) const;
      double getWidth() const { return sideLength[0]; }
      double getHeight() const { return sideLength[1]; }
protected:
      double sideLength[2];
}; // the derived class
```

Base Classes and Derived Classes

```
class Shape {
   public:
      Shape ( );
      double getArea( ) const;
   protected:
      double area; //inherited in the derived class
}; // the base class
```

```
class Circle: Shape {
   public:
      Circle( );
      double getArea( ) const;
      double getRadius() const { return radius;}
   protected:
      double radius;
}; // the derived class
```

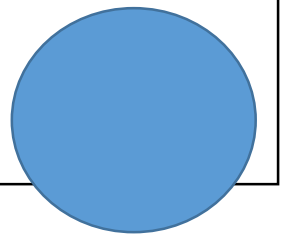


```
class Rectangle: Shape {
   public:
      Rectangle( );
      double getArea( ) const;
      double getWidth() const { return sideLength[0]; }
      double getHeight() const { return sideLength[1]; }
protected:
      double sideLength[2];
}; // the derived class
```

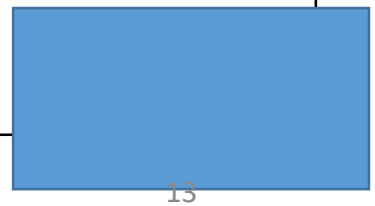
Base Classes and Derived Classes

```
class Shape {
   public:
      Shape ( );
      double getArea( ) const;
   protected:
      double area; //inherited in the derived class
}; // the base class
```

```
class Circle: Shape {
   public:
      Circle( );
      double getArea( ) const;
      double getRadius() const { return radius;}
   protected:
      double radius;
}; // the derived class
```



```
class Rectangle: Shape {
   public:
      Rectangle( );
      double getArea( ) const;
      double getWidth() const { return sideLength[0]; }
      double getHeight() const { return sideLength[1]; }
protected:
      double sideLength[2];
}; // the derived class
```

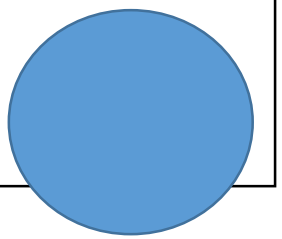


Base Classes and Derived Classes

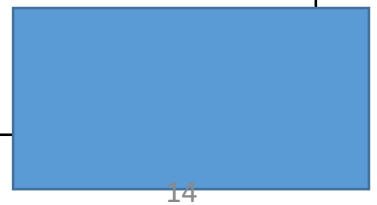
```
class Shape {
  public:              // other modifiers: protected, private
    Shape ( );
    double getArea( ) const;
  protected:
    double area; //inherited in the derived class
}; // the base class
```

Data members and methods are inherited in the derived class.

```
class Circle: Shape {
  public:
    Circle( );
    double getArea( ) const;
    double getRadius() const { return radius;}
  protected:
    double radius;
}; // the derived class
```



```
class Rectangle: Shape {
  public:
    Rectangle( );
    double getArea( ) const;
    double getWidth() const { return sideLength[0]; }
    double getHeight() const { return sideLength[1]; }
protected:
    double sideLength[2];
}; // the derived class
```



Class access specifiers

Derived classes: Class access specifiers

```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

Derived classes: Class access specifiers

```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

```
class B : public A {
        …
};
```

```
class B : protected A {
        …
};
```

```
class B : private A {
        …
};
```

```
In a client
B b;
b.x = …;
b.y = …;
c.z = …;
```

Derived classes: Class access specifiers

```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

```
class B : public A {
        …
};
```

```
class B : protected A {
        …
};
```

```
class B : private A {
        …
};
```

```
In a client
B b;
b.x = …;
b.y = …;
c.z = …;
```

Derived classes: Class access specifiers

What are the **access** A1 of the data members and functions for A3 **class access** A2?

```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

```
class B : public A {
        …
};
```

```
class B : protected A {
        …
};
```

```
class B : private A {
        …
};
```

```
In a client
B b;
b.x = …;
b.y = …;
c.z = …;
```

Derived classes: Class access specifiers

What are the **access restrictions** of the data members and functions for **different class access specifiers**?

```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

```
class B : public A {
        …
};
```

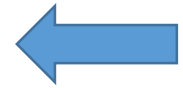
```
class B : protected A {
        …
};
```

```
class B : private A {
        …
};
```

```
In a client
B b;
b.x = …;
b.y = …;
c.z = …;
```

Class access specifiers: public, protected, private

Base class		public	protected	private
Derived class	**public**	public	protected	private
	protected	protected	protected	private
	private	private	private	private

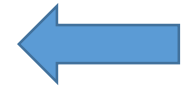


```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

```
class B : public A {
        …
};
```

Class access specifiers: public, protected, private

Base class		public	protected	private
Derived class	**public**	public	protected	private
	protected	protected	protected	private
	private	private	private	private



```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

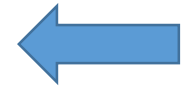
```
class B : public A {
        …
};
```

x: public
y: protected
z: private

Client

Class access specifiers: public, protected, private

Base class		public	protected	private
Derived class	**public**	public	protected	private
	protected	protected	protected	private
	private	private	private	private



```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

```
class B : public A {
        …
};
```

x: public
y: protected
z: private

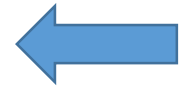
Client

In a client

A p; A1
B q; A2
p.x = 10; A3
q.x = 12; A4

public, protected, private

Base class		public	protected	private
Derived class	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private

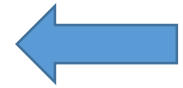


```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

```
class B : protected A {
        …
};
```

public, protected, private

Base class		public	protected	private
Derived class	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private



```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

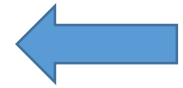
```
class B : protected A {
        …
};
```

x: protected
y: protected
z: private

Client

public, protected, private

Base class		public	protected	private
Derived class	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private



```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

```
class B : protected A {
        …
};
```

x: protected
y: protected
z: private

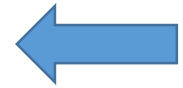
Client

In a client

```
void f() {
  A p;   A1
  B q;   A2
  p.x = 10;   A3
  q.x = 12;   A4
}
```

public, protected, private

Base class		public	protected	private
Derived class	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private



```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

```
class B : protected A {
        …
};
```

x: protected
y: protected
z: private

Client

```
void B::f() {
  A p;      A1
  B q;      A2
  p.x = 10; A3
  q.y = 12; A4
}
```

public, protected, private

Base class		public	protected	private
Derived class	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private

```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

```
class B : private A {
        …
};
```

public, protected, private

Base class		public	protected	private
Derived class	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private

```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

```
class B : private A {
        …
};
```

x: private
y: private
z: private

Client

public, protected, private

Base class		public	protected	private
Derived class	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private

```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

```
class B : private A {
        …
};
```

x: private
y: private
z: private

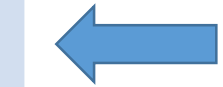
Client

In a client

```
void f( ) {
 A p;      A1
 B q;      A2
 p.x = 10; A3
 q.x = 12; A4
}
```

public, protected, private

Base class		public	protected	private
Derived class	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private



```
class A {
        public: int x;
        protected: int y;
        private: int z;
};
```

```
class B : private A {
        …
};
```

x: private
y: private
z: private

Client

In a client

```
void f( ) {
 A p;
 B q;
 p.x = 10;
 q.x = 12; //no
}
```

Generic Programming

void foo(Shape &a)

e.g., foo(x), where x is an instance of Circle or Rectangle.

Circle x; Rectangle y;

foo(x);

foo(y);

Generic Programming

Define a function with an object of a based type.

We can use an object of a derived class.

Benefit: The function is generic. The function supports object arguments which can be the base class and derived classes.

void foo(Shape &a)

e.g., foo(x), where x is an instance of Circle or Rectangle.

Circle x; Rectangle y;

foo(x);

foo(y);

Base and Derived Classes

Base Class Constructors

```
DerivedClass(parameter List): BaseClass()
{
  // initialization
}
Or
DerivedClass( parameter List ): BaseClass( argument List )
{
  // initialization
}
```

```
class A {
protected:
  int a; int b;
public:
  A( ) { a = 0; b = 1; }
  A(int a) { this->a = a; b = 3 }
};

class B : public A{
 public:
 B( ): A( ) { }
 or
 B( ): A( 2 ) { }
};
```

Base Class Constructors

A constructor constructs an instance of a class.
Purpose: Initialize the data fields in the base class.

The constructors of a base class are not inherited in the derived class.
They are invoked from the constructors of the derived classes.

```
DerivedClass(parameter List): BaseClass()
{
  // initialization
}
Or
DerivedClass( parameter List ): BaseClass( argument List )
{
  // initialization
}
```

No-Arg Constructor in Base Class

```
Square::Square( )
{
   area = 0;
}
```

No-Arg Constructor in Base Class

```
Square::Square( )
{
   area = 0;
}
```

equivalent

No-Arg Constructor in Base Class

```
Square::Square( )
{
   area = 0;
}
```

equivalent

```
Square::Square( ):Shape ( )
{
   area = 0;
}
```

No-Arg Constructor in Base Class

```
Square::Square( )
{
   area = 0;
}
```

equivalent

```
Square::Square( ):Shape ( )
{
   area = 0;
}
```

```
Square::Square( double side)
{
   this->side = side;
   computeArea( );
}
```

No-Arg Constructor in Base Class

```
Square::Square( )
{
   area = 0;
}
```

equivalent

```
Square::Square( ):Shape ( )
{
   area = 0;
}
```

```
Square::Square( double side)
{
   this->side = side;
   computeArea( );
}
```

equivalent

No-Arg Constructor in Base Class

```
Square::Square( )
{
   area = 0;
}
```

equivalent

```
Square::Square( ):Shape ( )
{
   area = 0;
}
```

```
Square::Square( double side)
{
   this->side = side;
   computeArea( );
}
```

equivalent

```
Square::Square( double side): Shape( )
{
   this->side = side;
   computeArea( );
}
```

No-Arg Constructor in Base Class

A constructor in a derived class must invoke a constructor in its base class.

The base class's A1 **constructor** is invoked A2 if the base constructor is not invoked explicitly.

```
Square::Square( )
{
   area = 0;
}
```

equivalent

```
Square::Square( ):Shape ( )
{
   area = 0;
}
```

```
Square::Square( double side)
{
   this->side = side;
   computeArea( );
}
```

equivalent

```
Square::Square( double side): Shape( )
{
   this->side = side;
   computeArea( );
}
```

No-Arg Constructor in Base Class

A constructor in a derived class must invoke a constructor in its base class.

The base class's no-arg constructor is invoked by default if the base constructor is not invoked explicitly.

```
Square::Square( )
{
  area = 0;
}
```

equivalent

```
Square::Square( ):Shape ( )
{
  area = 0;
}
```

```
Square::Square( double side)
{
  this->side = side;
  computeArea( );
}
```

equivalent

```
Square::Square( double side): Shape( )
{
  this->side = side;
  computeArea( );
}
```

Constructor and Destructor Chaining

Constructor and Destructor Chaining

Constructor and Destructor Chaining

C *x;

Constructor and Destructor Chaining

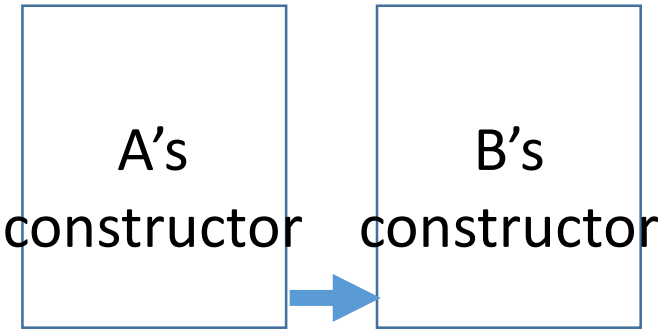
C *x; x = new C;

Constructor and Destructor Chaining

A’s constructor

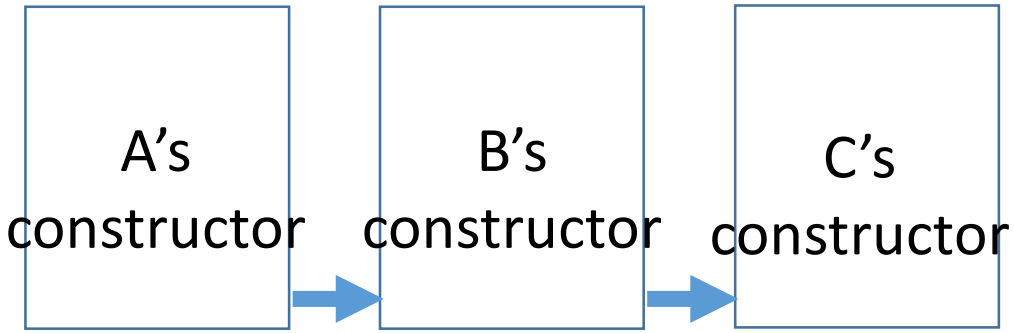
C *x; x = new C;

Constructor and Destructor Chaining



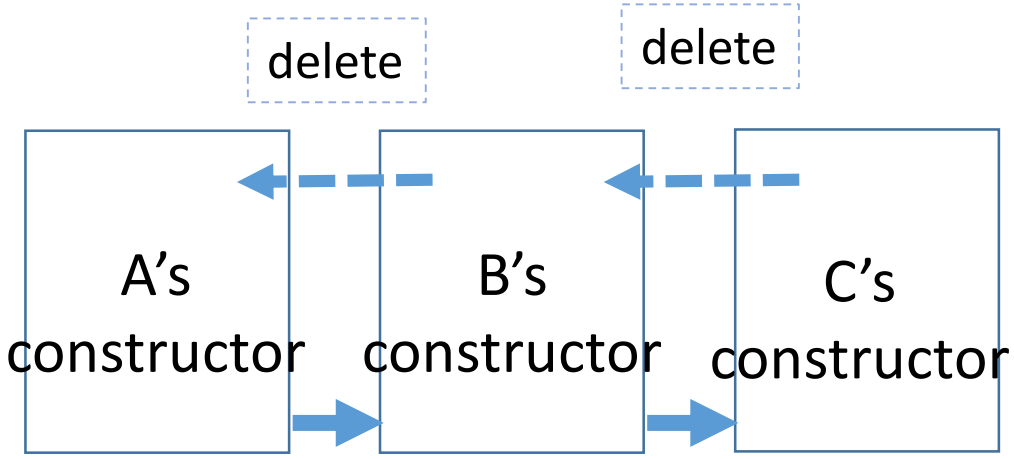
C *x; x = new C;

Constructor and Destructor Chaining



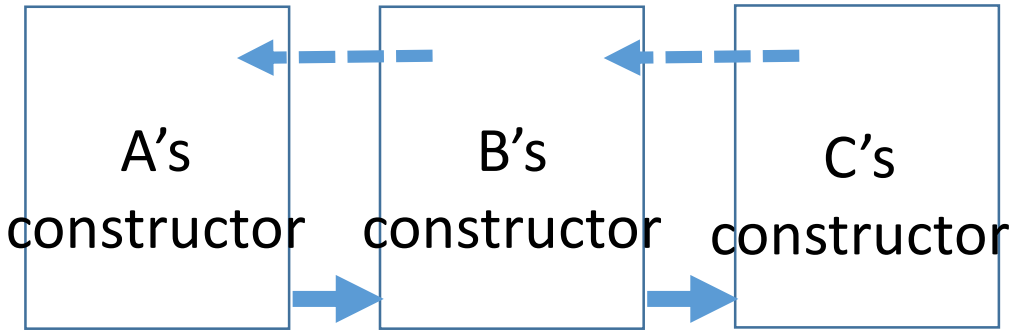
C *x; x = new C;

Constructor and Destructor Chaining



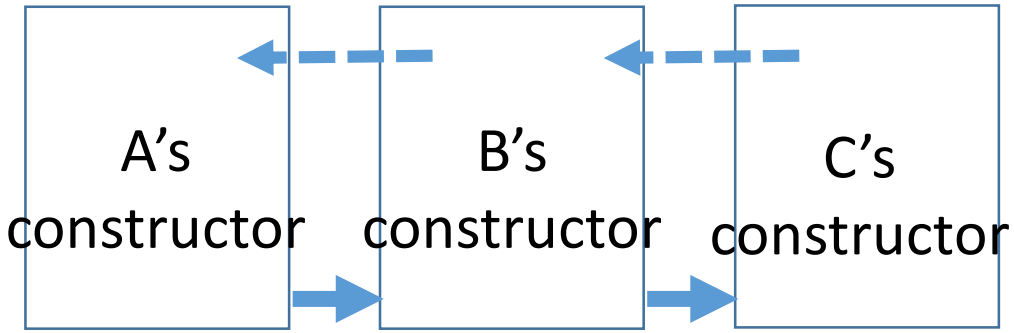
C *x; x = new C;

Constructor and Destructor Chaining



C *x; x = new C;

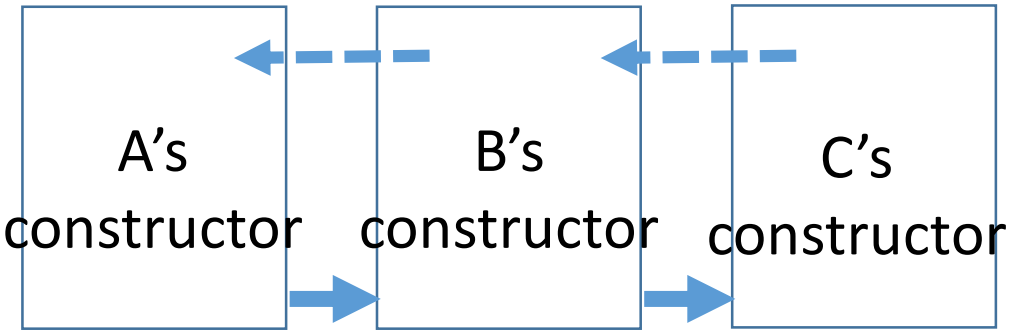
Constructor and Destructor Chaining

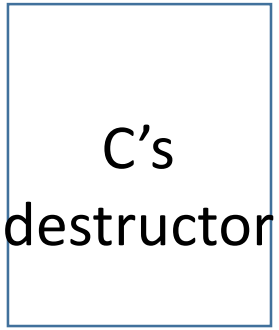


C *x; x = new C;

delete x;

Constructor and Destructor Chaining

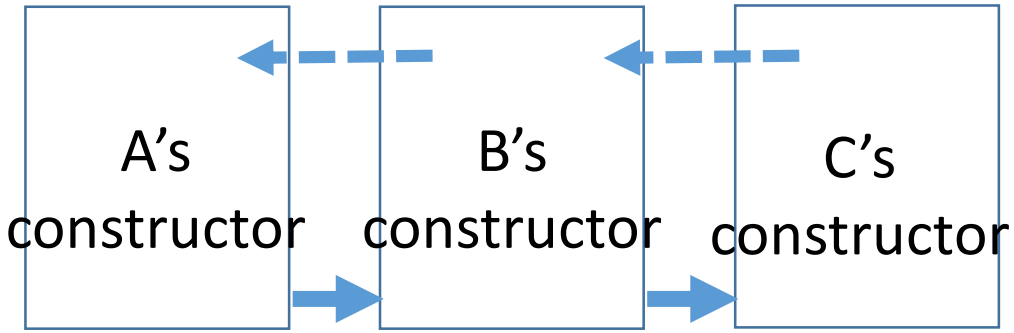




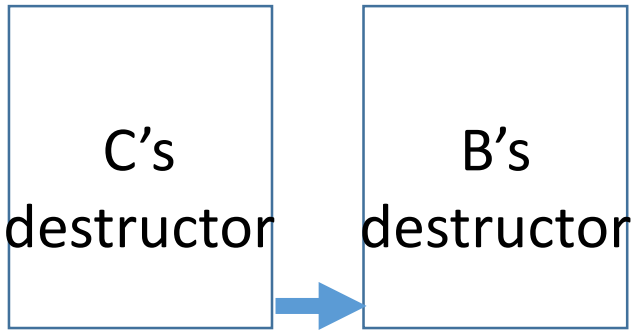
C *x; x = new C;

delete x;

Constructor and Destructor Chaining

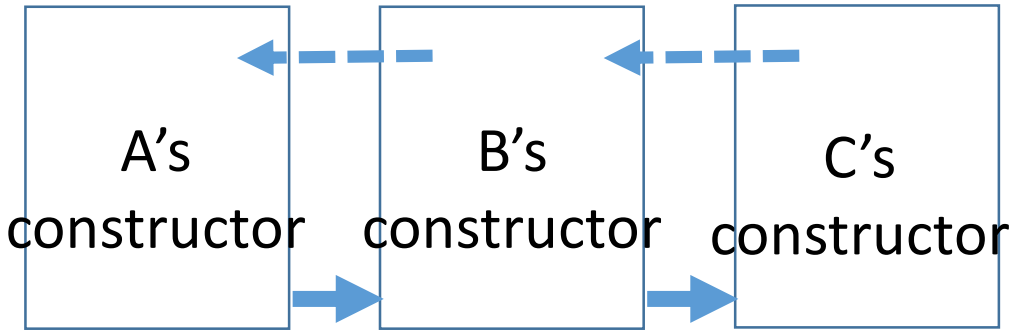


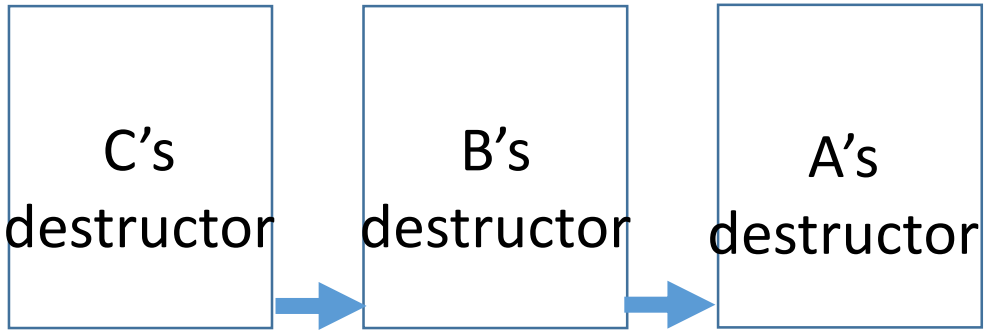
C *x; x = new C;



delete x;

Constructor and Destructor Chaining





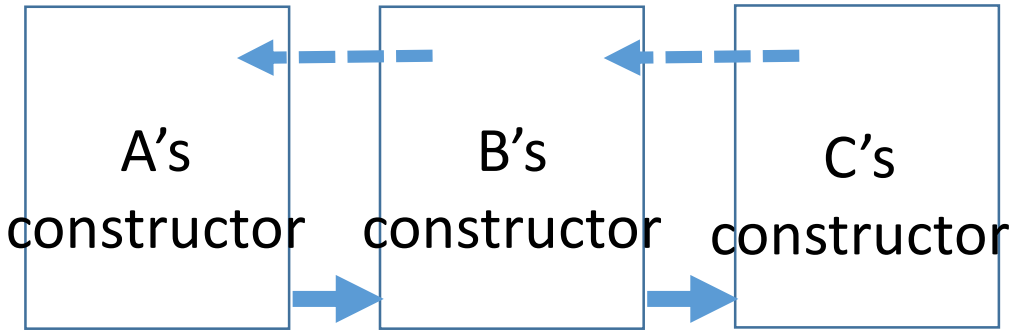
C *x; x = new C;

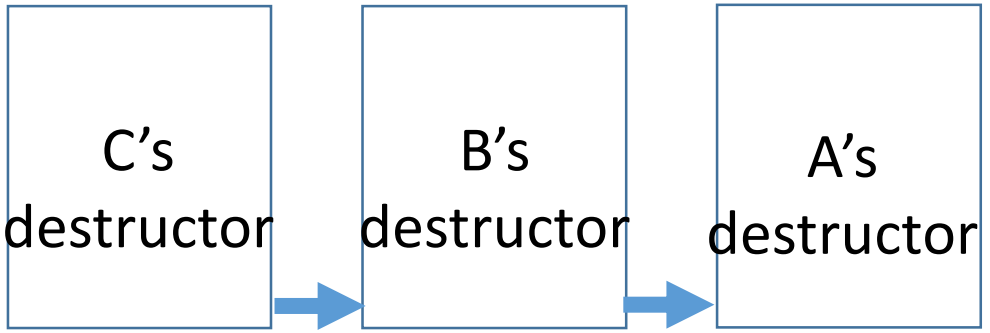
delete x;

Constructor and Destructor Chaining

- The constructors of all the base classes along the inheritance chain are invoked when an instance of class is constructed.

- A base class's constructor is called before the derived class's constructor.
- The destructors are invoked in reverse order, with the derived class's destructor invoked first.





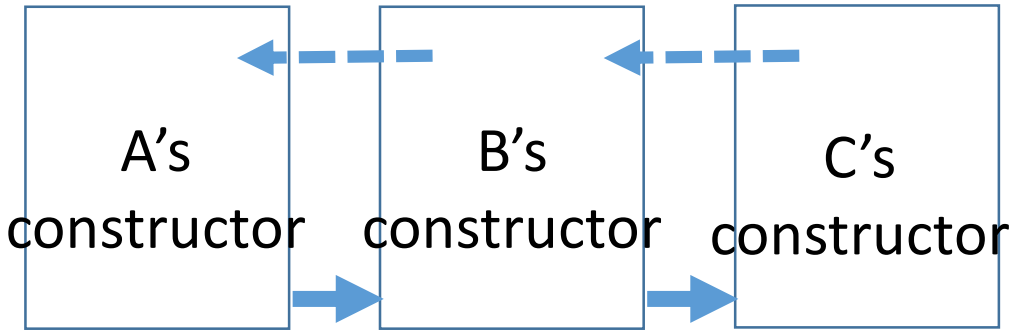
C *x; x = new C;

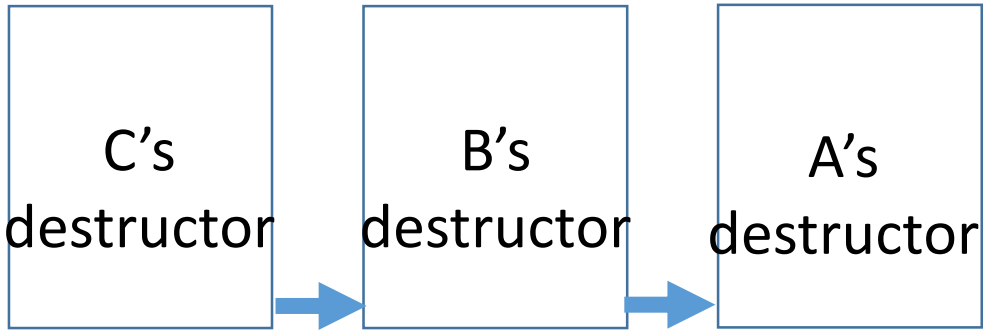
delete x;

Constructor and Destructor Chaining

- The constructors of all the base classes along the inheritance chain are invoked when an instance of class is constructed.

- A base class's constructor is called before the derived class's constructor.
- The destructors are invoked in reverse order, with the derived class's destructor invoked first.





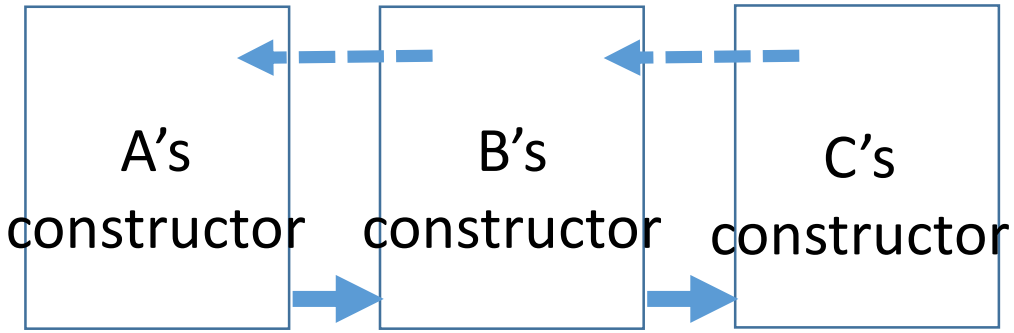
C *x; x = new C;

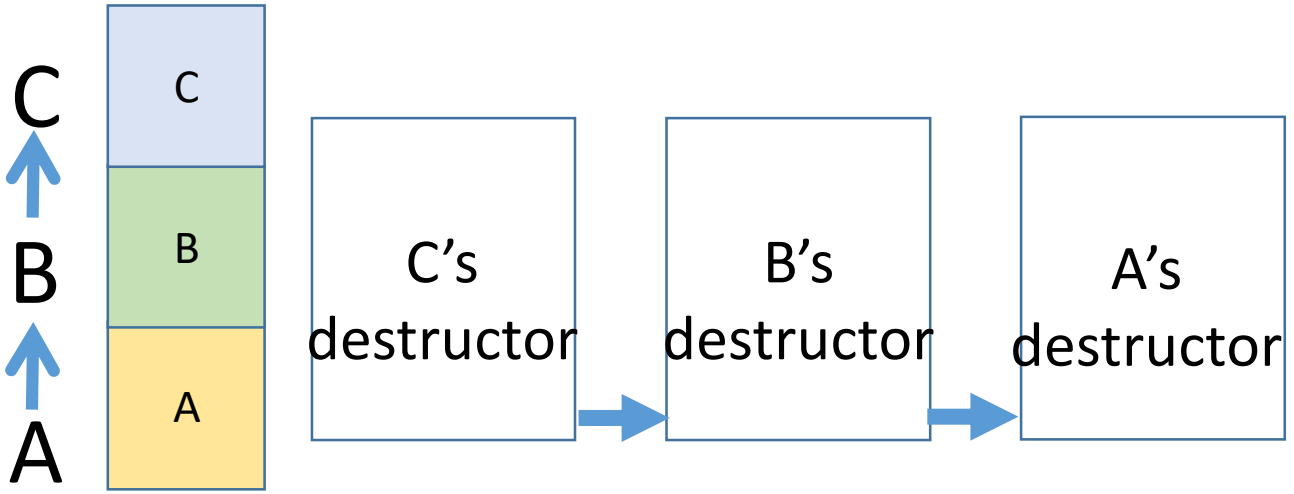
delete x;

Constructor and Destructor Chaining

- The constructors of all the base classes along the inheritance chain are invoked when an instance of class is constructed.

- A base class's constructor is called before the derived class's constructor.
- The destructors are invoked in reverse order, with the derived class's destructor invoked first.





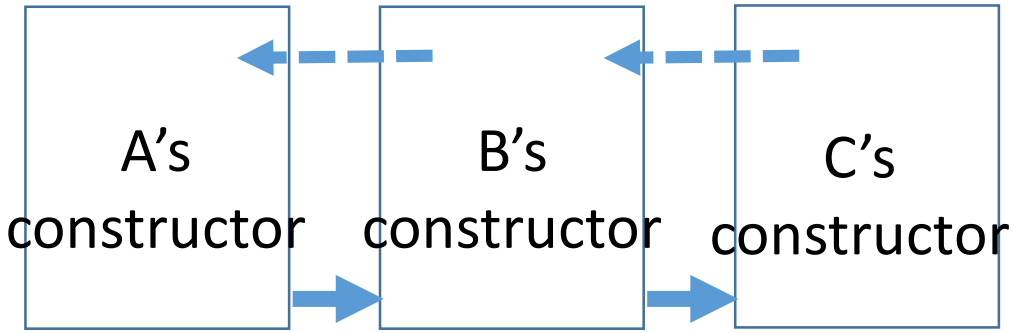
C *x; x = new C;

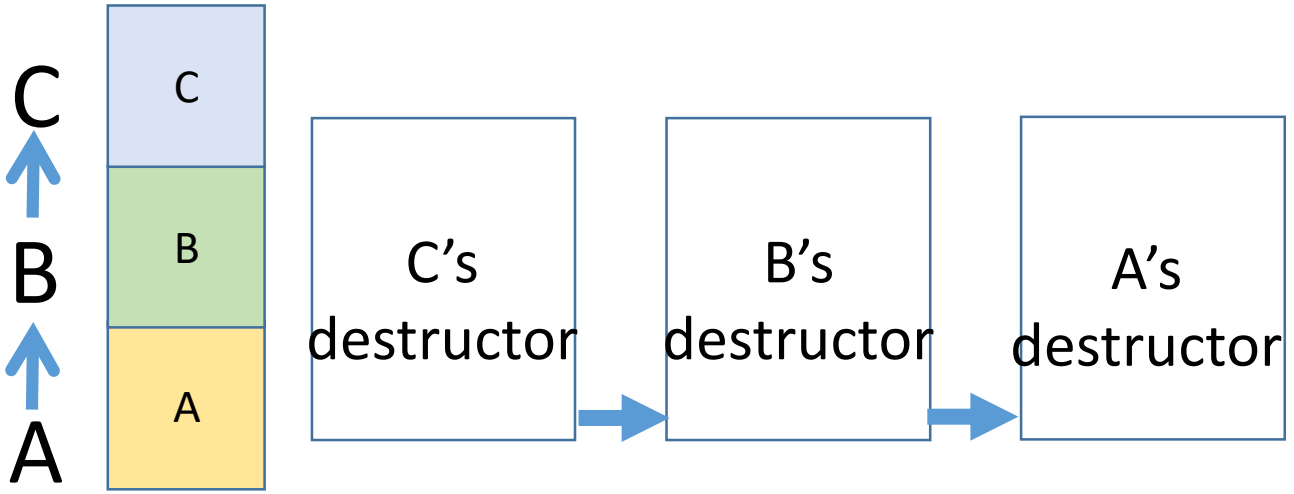
delete x;

Constructor and Destructor Chaining

- The constructors of all the base classes along the inheritance chain are invoked when an instance of class is constructed.

- A base class's constructor is called before the derived class's constructor.
- The destructors are invoked in reverse order, with the derived class's destructor invoked first.





C *x; x = new C;

delete x;

no-arg constructor

Provide a no-arg constructor to avoid programming errors for a derived class.

```
class Shape{
public:
  Shape(int id)  {
  }
};

class Rectangle: public Shape{
public:
  Rectangle( ) { }
};
```

no-arg constructor

Provide a no-arg constructor to avoid programming errors for a derived class.

```
class Shape{
public:
  Shape(int id)  {
  }
};

class Rectangle: public Shape{
public:
  Rectangle( ) { }
};
```

Rectangle a; Error?

no-arg constructor

Provide a no-arg constructor to avoid programming errors for a derived class.

```
class Shape{
public:
  Shape(int id)  {
  }
};

class Rectangle: public Shape{
public:
  Rectangle( ) { }
};
```

Rectangle a; // error

no-arg constructor

Provide a no-arg constructor to avoid programming errors for a derived class.

```
class Shape{
public:
  Shape(int id)  {
  }
};

class Rectangle: public Shape{
public:
  Rectangle( );
};

Rectangle::Rectangle( ):Shape( 10 ) { }
```

no-arg constructor

Provide a no-arg constructor to avoid programming errors for a derived class.

```
class Shape{
public:
  Shape(int id)  {
  }
};

class Rectangle: public Shape{
public:
  Rectangle( );
};

Rectangle::Rectangle( ):Shape( 10 ) { }
```

Rectangle a; Error?

no-arg constructor

Provide a no-arg constructor to avoid programming errors for a derived class.

```
class Shape{
public:
  Shape(int id)  {
  }
};

class Rectangle: public Shape{
public:
  Rectangle( );
};

Rectangle::Rectangle( ):Shape( 10 ) { }
```

Rectangle a; // ok

Redefining Functions

```
class Shape{
public:
   ...
   double printf( ) const;
};

class Rectangle: public Shape{
   double side;
public:

   double printf( ) const;
};
```

Redefining Functions

We can redefine the functions of the base class in the Circle and Rectangle classes.

Purpose: Implement a more specific description that is tailored to the derived objects.

```
class Shape{
public:
   ...
   double printf( ) const;
};

class Rectangle: public Shape{
   double side;
public:

   double printf( ) const;
};
```

Override

```
class A {
public:
        virtual void foo( );
};

class B : public A {
        void foo( ) override;                    //explicitly
}
A *x; x->foo( );
```

Invoke functions in the base

```
class Shape{
public:
   double printf( ) const;
};
class Circle: public Shape{
public:
  double printf( ) const;
};
```

Invoke functions in the base

Circle circle;

circle.Shape::printf( )

```
class Shape{
public:
   double printf( ) const;
};
class Circle: public Shape{
public:
  double printf( ) const;
};
```

Invoke functions in the base

- Invoke a function defined in the base class.
- Use the scope resolution operator (::) with the base class name.

Circle circle;

circle.Shape::printf( )

```
class Shape{
public:
   double printf( ) const;
};
class Circle: public Shape{
public:
  double printf( ) const;
};
```

Redefining vs. Overloading

- Overloading: Define functions with the same name but with different signatures.
- Redefining: the function must be defined in the derived class using the same signature and same return type as in its base class.

```
void printf( );

void printf( int a );
```

```
class Shape{
public:
    double printf( ) const;
};
class Rectangle: public Shape{
public:
    double printf( int a ) const;
};
```

Redefining vs. Overloading

- Overloading: Define functions with the same name but with different signatures.
- Redefining: the function must be defined in the derived class using the same signature and same return type as in its base class.

```
void printf( );

void printf( int a );
```

```
class Shape{
public:
    double printf( ) const;
};
class Rectangle: public Shape{
public:
    double printf( ) const;
};
```

Polymorphism

Polymorphism

Polymorphism: a variable of a supertype can refer to a subtype object.

The base class must have at least one virtual function.

Three major concepts in object-oriented programming: encapsulation, inheritance, and polymorphism.

B is derived from A.

```
A *x;
x = new B;
```

Virtual Functions

```
class Shape{
public:
   double printf( ) const;
};

class Rectangle: public Shape{
public:
  double printf( ) const;
};

class Circle: public Shape{
public:
  double printf( ) const;
};
```

```
Shape *a, *b;

a = new Rectangle;

a->printf( );

b = new Circle;

b->printf( );

How the function printf
can be invoked based on
the object's type?
```

Virtual Functions

```
class Shape{
public:
  double printf( ) const;
};

class Rectangle: public Shape{
public:
  double printf( ) const;
};

class Circle: public Shape{
public:
  double printf( ) const;
};
```

Invoke which function?

```
Shape *a, *b;

a = new Rectangle;

a->printf( );

b = new Circle;

b->printf( );

How the function printf
can be invoked based on
the object's type?
```

Virtual Functions

```
class Shape{
public:
  double printf( ) const;
};

class Rectangle: public Shape{
public:
  double printf( ) const;
};

class Circle: public Shape{
public:
  double printf( ) const;
};
```

Invoke which function?

```
Shape *a, *b;

a = new Rectangle;

a->printf( );

b = new Circle;

b->printf( );

How the function printf
can be invoked based on
the object's type?
```

Virtual Functions

```
class Shape{
public:
   double printf( ) const;
};

class
pub
   d
c
p

};
```

Need Dynamic Binding.

Determine the type of an object at runtime and invoke the required function.

```
Shape *a, *b;

a = new Rectangle;

a->printf( );

b = new Circle;

b->printf( );

How the function printf
can be invoked based on
the object's type?
```

Virtual Functions

```
class Shape{
public:
   virtual double printf( ) const;
};

class Rectangle: public Shape{
public:
  double printf( ) const;
};

class Circle: public Shape{
public:
  double printf( ) const;
};
```

```
Shape *a, *b;

a = new Rectangle;

a->printf( );

b = new Circle;

b->printf( );
```

How the function printf can be invoked based on the object's type?

Virtual Functions

```
class Shape{
public:
 virtual double printf( ) const;
};

class Rectangle: public Shape{
public:
double printf( ) const;
};

class Circle: public Shape{
public:
  double printf( ) const;
};
```

```
Shape *a, *b;

a = new Rectangle;

a->printf( );

b = new Circle;

b->printf( );
```

How the function printf can be invoked based on the object's type?

Virtual Functions

```
class Shape{
public:
 virtual double printf( ) const;
};

class Rectangle: public Shape{
public:
  double printf( ) const;
};

class Circle: public Shape{
public:
double printf( ) const;
};
```

```
Shape *a, *b;

a = new Rectangle;

a->printf( );

b = new Circle;

b->printf( );
```

How the function printf can be invoked based on the object's type?

Virtual Functions

```
class Shape{
public:
   virtual double printf( ) const;
};

class Rectangle: public Shape{
public:
  double printf( ) const;
};

class Circle: public Shape{
public:
  double printf( ) const;
};
```

```
Shape *a, *b;

a = new Rectangle;

a->printf( );

b = new Circle;

b->printf( );
```

How the function printf can be invoked based on the object's type?

static matching vs. dynamic binding

Two separate issues:

- matching a function signature
- binding a function implementation

```
void A::printf() const
void B::printf() const
```

- The compiler finds a matching function based on parameter information: the parameter type, number of parameters, and order of the parameters at compile time.

- C++ dynamically binds the implementation of the function at runtime, decided by the *actual class* of the object referenced by the variable.

When should we use virtual functions?

- Yes: If a function defined in a base class needs to be redefined in its derived classes

- No: if a function will not be redefined.

Benefit: more efficient. It takes longer time and system resources to bind virtual functions dynamically.

Abstract Classes

Static Casting and Dynamic Casting

Abstract Classes

- In the inheritance hierarchy, classes become more specific and concrete with each new derived class.
- Moving from a derived class back up to its parent and ancestor classes, these classes become more general and less specific.

- A base class should contain common features of its derived classes.
- However, a base class is so abstract that it cannot have any specific instances. Then we design it as an abstract class.

Abstract function and class

```
class Shape {                          // abstract class
     public:
     virtual double getArea() const = 0; // abstract function
};

class Diamond: public Shape {
     public:
     virtual double getArea() const { … } // define getArea
};
class Square: public Diamond {
     public:
     virtual double getArea() const { … } // define getArea
};
```

Abstract Class Example

```
class Shape {
   public:
      BaseShape( );
      double getArea( )
const = 0;
   protected:
}; // the base class
```

```
class Circle: Shape {
   public:
      Circle( );
      double getArea( )
const;
   protected:
      double radius;
}; // the derived class
```

```
class Rectangle: Shape {
   public:
      Circle( );
      double getArea( ) const;
   protected:
      double sideLength[2];
}; // the derived class

Shape a; // error.
```

Casting: static_cast versus dynamic_cast

void Shape::displayShapeInfo(const Shape& object)

We want to modify this function to display the information of the object based on its shape.

For example, for a circle object, we display radius, diameter, area, and perimeter.

Static Casting: normal/ordinary type conversion

```
void displayShapeInfo(Shape& g)    // generic function
{
  cout << "The raidus is "    << g.getRadius() << endl;
  cout << "The diameter is "  << g.getDiameter() << endl;
  cout << "The width is "     << g.getWidth() << endl;
  cout << "The height is "    << g.getHeight() << endl;
  cout << "The area is "      << g.getArea() << endl;
  cout << "The perimeter is " << g.getPerimeter() << endl;
}
// casting mechanism
What should we do if we want to show the information of a
rectangle or another shape?
```

Static Casting

```
void displayShapeInfo(Shape& g) {
  Shape* p = &g;
  cout << "The raidus is "       <<  static_cast<Circle*>(p)->getRadius() << endl;

  cout << "The diameter is "     <<  static_cast<Circle*>(p)->getDiameter() << endl;

  cout << "The width is "        <<  static_cast<Rectangle*>(p)->getWidth() << endl;

  cout << "The height is "       <<  static_cast<Rectangle*>(p)->getHeight() << endl;

  cout << "The area is "         << g.getArea() << endl;

  cout << "The perimeter is "    << g.getPerimeter() << endl;
}
```

Static Casting

```
void displayShapeInfo(Shape& g) {
  Shape* p = &g;
  cout << "The raidus is "       <<  static_cast<Circle*>(p)->getRadius() << endl;

  cout << "The diameter is "     <<  static_cast<Circle*>(p)->getDiameter() << endl;

  cout << "The width is "        <<  static_cast<Rectangle*>(p)->getWidth() << endl;

  cout << "The height is "       <<  static_cast<Rectangle*>(p)->getHeight() << endl;

  cout << "The area is "         << g.getArea() << endl;

  cout << "The perimeter is "    << g.getPerimeter() << endl;
}
```

Cannot guarantee that the type conversion is correct.

Static Casting

```
void displayShapeInfo(Shape& g) {
  Shape* p = &g;
  cout << "The raidus is "       <<  static_cast<Circle*>(p)->getRadius() << endl;

  cout << "The diameter is "     <<  static_cast<Circle*>(p)->getDiameter() << endl;

  cout << "The width is "        <<  static_cast<Rectangle*>(p)->getWidth() << endl;

  cout << "The height is "       <<  static_cast<Rectangle*>(p)->getHeight() << endl;

  cout << "The area is "         << g.getArea() << endl;

  cout << "The perimeter is "    << g.getPerimeter() << endl;
}
```

Cannot guarantee that the type conversion is correct.

Dynamic Casting

```
Shape *p = &object;
Circle *p1 = dynamic_cast<Circle*>(p);
if (p1 != 0)
{
  cout << "The radius is "    << p1->getRadius() << endl;
  cout << "The diameter is "  << p1->getDiameter() << endl;
}
```

Dynamic Casting

Use dynamic_cast to perform type conversion.

We can check whether the conversion is successful.

```
Shape *p = &object;
Circle *p1 = dynamic_cast<Circle*>(p);
if (p1 != 0)
{
  cout << "The radius is "    << p1->getRadius() << endl;
  cout << "The diameter is "  << p1->getDiameter() << endl;
}
```

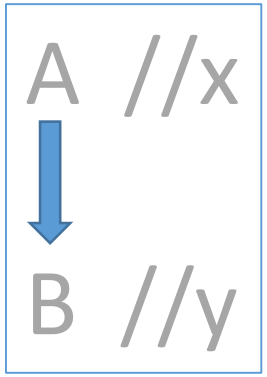
Upcasting and Downcasting

Upcasting: assigning a pointer of a derived class type to a pointer of its base class type.

Downcasting: assigning a pointer of a base class type to a pointer of its derived class type.

B is derived from A.

```
A *x;
B *y;
x = y; // upcasting
```




```
A *x;
B *y;
y = x; // downcasting
```

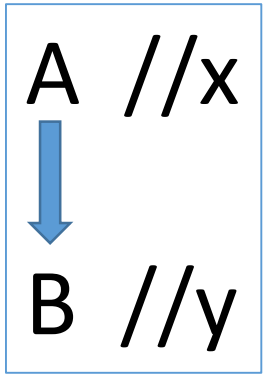
Upcasting and Downcasting

Upcasting: assigning a pointer of a derived class type to a pointer of its base class type.

Downcasting: assigning a pointer of a base class type to a pointer of its derived class type.

B is derived from A.

```
A *x;
B *y;
x = y; // upcasting
```



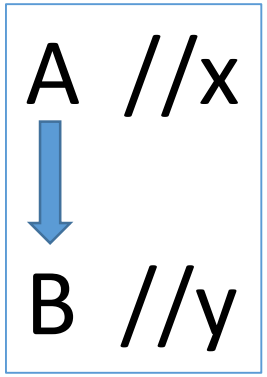
Upcasting and Downcasting

Upcasting: assigning a pointer of a derived class type to a pointer of its base class type.

Downcasting: assigning a pointer of a base class type to a pointer of its derived class type.

B is derived from A.

```
A *x;
B *y;
x = y; // upcasting
```

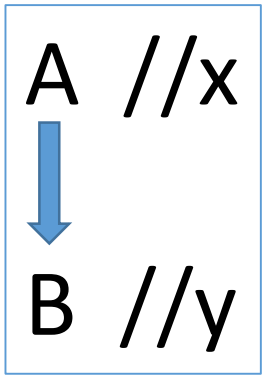



```
A *x;
B *y;
y = x; // downcasting
```

Upcasting and Downcasting

B is derived from A.

```
A *x;
B *y;
x = y; // upcasting
```




```
A *x;
B *y;
y = x; // downcasting
```

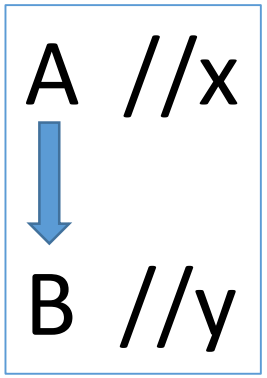
```
A *x;
B *y;
x = y; // ok
```

```
A *x;
B *y;
y = x; // error! How?
       // what can we do?
```

Upcasting and Downcasting

B is derived from A.

```
A *x;
B *y;
x = y; // upcasting
```



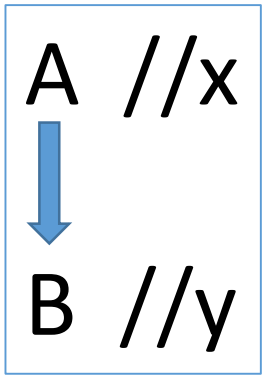

```
A *x;
B *y;
y = x; // downcasting
```

```
A *x;
B *y;
x = y; // ok
```

Upcasting and Downcasting

B is derived from A.

```
A *x;
B *y;
x = y; // upcasting
```




```
A *x;
B *y;
y = x; // downcasting
```

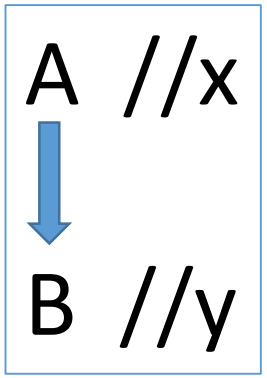
```
A *x;
B *y;
x = y; // ok
```

```
A *x;
B *y;
y = x; // error! How?
       // what can we do?
```

Upcasting and Downcasting

B is derived from A.

```
A *x;
B *y;
x = y; // upcasting
```




```
A *x;
B *y;
y = x; // downcasting
```

```
A *x;
B *y;
x = y; // ok
```

```
//use dynamic cast
y = dynamic_cast<B*> (x);
```

```
A *x;
B *y;
y = x; // error! How?
       // what can we do?
```

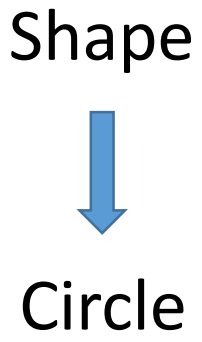
Upcasting and Downcasting

Upcasting can be performed implicitly without using the dynamic_cast operator.

Upcasting and Downcasting

Upcasting can be performed implicitly without using the dynamic_cast operator.

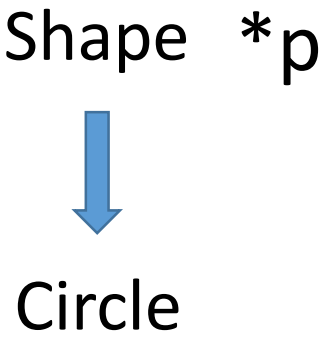
```
Shape *p = new Circle(1);
Circle *p1 = new Circle(2);
p = p1;
```



Upcasting and Downcasting

Upcasting can be performed implicitly without using the dynamic_cast operator.

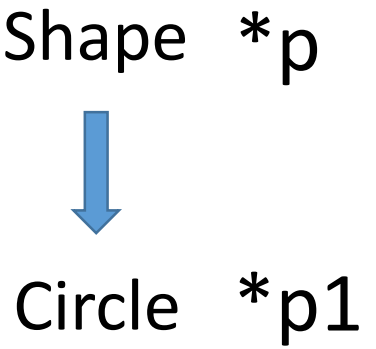
```
Shape *p = new Circle(1);
Circle *p1 = new Circle(2);
p = p1;
```



Upcasting and Downcasting

Upcasting can be performed implicitly without using the dynamic_cast operator.

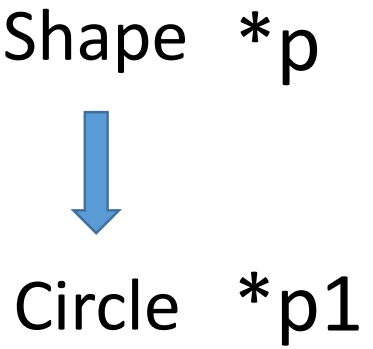
```
Shape *p = new Circle(1);
Circle *p1 = new Circle(2);
p = p1;
```



Upcasting and Downcasting

Upcasting can be performed implicitly without using the dynamic_cast operator.

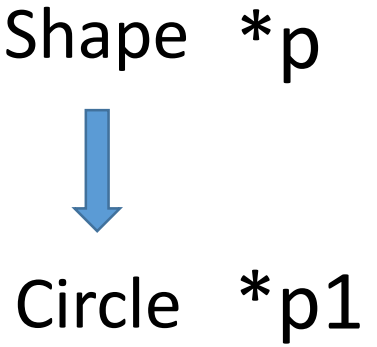
```
Shape *p = new Circle(1);
Circle *p1 = new Circle(2);
p = p1;
```



Upcasting and Downcasting

Upcasting can be performed implicitly without using the dynamic_cast operator.

```
Shape *p = new Circle(1);
Circle *p1 = new Circle(2);
p = p1;
```



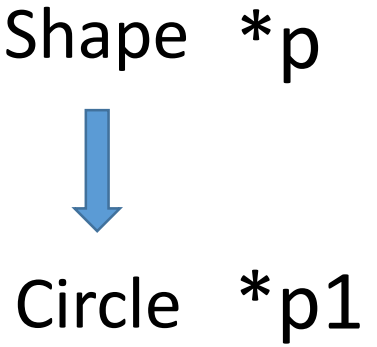
Downcasting must be performed explicitly.

```
p1 = dynamic_cast<Circle*>(p);
```

Upcasting and Downcasting

Upcasting can be performed implicitly without using the dynamic_cast operator.

```
Shape *p = new Circle(1);
Circle *p1 = new Circle(2);
p = p1;
```



Downcasting must be performed explicitly.

```
p1 = dynamic_cast<Circle*>(p);
```

typeid operator

We can use the typeid operator to return a reference to an object of class type_info.

For example:

string x;

```
cout << typeid(x).name() << endl;
```

It displays *string*.

x is an object of the *string* class.

Templates

Templates

How do we design a generic function that can perform similar tasks for inputs with different data types?

```
// return the sum of two numbers
int add( int, int );
int add( float, float );
int add( double, double );
```

Templates

```
int maxValue(
    const int& value1,
    const int& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

Templates

```
double maxValue(
     const double& value1,
     const double& value2 )
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

```
int maxValue(
     const int& value1,
     const int& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

Templates

```
char maxValue(
    const char& value1,
    const char& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

```
int maxValue(
    const int& value1,
    const int& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

Generic Functions

```
template<typename GenericType>
GenericType maxValue(
     const GenericType& value1,
     const GenericType& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

```
int maxValue(
     const int& value1,
     const int& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

Generic Functions

```
template<typename GenericType>
GenericType maxValue(
     const GenericType& value1,
     const GenericType& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

```
maxValue(1.0, 2.0);
maxValue(7, 3);

maxValue(7, 3.0); // error. Why?
```

Generic Functions

```
template<typename GenericType>
GenericType maxValue(
     const GenericType& value1,
     const GenericType& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

```
maxValue(1.0, 2.0);
maxValue(7, 3);
maxValue(7, 3.0); // error. Why?

int maxValue(int, int);
int maxValue(double, double);
// ambiguous
```

Generic Functions

```
A x, y;
x > y   //operand
class A {
        int c0, c1;
        friend bool operator>(
        const A &a, const A &b);
};
bool operator>(const A &a, const A &b) {
        ……
}
//Must define the comparison operator
```

```
template<typename GenericType>
GenericType maxValue(
      const GenericType& value1,
      const GenericType& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

Match parameter

The generic maxValue function can be used to return a maximum of two values of *any type*, provided that

- The two values have the same type;
- The two values **can be compared**

  using the > operator.

```
template<typename GenericType>
GenericType maxValue(
  const GenericType& value1,
  const GenericType& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}


A y, z;
A x = maxValue(y, z);
```

<typename T>

Use either <typename T> or <class T> to specify a type parameter.

Using <typename T> is better because <typename T> is **descriptive**.

<class T> could be confused with class declaration.

```
template<typename T>
T maxValue(
  const T& value1,
  const T& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}

A y, z;
A x = maxValue(y, z);
```

Multiple type parameters

Syntax: <typename T1, typename T2, typename T3, …>.

```
template<
        typename T1,
        typename T2,
        typename T3 >
        T1 foo(T1 a, T2 b)
{
        …
}
```

Multiple type parameters

Syntax: **<**typename T1, typename T2, typename T3, …**>**.

```
template<
        typename T1,
        typename T2,
        typename T3 >
        T1 foo(T1 a, T2 b)
{
        …
}
```

Let's review what we have done ☺ Any error?

```
template<typename myType>
T maxValue( const T& value1, const T& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

Let's review what we have done ☺ Any error?

```
template<typename myType>
T maxValue( const T& value1, const T& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

Let's review what we have done ☺ Any error?

```
template<typename myType>
T maxValue( const T& value1, const T& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

```
template<typename T>
T maxValue( const T& value1, const T& value2)
{
 if (value1 > value2)
   return value1;
 else
   return value2;
}
```

Let's review what we have done ☺ Any error?

```
template<typename myType>
T maxValue( const T& value1, const T& value2)
{
  if (value1 > value2)
    return value1;
  else
    return value2;
}
```

```
template<typename mT>
mT maxValue( const mT& value1, const mT& value2)
{
 if (value1 > value2)
   return value1;
 else
   return value2;
}
```

Example: A Generic Sort

Design a generic sort function.

Example: A Generic Sort

```
void sort( vector<int> &arr ) {
        for (int i = 0; i < arr.size(); ++i) {
                for (int j = i+1; j < arr.size(); ++j) {
                        ……
                }
        }
}
e.g.,
vector<double> x;
// generate elements for x
sort(x);
```

Example: A Generic Sort

```
template<typename T> void sort( T &arr ) {
        for (int i = 0; i < arr.size(); ++i) {
                for (int j = i+1; j < arr.size(); ++j) {
                        ……
                }
        }
}
```

```
void sort( vector<int> &arr )
```

e.g.,

```
vector<double> x;
// generate elements for x
sort(x);
```

Example: A Generic Sort

```
template<typename T> void sort( T &arr ) {
        for (int i = 0; i < arr.size(); ++i) {
                for (int j = i+1; j < arr.size(); ++j) {
                        ……
                }
        }
}
```

```
e.g.,
vector<double> x;
// generate elements for x
sort(x);

//any error?
 vector<double> y;
 sort<vector<double>>(y);
```

Example: A Generic Sort

```
template<typename T> void sort( T &arr ) {
        for (int i = 0; i < arr.size(); ++i) {
                for (int j = i+1; j < arr.size(); ++j)
{
                        ……
                }
        }
}
```

```
e.g.,
vector<double> x;
// generate elements for x
sort(x);

//any error?
 vector<double> y;
 sort<vector<double>>(y);
```

A General Approach for Designing Generic Functions

We should do the following steps to define a generic function:

1. Start with a non-generic function,
2. Debug
3. Test it
4. Convert it to a generic function, i.e., changing the datatypes of variables as type parameters

How to implement a generic class?

```
class stack {
        protected:
                int arr[100];
        public:
                stack( ) { … }
                int top( ) {…}
                ……
};


stack x;
```

How to implement a generic class?

```
class stack {
        protected:
                int arr[100];
        public:
                stack( ) { … }
                int top( ) {…}
                ……
};

stack x;
```

```
A1  A2  class stack {
        protected:
            A3  arr[100];
        public:
                stack( ) { … }
            A4  top( ) {…}
                ……
};
```

How to implement a generic class?

```
class stack {
        protected:
                int arr[100];
        public:
                stack( ) { … }
                int top( ) {…}
                ……
};

stack x;
```

```
template <typename T> class stack {
        protected:
                T arr[100];
        public:
                stack( ) { … }
                T top( ) {…}
                ……
};
stack<double> x;
stack><int> y;
stack<myType> z;
```

Compilation for templates

We put **class definition**

and **class implementation**

into two separate files (.h and .cpp).

Of course we can put them together for class templates.

This is because some compliers cannot compile them for templates separately.

Default type

```
template<typename T = int>
class Stack
{
   ...
};

Stack<> stack;
```

Nontype parameters

We can use nontype parameters along with type parameters in a template prefix.

```
template<typename T, int capacity>
class Stack {
  ...
private:
  T elements[capacity]; // capacity is a constant
  int size;
};
```

Static members in templates

Static data fields are created for each class when a template is used.

In other words, a class created by a template has their own data fields.

```
template<typename T, int capacity>
class Stack {
  ...
private:
  static int c;
  T elements[capacity];
  int size;
};
int Stack<double>::c = 0;
Stack<double> x;
Stack<int> y;
```

Static members in templates

Static data fields are created for each class when a template is used.

In other words, a class created by a template has their own data fields.

```
template<typename T, int capacity>
class Stack {
  ...
private:
  static int c;
  T elements[capacity];
  int size;
};
int Stack<double>::c = 0;
Stack<double> x;
Stack<int> y;
```

Dynamic memory allocation

Avoid using arrays of fixed size.

Allocate memory space when necessary.

```
template<typename T, int capacity>
class Stack {
public:
  Stack( ) {
      this->capacity = capacity;
      elements = new T[capacity];
      size = 0;
  }
private:
  T *elements;
  int capacity;
  int size;
};
Stack<double, 99> x;
```

Dynamic memory allocation: Debug

```
template<Typename T, int capacity>
class Stack {
public:
  Stack( int capacity ) {
      capacity = this->capacity;
      elements = new T[capacity];
      size = 0;
  }
private:
  T *element;
  int capacity;
  int size;
};
Stack<double> x( 99 );
```

Dynamic memory allocation: Debug

```
template<Typename T, int capacity>
class Stack {
public:
  Stack( int capacity ) {
      capacity = this->capacity;
      elements = new T[capacity];
      size = 0;
  }
private:
  T *element;
  int capacity;   // maximum number of elements
  int size;       // the current size
};
stack<double> x( 99 );
```

Dynamic memory allocation: Debug

```
template<typename T, int capacity>
class Stack {
public:
  Stack( int capacity ) {
      capacity = this->capacity;
      elements = new T[capacity];
      size = 0;
  }
private:
  T *element;
  int capacity;
  int size;
};
stack<double> x( 99 );
```

```
template<typename T, int capacity>
class Stack {
public:
  Stack( ) {
      this->capacity = capacity;
      elements = new T[capacity];
      size = 0;
  }
private:
  T *elements;
  int capacity;
  int size;
};
Stack<double, 99> x;
```

The vector Class

A vector is a resizable array.

```
template<typename T, int capacity>
class Stack {
public:
  Stack( ) {
      this->capacity = capacity;
      elements.resize(capacity);
      size = 0;
  }
private:
  vector<T> elements;
  int capacity;
  int size;
};
Stack<double, 99> x;
```

Processing an expression

Given an expression:

5+(4+5)*7 + 5 – 7/8*4

How do we evaluate an expression?

Develop a program that uses a stack data structure to evaluate the expression from left to right.

Processing an expression

Implement a program to evaluate an expression.

Given an expression:

```
5+(4+5)*7 + 5 - 7/8*4
```

How do we evaluate an expression?

Use a stack data structure.

Push elements and pop them if necessary

Templates and Inheritance

NT: A non-template class

TS: A class template specialization.

We can derive a new class by NT and TS together.

```
template<typename G, int n>
class XZ: Stack<G, n> {
public:
   XZ( ) {
      cout << "Ctor: XZ“ << endl;
   }
};
```

```
template<typename T, int capacity>
class Stack : Shape {
public:

  Stack( ) {
     cout << "Ctor: Stack" << endl;
  }
};

class Z : Stack<double, 100> {
public:
    Z( ) {
      cout << "Ctor: Z" << endl;
    }
};
```

Templates and Inheritance

NT: A non-template class

TS: A class template specialization.

We can derive a new class by NT and TS together.

```
template<typename G, int n>
class XZ: Stack<G, n> {
public:
   XZ( ) {
      cout << "Ctor: XZ" << endl;
   }
};
```

```
template<typename T, int capacity>
class Stack : Shape {
public:

  Stack( ) {
     cout << "Ctor: Stack" << endl;
  }
};

class Z : Stack<double, 100> {
public:
    Z( ) {
      cout << "Ctor: Z" << endl;
    }
};
```

Templates and Inheritance

```
template<typename T, int capacity>
class Stack : Shape {
public:

  Stack( ) {
     cout << "Ctor: Stack:"
       << capacity << endl;
  }
};

class Z : Stack<double, 100> {
public:
    Z( ) {
      cout << "Ctor: Z" << endl;
    }
};

The Shape no-arg constructor does not print
anything.
```

```
template<typename G, int n>
class XZ: Stack<G, n-1> {
public:
    XZ( ) {
       cout << "Ctor: XZ" << endl;
    }
};
void main () {
    Stack<double, 99> x;
    Z y;
    XZ<float, 22> xz;
}

What are the output?
```

Templates and Inheritance

```
template<typename T, int capacity>
class Stack : Shape {
public:

  Stack( ) {
     cout << "Ctor: Stack:"
       << capacity << endl;
  }
};

class Z : Stack<double, 100> {
public:
    Z( ) {
      cout << "Ctor: Z" << endl;
    }
};

The Shape no-arg constructor does not print
anything.
```

```
template<typename G, int n>
class XZ: Stack<G, n-1> {
public:
    XZ( ) {
       cout << "Ctor: XZ" << endl;
    }
};
void main () {
    Stack<double, 99> x;
    Z y;
    XZ<float, 22> xz;
}

What are the output?
```

Templates and Inheritance

```
template<typename T, int capacity>
class Stack : Shape {
public:

  Stack( ) {
     cout << "Ctor: Stack:"
       << capacity << endl;
  }
};

class Z : Stack<double, 100> {
public:
    Z( ) {
      cout << "Ctor: Z" << endl;
    }
};
```

The Shape no-arg constructor does not print anything.


```
template<typename G, int n>
class XZ: Stack<G, n-1> {
public:
    XZ( ) {
       cout << "Ctor: XZ" << endl;
    }
};
void main () {
    Stack<double, 99> x;
    Z y;
    XZ<float, 22> xz;
}
```

What are the output?

A1

A2

A3

A4

A5

Templates and Inheritance

```
template<typename T, int capacity>
class Stack : Shape {
public:

  Stack( ) {
     cout << "Ctor: Stack:"
       << capacity << endl;
  }
};

class Z : Stack<double, 100> {
public:
    Z( ) {
      cout << "Ctor: Z" << endl;
    }
};

The Shape no-arg constructor does not print
anything.
```



```
template<typename G, int n>
class XZ: Stack<G, n-1> {
public:
    XZ( ) {
       cout << "Ctor: XZ" << endl;
    }
};
void main () {
    Stack<double, 99> x;
    Z y;
    XZ<float, 22> xz;
}

What are the output?
Ctor: Stack:99
Ctor: Stack:100
Ctor: Z
Ctor: Stack:21
Ctor: XZ
```

Intended Learning Outcomes

- Use the base class to create derived classes
- Distinguish between the base class and derived class
- List the properties of the data fields with priviate, protected, and public attributes
- Describe the process of invoking constructors when an object is created
- Describe the process of invoking destructors when an objectis destroyed
- Define an abstract class and an abstract function
- Use "virtual" to declare a function of a class
- Distinguish between static casting and dynamic casting
- Define a template class

Supplemental Materials

Static vs. Dynamic Casting

dynamic_cast can be performed only on the pointer or a reference of a polymorphic type; i.e., the type contains a virtual function.

dynamic_cast can be used to check whether casting is performed successfully **at runtime**.

static_cast can be performed **at compile time**. No run-time type check is performed. Can lead to problems when pointers are used.

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;    // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double p;             // perimeter
  double len;           // side length
  void initData( ){ len=1.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
  displayShapeInfo( SQUARE() );
  displayShapeInfo( CIRCLE() );
  SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
  CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

```
//show: display the parameter(s).
```

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;    // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double p;             // perimeter
  double len;           // side length
  void initData( ){ len=1.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
  displayShapeInfo( SQUARE() );
  displayShapeInfo( CIRCLE() );
  SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
  CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

```
//show: display the parameter(s).
```

Are there any compilation errors?

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;    // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double p;             // perimeter
  double len;           // side length
  void initData( ){ len=1.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
  displayShapeInfo( SQUARE() );
  displayShapeInfo( CIRCLE() );
  SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
  CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

```
//show: display the parameter(s).
```

Call `checkShapeInfo()`.

What are the output?

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;     // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double p;              // perimeter
  double len;            // side length
  void initData( ){ len=1.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
  displayShapeInfo( SQUARE() );
  displayShapeInfo( CIRCLE() );
  SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
  CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

```
SH: no-arg Ctor
SQ: no-arg Ctor
SQ:1
CI:4
SH: no-arg Ctor
C:I no-arg Ctor
SQ:6.28
CI:1
SH: no-arg Ctor
SQ: no-arg Ctor
SQ:1
CI:4
SH: no-arg Ctor
CI: no-arg Ctor
SQ:6.28
CI:1
```

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;    // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double p;              // perimeter
  double len;            // side length
  void initData( ){ len=1.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
A displayShapeInfo( SQUARE() );
B displayShapeInfo( CIRCLE() );
C SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
D CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

```
SH: no-arg Ctor
SQ: no-arg Ctor
SQ:1
CI:4
SH: no-arg Ctor
C:I no-arg Ctor
SQ:6.28
CI:1
SH: no-arg Ctor
SQ: no-arg Ctor
SQ:1
CI:4
SH: no-arg Ctor
CI: no-arg Ctor
SQ:6.28
CI:1
```

A

B

```
SQUARE {
  p = 4;
  len = 1;
}
```

```
CIRCLE {
  r = 1;
  p = 6.28;
}
```

There is no r in the square object!

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;    // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double p;              // perimeter
  double len;            // side length
  void initData( ){ len=1.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
A displayShapeInfo( SQUARE() );
B displayShapeInfo( CIRCLE() );
C SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
D CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

```
SH: no-arg Ctor
SQ: no-arg Ctor
SQ:1
CI:4
SH: no-arg Ctor
C:I no-arg Ctor
SQ:6.28
CI:1
SH: no-arg Ctor
SQ: no-arg Ctor
SQ:1
CI:4
SH: no-arg Ctor
CI: no-arg Ctor
SQ:6.28
CI:1
```

A

B

```
SQUARE {
  p = 4;
  len = 1;
}
```

```
CIRCLE {
  r = 1;
  p = 6.28;
}
```

There is no len in the circle object!

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;    // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double p;              // perimeter
  double len;            // side length
  void initData( ){ len=1.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
A displayShapeInfo( SQUARE() );
B displayShapeInfo( CIRCLE() );
C SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
D CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

```
SH: no-arg Ctor
SQ: no-arg Ctor
SQ:1
CI:4
SH: no-arg Ctor
C:I no-arg Ctor
SQ:6.28
CI:1
SH: no-arg Ctor
SQ: no-arg Ctor
SQ:1
CI:4
SH: no-arg Ctor
CI: no-arg Ctor
SQ:6.28
CI:1
```

A

B

C

D

Although we do not have any compilation error for static_cast, the results are not what we want.

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;    // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double p;              // perimeter
  double len;            // side length
  void initData( ){ len=2.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
A displayShapeInfo( SQUARE() );
B displayShapeInfo( CIRCLE() );
C SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
D CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

A

B

C

D

Call `checkShapeInfo()`.
What are the output?

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;    // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double p;             // perimeter
  double len;           // side length
  void initData( ){ len=2.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
A displayShapeInfo( SQUARE() );
B displayShapeInfo( CIRCLE() );
C SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
D CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

A

B

C

D

Call `checkShapeInfo()`.
What are the output?

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;    // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double p;              // perimeter
  double len;            // side length
  void initData( ){ len=2.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
A displayShapeInfo( SQUARE() );
B displayShapeInfo( CIRCLE() );
C SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
D CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

```
SH: no-arg Ctor
SQ: no-arg Ctor
SQ:2
CI:8
SH: no-arg Ctor
CI: no-arg Ctor
SQ:6.28
CI:1
```

A

B

C

D

```
SQUARE {
  p = 8;
  len = 2;
}
```

```
CIRCLE {
  r = 1;
  p = 6.28;
}
```

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;    // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double p;              // perimeter
  double len;            // side length
  void initData( ){ len=2.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
A displayShapeInfo( SQUARE() );
B displayShapeInfo( CIRCLE() );
C SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
D CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

```
SH: no-arg Ctor
SQ: no-arg Ctor
SQ:2
CI:8
SH: no-arg Ctor
CI: no-arg Ctor
SQ:6.28
CI:1
SH: no-arg Ctor
SQ: no-arg Ctor
SQ:2
CI:8
SH: no-arg Ctor
CI: no-arg Ctor
SQ:6.28
CI:1
```

A

B

C

D

```
SQUARE {
  p = 8;
  len = 2;
}
```

```
CIRCLE {
  r = 1;
  p = 6.28;
}
```

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;    // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double p;              // perimeter
  double len;            // side length
  void initData( ){ len=2.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
A displayShapeInfo( SQUARE() );
B displayShapeInfo( CIRCLE() );
C SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
D CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

```
SH: no-arg Ctor
SQ: no-arg Ctor
SQ:2
CI:8
SH: no-arg Ctor
CI: no-arg Ctor
SQ:6.28
CI:1
SH: no-arg Ctor
SQ: no-arg Ctor
SQ:2
CI:8
SH: no-arg Ctor
CI: no-arg Ctor
SQ:6.28
CI:1
```

A

B

C

D

Do we want such results?

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;    // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double len;
  double p;
  void initData( ){ len=2.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
A displayShapeInfo( SQUARE() );
B displayShapeInfo( CIRCLE() );
C SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
D CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

A

B

C

D

Call `checkShapeInfo()`.
What are the output?

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  double r;    // radius;
  double p;    // perimeter
  void initData( ) { r = 1; p = 2*r*3.14;}
  CIRCLE() {
     show("CI: no-arg Ctor");
     initData( );
  }
  double getR() const { return r; }
};
```

```
class SQUARE : public SHAPE {
public:
  double len;
  double p;
  void initData( ){ len=2.0; p = 4*len;}
  SQUARE() { show("SQ: no-arg Ctor");
    initData( );
  }
  double getL( ) const { return len; }
};
```

```
void displayShapeInfo( SHAPE &g ) {
  show("SQ:", static_cast<SQUARE*>(&g)->getL());
  show("CI:", static_cast<CIRCLE*>(&g)->getR());
}
void checkShapeInfo( ) {
A displayShapeInfo( SQUARE() );
B displayShapeInfo( CIRCLE() );
C SQUARE y; displayShapeInfo( *((SHAPE*)(&y)) );
D CIRCLE x; displayShapeInfo( *((SHAPE*)(&x)) );
}
```

SH: no-arg Ctor SQ: no-arg Ctor SQ:2 CI:2	A
SH: no-arg Ctor CI: no-arg Ctor SQ:1 CI:1	B
SH: no-arg Ctor SQ: no-arg Ctor SQ:2 CI:2	C
SH: no-arg Ctor CI: no-arg Ctor SQ:1 CI:1	D

previous

SH: no-arg Ctor SQ: no-arg Ctor SQ:2 CI:8
SH: no-arg Ctor CI: no-arg Ctor SQ:6.28 CI:1
SH: no-arg Ctor SQ: no-arg Ctor SQ:2 CI:8
SH: no-arg Ctor CI: no-arg Ctor SQ:6.28 CI:1

Exercise: Inheritance

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  CIRCLE() { name = "C";
     show("CI: no-arg Ctor"); }
  CIRCLE(const string &name) {
      show("CI: Ctor"); this->name = name;
  }
};
class SQUARE : public SHAPE {
public:
  SQUARE() { show("SQ: no-arg Ctor");
    name = "Na";
  }
  SQUARE(const string &name) {
      show("SQ: Ctor"); this->name = name;
  }
};
```

```
void show(const string &msg)
{
    cout << msg << endl;
}
void show(const string &msg, double v)
{
    cout << msg << v << endl;
}

void checkShapeInfo( ) {
    SQUARE x('x');
    SQUARE('a');
    CIRCLE('b');
    CIRCLE y('y');
}
```

Exercise: Inheritance

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  CIRCLE() { name = "C";
     show("CI: no-arg Ctor"); }
  CIRCLE(const string &name) {
      show("CI: Ctor"); this->name = name;
  }
};
class SQUARE : public SHAPE {
public:
  SQUARE() { show("SQ: no-arg Ctor");
    name = "Na";
  }
  SQUARE(const string &name) {
      show("SQ: Ctor"); this->name = name;
  }
};
```

```
void show(const string &msg)
{
    cout << msg << endl;
}
void show(const string &msg, double v)
{
    cout << msg << v << endl;
}

void checkShapeInfo( ) {
    SQUARE x('x');
    SQUARE('a');
    CIRCLE('b');
    CIRCLE y('y');
}
```

Exercise: Inheritance

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  CIRCLE() { name = "C";
     show("CI: no-arg Ctor"); }
  CIRCLE(const string &name) {
      show("CI: Ctor"); this->name = name;
  }
};
class SQUARE : public SHAPE {
public:
  SQUARE() { show("SQ: no-arg Ctor");
    name = "Na";
  }
  SQUARE(const string &name) {
      show("SQ: Ctor"); this->name = name;
  }
};
```

```
void show(const string &msg)
{
    cout << msg << endl;
}
void show(const string &msg, double v)
{
    cout << msg << v << endl;
}

void checkShapeInfo( ) {
    SQUARE x('x');
    SQUARE('a');
    CIRCLE('b');
    CIRCLE y('y');
}
```

Are there any compilation errors?

Exercise: Inheritance

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  CIRCLE() { name = "C";
     show("CI: no-arg Ctor"); }
  CIRCLE(const string &name) {
      show("CI: Ctor"); this->name = name;
  }
};
class SQUARE : public SHAPE {
public:
  SQUARE() { show("SQ: no-arg Ctor");
    name = "Na";
  }
  SQUARE(const string &name) {
      show("SQ: Ctor"); this->name = name;
  }
};
```

```
void checkShapeInfo( ) {
    SQUARE x('x');
    SQUARE('a');
    CIRCLE('b');
    CIRCLE y('y');
}
```

```
void checkShapeInfo( ) {
    SQUARE x("x");
    SQUARE("a");
    CIRCLE("b");
    CIRCLE y("y");
}
```

Fixed the errors.
What are the output?

Exercise: Inheritance

```
class SHAPE {
public:
  string name;
  SHAPE() { show("SH: no-arg Ctor"); }
};
class CIRCLE : public SHAPE {
public:
  CIRCLE() { name = "C";
     show("CI: no-arg Ctor"); }
  CIRCLE(const string &name) {
      show("CI: Ctor"); this->name = name;
  }
};
class SQUARE : public SHAPE {
public:
  SQUARE() { show("SQ: no-arg Ctor");
    name = "Na";
  }
  SQUARE(const string &name) {
      show("SQ: Ctor"); this->name = name;
  }
};
```

```
void show(const string &msg)
{
    cout << msg << endl;
}
void show(const string &msg, double v)
{
    cout << msg << v << endl;
}

void checkShapeInfo( ) {
    SQUARE x("x");
    SQUARE("a");
    CIRCLE("b");
    CIRCLE y("y");
}
```

SH: no-arg Ctor
SQ: Ctor
SH: no-arg Ctor
SQ: Ctor
SH: no-arg Ctor
CI: Ctor
SH: no-arg Ctor
CI: Ctor

```
class SHAPE {
public: string name;
   SHAPE() { show("SH: no-arg Ctor"); }
   SHAPE(const string &name) { show("SH: Ctor"); }
   ~SHAPE() { show("SH: Dtor");}     };

class CIRCLE : public SHAPE {
public:
   CIRCLE() { name = "C";
      show("CI: no-arg Ctor"); }
   CIRCLE(const string &name) {
      show("CI: Ctor"); this->name = name;
   }
   ~CIRCLE() { show("CI: Dtor");}    };

class SQUARE : public SHAPE {
public:
   SQUARE() { show("SQ: no-arg Ctor");
     name = "Na";
   }
   SQUARE(const string &name) {
       show("SQ: Ctor"); this->name = name;
   }
   ~SQUARE() { show("SQ: Dtor");}    };
```

Exercise 2

```
void show(const string &msg)
{
    cout << msg << endl;
}
void show(const string &msg, double v)
{
    cout << msg << v << endl;
}

void checkShapeInfo( ) {
    SQUARE x();
    SQUARE('a');
    CIRCLE("b");
    CIRCLE y( );
}
```

```
class SHAPE {
public: string name;
   SHAPE() { show("SH: no-arg Ctor"); }
   SHAPE(const string &name) { show("SH: Ctor"); }
   ~SHAPE() { show("SH: Dtor");}     };

class CIRCLE : public SHAPE {
public:
   CIRCLE() { name = "C";
      show("CI: no-arg Ctor"); }
   CIRCLE(const string &name) {
      show("CI: Ctor"); this->name = name;
   }
   ~CIRCLE() { show("CI: Dtor");}    };

class SQUARE : public SHAPE {
public:
   SQUARE() { show("SQ: no-arg Ctor");
     name = "Na";
   }
   SQUARE(const string &name) {
       show("SQ: Ctor"); this->name = name;
   }
   ~SQUARE() { show("SQ: Dtor");}    };
```

Exercise 2

```
void show(const string &msg)
{
    cout << msg << endl;
}
void show(const string &msg, double v)
{
    cout << msg << v << endl;
}

void checkShapeInfo( ) {
    SQUARE x();
    SQUARE('a');
    CIRCLE("b");
    CIRCLE y( );
}
```

```
class SHAPE {
public: string name;
   SHAPE() { show("SH: no-arg Ctor"); }
   SHAPE(const string &name) { show("SH: Ctor"); }
   ~SHAPE() { show("SH: Dtor");}     };

class CIRCLE : public SHAPE {
public:
   CIRCLE() { name = "C";
      show("CI: no-arg Ctor"); }
   CIRCLE(const string &name) {
      show("CI: Ctor"); this->name = name;
   }
   ~CIRCLE() { show("CI: Dtor");}    };

class SQUARE : public SHAPE {
public:
   SQUARE() { show("SQ: no-arg Ctor");
     name = "Na";
   }
   SQUARE(const string &name) {
       show("SQ: Ctor"); this->name = name;
   }
   ~SQUARE() { show("SQ: Dtor");}    };
```

Exercise 2

```
void show(const string &msg)
{
    cout << msg << endl;
}
void show(const string &msg, double v)
{
    cout << msg << v << endl;
}

void checkShapeInfo( ) {
    SQUARE x();
    SQUARE('a');
    CIRCLE("b");
    CIRCLE y( );
}
```

Are there any compilation errors?
Fix the errors.

```
class SHAPE {
public: string name;
   SHAPE() { show("SH: no-arg Ctor"); }
   SHAPE(const string &name) { show("SH: Ctor"); }
   ~SHAPE() { show("SH: Dtor");}     };

class CIRCLE : public SHAPE {
public:
   CIRCLE() { name = "C";
      show("CI: no-arg Ctor"); }
   CIRCLE(const string &name) {
      show("CI: Ctor"); this->name = name;
   }
   ~CIRCLE() { show("CI: Dtor");}    };

class SQUARE : public SHAPE {
public:
   SQUARE() { show("SQ: no-arg Ctor");
     name = "Na";
   }
   SQUARE(const string &name) {
       show("SQ: Ctor"); this->name = name;
   }
   ~SQUARE() { show("SQ: Dtor");}    };
```

Exercise 2

```
void checkShapeInfo( ) {
     SQUARE x();
     SQUARE('a');
     CIRCLE("b");
     CIRCLE y( );
}
```

```
void checkShapeInfo( ) {
     SQUARE x;
     SQUARE("a");
     CIRCLE("b");
     CIRCLE y;
}
```

```
class SHAPE {
public: string name;
   SHAPE() { show("SH: no-arg Ctor"); }
   SHAPE(const string &name) { show("SH: Ctor"); }
   ~SHAPE() { show("SH: Dtor");}     };

class CIRCLE : public SHAPE {
public:
   CIRCLE() { name = "C";
      show("CI: no-arg Ctor"); }
   CIRCLE(const string &name) {
      show("CI: Ctor"); this->name = name;
   }
   ~CIRCLE() { show("CI: Dtor");}    };

class SQUARE : public SHAPE {
public:
   SQUARE() { show("SQ: no-arg Ctor");
     name = "Na";
   }
   SQUARE(const string &name) {
       show("SQ: Ctor"); this->name = name;
   }
   ~SQUARE() { show("SQ: Dtor");}    };
```

Exercise 2

```
void checkShapeInfo( ) {
     SQUARE x();
     SQUARE('a');
     CIRCLE("b");
     CIRCLE y( );
}
```

```
void checkShapeInfo( ) {
     SQUARE x;
     SQUARE("a");
     CIRCLE("b");
     CIRCLE y;
}
```

What are the output?

```
class SHAPE {
public: string name;
   SHAPE() { show("SH: no-arg Ctor"); }
   SHAPE(const string &name) { show("SH: Ctor"); }
   ~SHAPE() { show("SH: Dtor");}      };

class CIRCLE : public SHAPE {
public:
   CIRCLE() { name = "C";
      show("CI: no-arg Ctor"); }
   CIRCLE(const string &name) {
      show("CI: Ctor"); this->name = name;
   }
   ~CIRCLE() { show("CI: Dtor");}    };

class SQUARE : public SHAPE {
public:
   SQUARE() { show("SQ: no-arg Ctor");
     name = "Na";
   }
   SQUARE(const string &name) {
       show("SQ: Ctor"); this->name = name;
   }
   ~SQUARE() { show("SQ: Dtor");}    };
```

Exercise 2

```
void checkShapeInfo( ) {
     SQUARE x;
     SQUARE("a");
     CIRCLE("b");
     CIRCLE y;
}
```

```
SH: no-arg Ctor
SQ: no-arg Ctor
SH: no-arg Ctor
SQ: Ctor
SQ: Dtor
SH: Dtor
SH: no-arg Ctor
CI: Ctor
CI: Dtor
SH: Dtor
SH: no-arg Ctor
CI: no-arg Ctor
CI: Dtor
SH: Dtor
SQ: Dtor
SH: Dtor
```

```
class SHAPE {
public: string name;
   SHAPE() { show("SH: no-arg Ctor"); }
   SHAPE(const string &name) { show("SH: Ctor"); }
   ~SHAPE() { show("SH: Dtor");}      };

class CIRCLE : public SHAPE {
public:
   CIRCLE() { name = "C";
       show("CI: no-arg Ctor"); }
   CIRCLE(const string &name) {
       show("CI: Ctor"); this->name = name;
   }
   ~CIRCLE() { show("CI: Dtor");}    };

class SQUARE : public SHAPE {
public:
   SQUARE() { show("SQ: no-arg Ctor");
     name = "Na";
   }
   SQUARE(const string &name) {
       show("SQ: Ctor"); this->name = name;
   }
   ~SQUARE() { show("SQ: Dtor");}    };
```

Exercise 2

```
void checkShapeInfo( ) {
A    SQUARE x;
B    SQUARE("a");
C    CIRCLE("b");
D    CIRCLE y;
}
```

```
SH: no-arg Ctor
SQ: no-arg Ctor
SH: no-arg Ctor
SQ: Ctor
SQ: Dtor
SH: Dtor
SH: no-arg Ctor
CI: Ctor
CI: Dtor
SH: Dtor
SH: no-arg Ctor
CI: no-arg Ctor
CI: Dtor
SH: Dtor
SQ: Dtor
SH: Dtor
```

A

B

C

D

Finish the function call

```
class SHAPE {
public: string name;
   SHAPE() { show("SH: no-arg Ctor"); }
   SHAPE(const string &name) { show("SH: Ctor"); }
   ~SHAPE() { show("SH: Dtor");}     };

class CIRCLE : public SHAPE {
public:
   CIRCLE() { name = "C";
      show("CI: no-arg Ctor"); }
   CIRCLE(const string &name) {
      show("CI: Ctor"); this->name = name;
   }
   ~CIRCLE() { show("CI: Dtor");}    };

class SQUARE : public SHAPE {
public:
   SQUARE() { show("SQ: no-arg Ctor");
     name = "Na";
   }
   SQUARE(const string &name) {
       show("SQ: Ctor"); this->name = name;
   }
   ~SQUARE() { show("SQ: Dtor");}    };
```

Exercise 2

```
void checkShapeInfo( ) {
A    SQUARE x;
B    SQUARE("a");
C    CIRCLE("b");
D    CIRCLE y;
}
```

```
SH: no-arg Ctor
SQ: no-arg Ctor
SH: no-arg Ctor
SQ: Ctor
SQ: Dtor
SH: Dtor
SH: no-arg Ctor
CI: Ctor
CI: Dtor
SH: Dtor
SH: no-arg Ctor
CI: no-arg Ctor
CI: Dtor
SH: Dtor
SQ: Dtor
SH: Dtor
```

A

B

C

D

Finish the function call

The constructor of SHAPE with an argument is not invoked.

```
class SHAPE {
public: string name;
   SHAPE() { show("SH: no-arg Ctor"); }
   SHAPE(const string &name) { show("SH: Ctor"); }
   ~SHAPE() { show("SH: Dtor");}
};
class CIRCLE : public SHAPE {
public:
   CIRCLE() { name = "C";
      show("CI: no-arg Ctor"); }
   CIRCLE(const string &name) {
      show("CI: Ctor"); this->name = name;
   }
   ~CIRCLE() { show("CI: Dtor");}
};
```

The constructor of SHAPE with an argument is not invoked. **Need to specify which base constructor that we want to invoke.**

Exercise 2

```
class SQUARE : public SHAPE {
public:
   SQUARE() { show("SQ: no-arg Ctor");
     name = "Na";
   }
   SQUARE(const string &name):
       SHAPE( name )
   {
     show("SQ: Ctor");
     this->name = name;
   }
   ~SQUARE() { show("SQ: Dtor"); }
};
```

Exercise: Static vs. Dynamic Casting

```
class Shape{
public:
    Shape () { }
  Shape(int id)  {
  }
  virtual void printf() const {
       cout << "S" << endl;
  }
};

class Rectangle: public Shape{
public:
  Rectangle( ) { }
  Rectangle( int id ) {
  }
  virtual void printf() const {
   cout << "R" << endl;
  }
};
```

```
void displayObj(Shape &g)
{
    Shape *p = &g;
    static_cast<Rectangle*>(p)->printf();
}

class Y {
protected:
    int a;
};

void main () {
    Rectangle x;
    Shape y;
    displayObj(x);
    displayObj(y);
    Y *g = new Y;
    displayObj(*((Shape*)g));
}
```

What are the output?

Exercise: Static vs. Dynamic Casting

```
class Shape{
public:
    Shape () { }
  Shape(int id)  {
  }
  virtual void printf() const {
       cout << "S" << endl;
  }
};

class Rectangle: public Shape{
public:
  Rectangle( ) { }
  Rectangle( int id ) {
  }
  virtual void printf() const {
   cout << "R" << endl;
  }
};
```

```
void displayObj(Shape &g)
{
    Shape *p = &g;
    static_cast<Rectangle*>(p)->printf();
}

class Y {
protected:
    int a;
};

void main () {
    Rectangle x;
    Shape y;
    displayObj(x);
    displayObj(y);
    Y *g = new Y;
    displayObj(*((Shape*)g));
}
```

What are the output?

Type the program and run it.