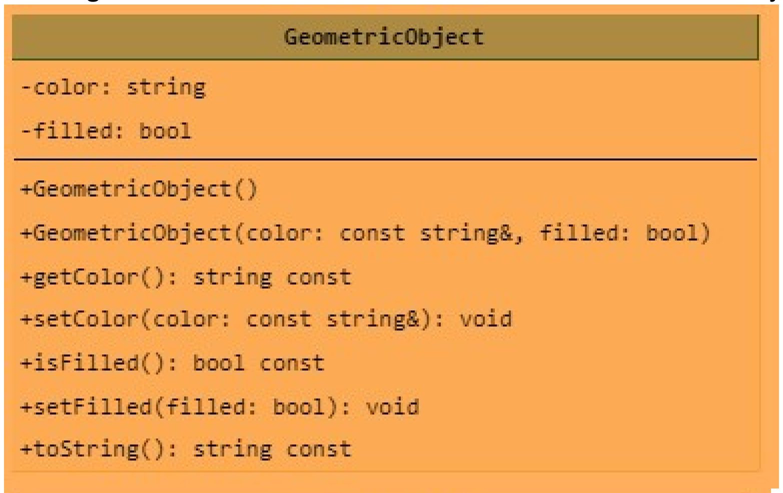
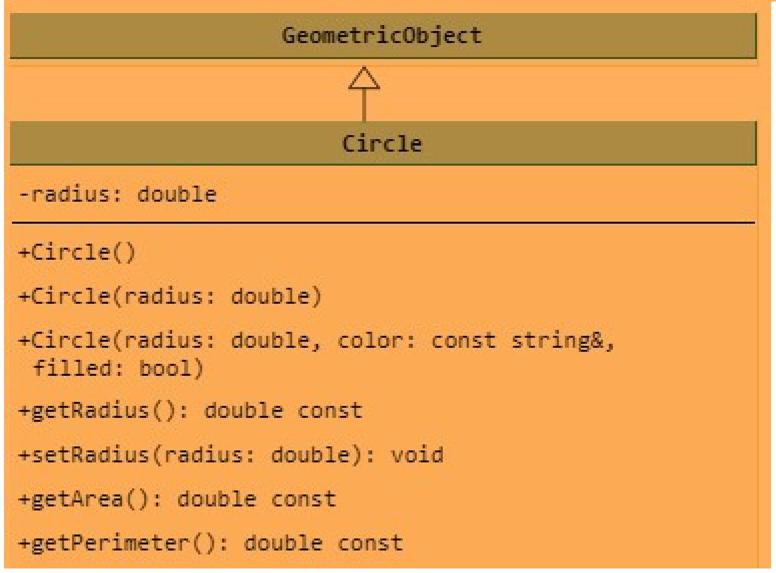
15.2 Base Classes and Derived Classes

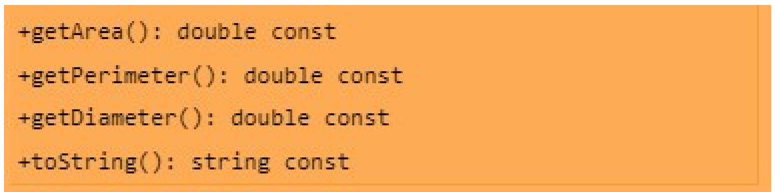
Monday, April 17, 2023 10:15 AM

- Inheritance enables you to define a general class (like a base class), and later extend it to more specialized classes (like derived classes)
- Use a class to model objects of the same type
- Diff classes can have some common properties and behaviors, which can be generalized in a class that can be shared by other classes
- Inheritance lets define general class, later extend to more specialized ones
- Specialized classes inherit properties and fns from the general class
- Think geometric objects
  - If want to design classes to model geometric objects, like circle and rectangle
  - Many common properties
    - Drawn in certain color
    - Filled/unfilled
  - General class GeometricObject can be used to model all geometric objects
  - This class has properties color and filled and their getter and setter fns
  - Also has the toString() fn, returns a string rep for the object
  - Since circle is a special type of geometric object, shares common properties and fns w/ other geometric objects, so extend the Circle class from the GeometricObject class
  - Rectangle can also be defined as a derived class of GeometricObject
  - 

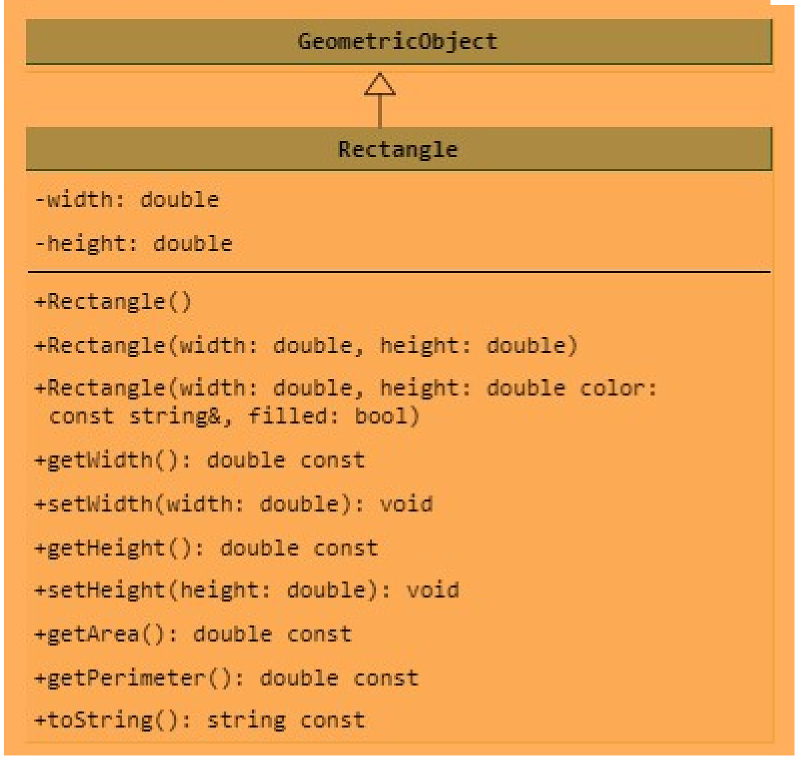
```
GeometricObject
-color: string
-filled: bool
+GeometricObject()
+GeometricObject(color: const string&, filled: bool)
+getColor(): string const
+setColor(color: const string&): void
+isFilled(): bool const
+setFilled(filled: bool): void
+toString(): string const
```

  - 

```
GeometricObject

Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: const string&,
 filled: bool)
+getRadius(): double const
+setRadius(radius: double): void
+getArea(): double const
+getPerimeter(): double const
```

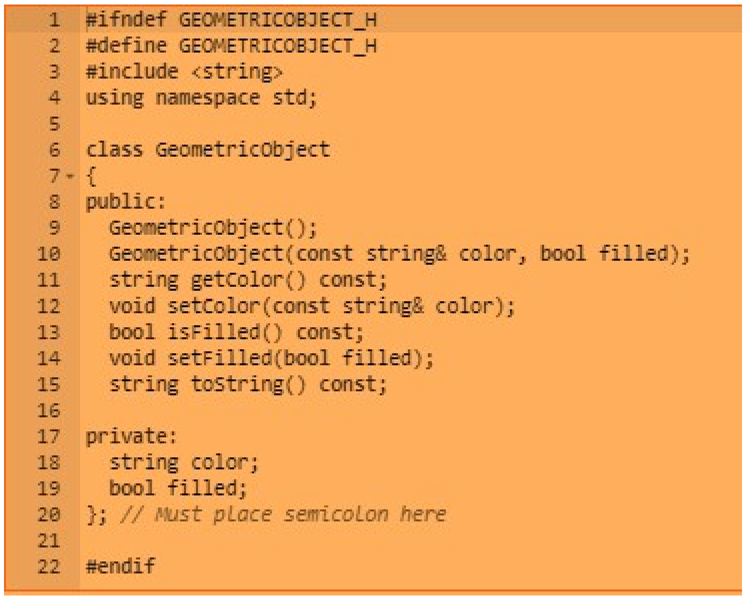


```
+getArea(): double const
+getPerimeter(): double const
+getDiameter(): double const
+toString(): string const
```

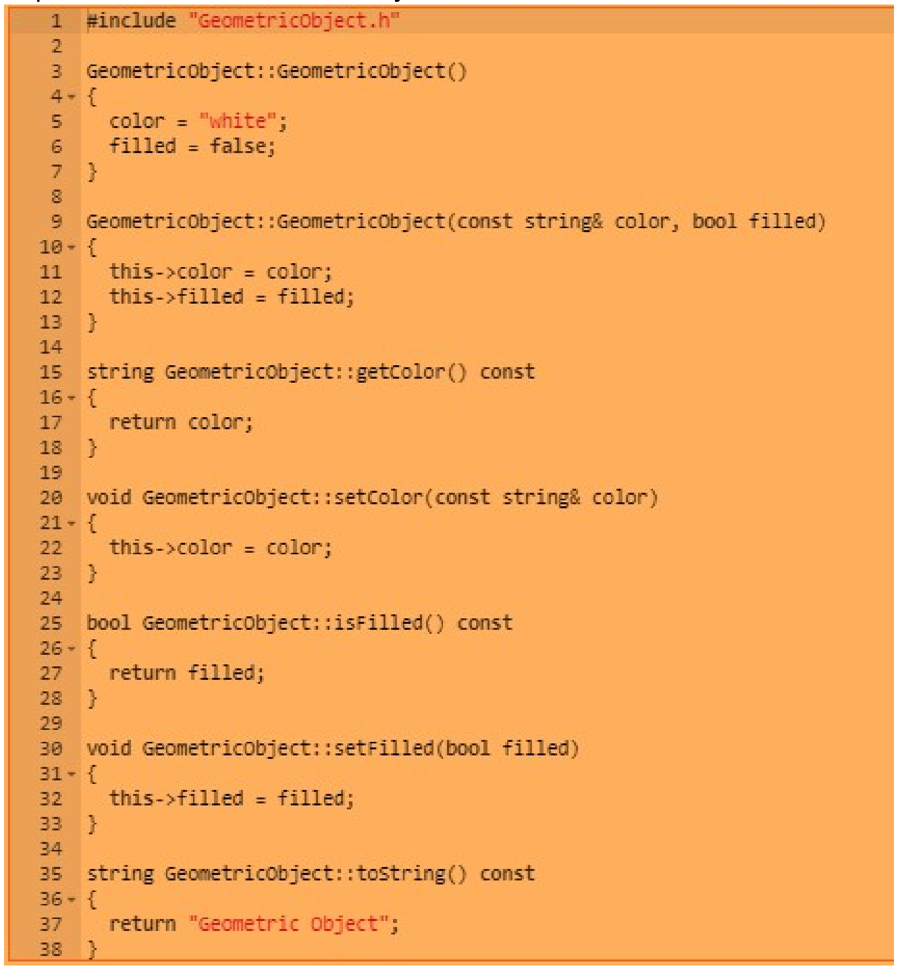
- 

```
GeometricObject
      △
      |
Rectangle
-width: double
-height: double
--------------------------------------------
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color:
 const string&, filled: bool)
+getWidth(): double const
+setWidth(width: double): void
+getHeight(): double const
+setHeight(height: double): void
+getArea(): double const
+getPerimeter(): double const
+toString(): string const
```

- In C++ terms, class C1 extended from another class C2 is called a derived class, and C2 is called base class
- Base class (aka parent class aka superclass), derived class (aka child class aka subclass)
- Derived class inherits accessible data fields from its base class and can also add new data fields and fns
  - The Circle class inherits all accessible data fields and fns from the GeometricObject class
  - Also has new data field radius along w/ its getter and setter fns
  - Also has getArea() , getPerimeter(), and getDiameter() fns for returning area, perimeter, and diameter of the circle
  -
  - The Rectangle class inherits all accessible data fields and fns from the GeometricObject class
  - Also has data fields width and height and their getter and setter fns
  - Also has the getArea() and getPerimeter() fns for returning the area and perimeter of the rectangle
- Class declaration for GeometricObject:

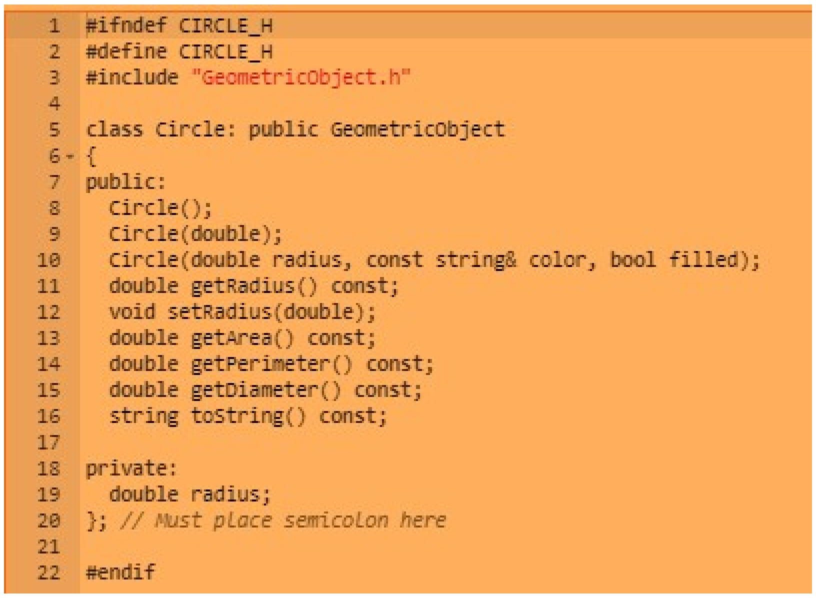
- 

```
1  #ifndef GEOMETRICOBJECT_H
2  #define GEOMETRICOBJECT_H
3  #include <string>
4  using namespace std;
5
6  class GeometricObject
7  {
8  public:
9    GeometricObject();
10   GeometricObject(const string& color, bool filled);
11   string getColor() const;
12   void setColor(const string& color);
13   bool isFilled() const;
14   void setFilled(bool filled);
15   string toString() const;
16
17 private:
18   string color;
19   bool filled;
20 }; // Must place semicolon here
21
22 #endif
```

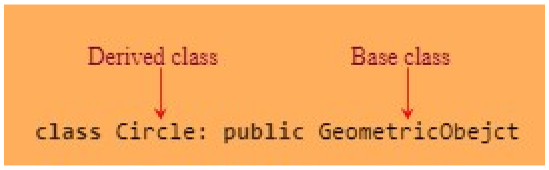
   - Ln 1 & 2 are preprocessor directives, so no multiple declarations
   - C++ string class header included to support the use of the string class in GeometricObject
   - isFilled() fn is accessor for the filed data field, its bool type, accessor fn named isFilled() by convention
- Implementation of GeometricObject class:
- 

```
#include "GeometricObject.h"

GeometricObject::GeometricObject()
{
  color = "white";
  filled = false;
}

GeometricObject::GeometricObject(const string& color, bool filled)
{
  this->color = color;
  this->filled = filled;
}

string GeometricObject::getColor() const
{
  return color;
}

void GeometricObject::setColor(const string& color)
{
  this->color = color;
}

bool GeometricObject::isFilled() const
{
  return filled;
}

void GeometricObject::setFilled(bool filled)
{
  this->filled = filled;
}

string GeometricObject::toString() const
{
  return "Geometric Object";
}
```

   - The toString() fn returns a string that describes the object
   - The string operator + used to concatenate 2 strings and returns a new string object
- Class def for Circle

- 

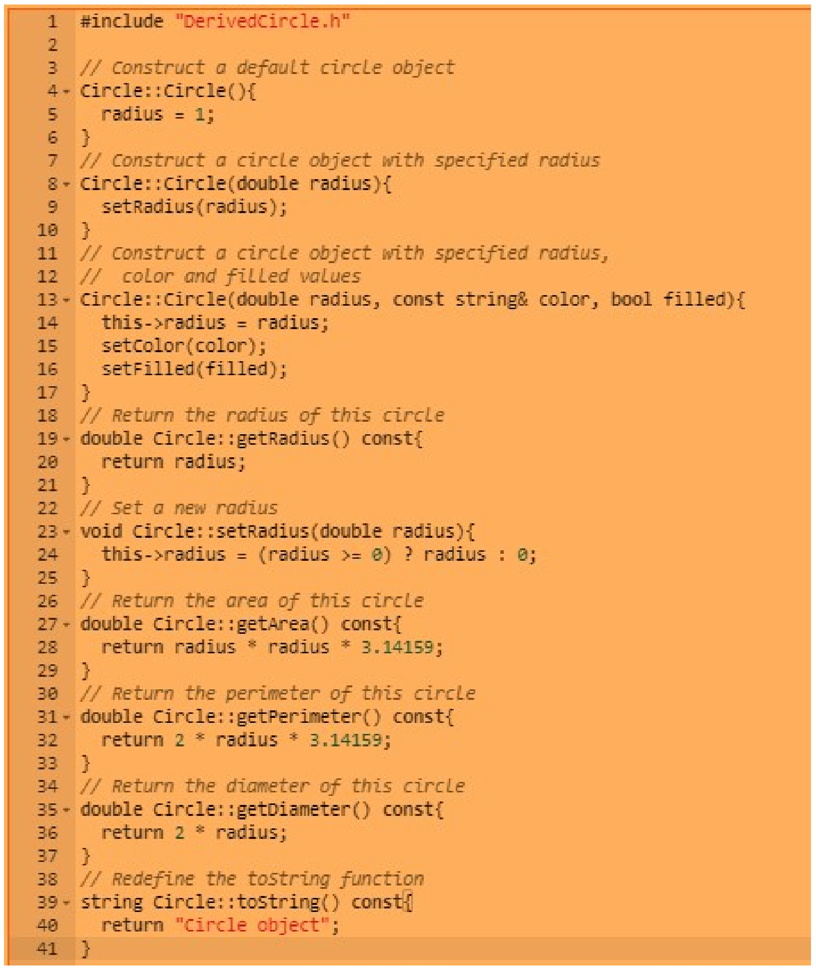
```cpp
1  #ifndef CIRCLE_H
2  #define CIRCLE_H
3  #include "GeometricObject.h"
4
5  class Circle: public GeometricObject
6  {
7  public:
8    Circle();
9    Circle(double);
10   Circle(double radius, const string& color, bool filled);
11   double getRadius() const;
12   void setRadius(double);
13   double getArea() const;
14   double getPerimeter() const;
15   double getDiameter() const;
16   string toString() const;
17
18 private:
19   double radius;
20 }; // Must place semicolon here
21
22 #endif
```

- Ln 5 defines that the Circle class derived from the base class GeometricObject, syntax is
- 

```cpp
class Circle: public GeometricObejct
```

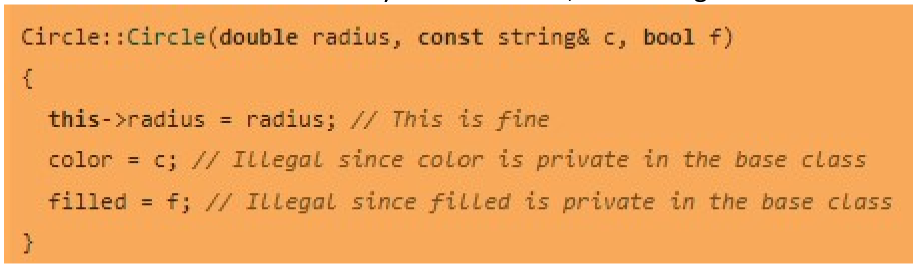
- Tells compiler that the class is derived from the base class, so all public members in GeometricObject are inherited in Circle

• Implementation of Circle class:

- 

```cpp
1  #include "DerivedCircle.h"
2
3  // Construct a default circle object
4  Circle::Circle(){
5    radius = 1;
6  }
7  // Construct a circle object with specified radius
8  Circle::Circle(double radius){
9    setRadius(radius);
10 }
11 // Construct a circle object with specified radius,
12 //  color and filled values
13 Circle::Circle(double radius, const string& color, bool filled){
14   this->radius = radius;
15   setColor(color);
16   setFilled(filled);
17 }
18 // Return the radius of this circle
19 double Circle::getRadius() const{
20   return radius;
21 }
22 // Set a new radius
23 void Circle::setRadius(double radius){
24   this->radius = (radius >= 0) ? radius : 0;
25 }
26 // Return the area of this circle
27 double Circle::getArea() const{
28   return radius * radius * 3.14159;
29 }
30 // Return the perimeter of this circle
31 double Circle::getPerimeter() const{
32   return 2 * radius * 3.14159;
33 }
34 // Return the diameter of this circle
35 double Circle::getDiameter() const{
36   return 2 * radius;
37 }
38 // Redefine the toString function
39 string Circle::toString() const{
40   return "Circle object";
41 }
```

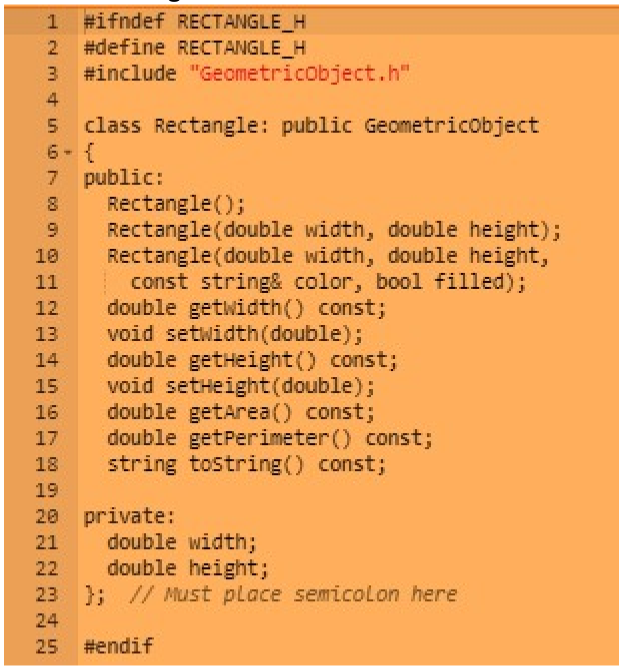
- Constructor Circle(double radius, const string& color, bool filled) is implemented by invoking the setColor and setFilled fns to set the color and filled properties

- These public fns are defined in the base class and inherited in circle, so can be used in derived class
- Can't use color and filled directly in constructor, this wrong
- 

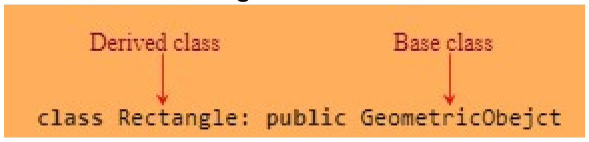
```
Circle::Circle(double radius, const string& c, bool f)
{
  this->radius = radius; // This is fine
  color = c; // Illegal since color is private in the base class
  filled = f; // Illegal since filled is private in the base class
}
```

- Wrong bc private data fields color and filled in the GeometricObject class can't be accessed in any class other than in the GeometricObject class itself, only way to read and mod color and filled is thru their getter and setter fns

• Class def for Rectangle:

- 

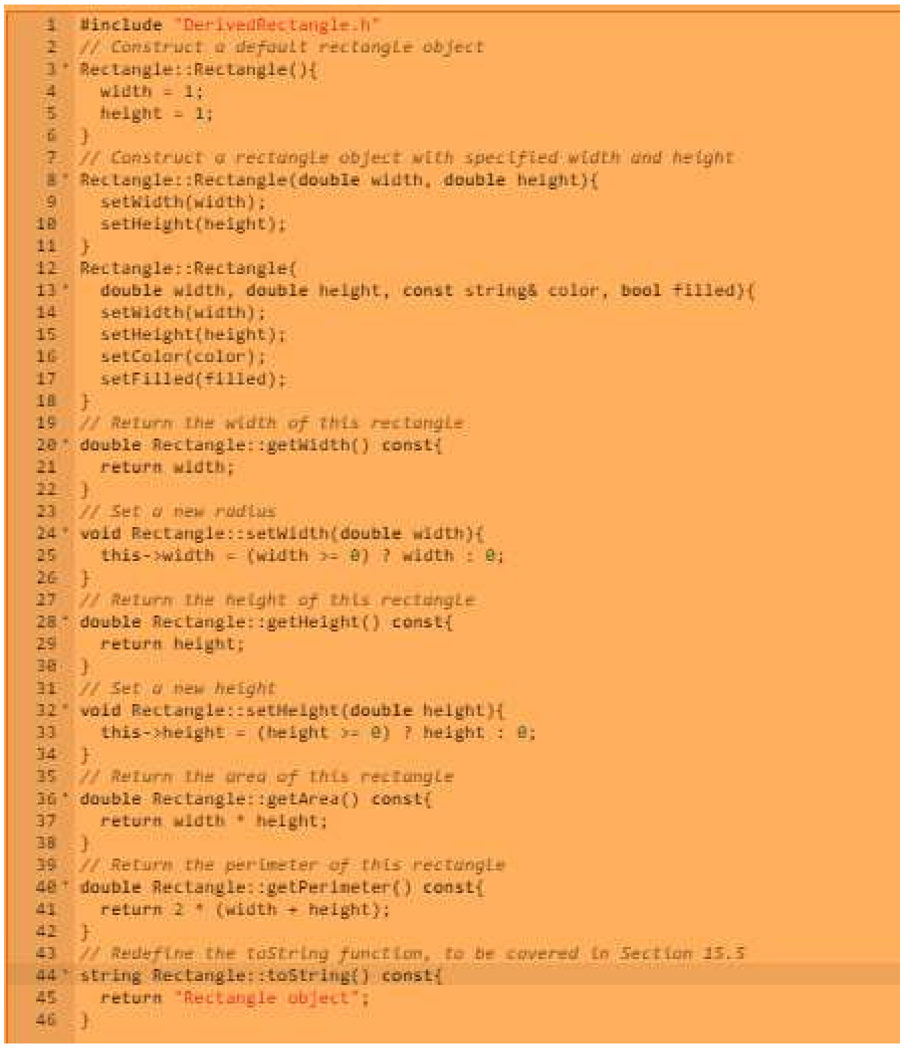
```
 1  #ifndef RECTANGLE_H
 2  #define RECTANGLE_H
 3  #include "GeometricObject.h"
 4
 5  class Rectangle: public GeometricObject
 6  {
 7  public:
 8    Rectangle();
 9    Rectangle(double width, double height);
10    Rectangle(double width, double height,
11      const string& color, bool filled);
12    double getWidth() const;
13    void setWidth(double);
14    double getHeight() const;
15    void setHeight(double);
16    double getArea() const;
17    double getPerimeter() const;
18    string toString() const;
19
20  private:
21    double width;
22    double height;
23  };  // Must place semicolon here
24
25  #endif
```

- Ln 5 defines Rectangle class from base class GeometricObject
- 

```
class Rectangle: public GeometricObejct
```

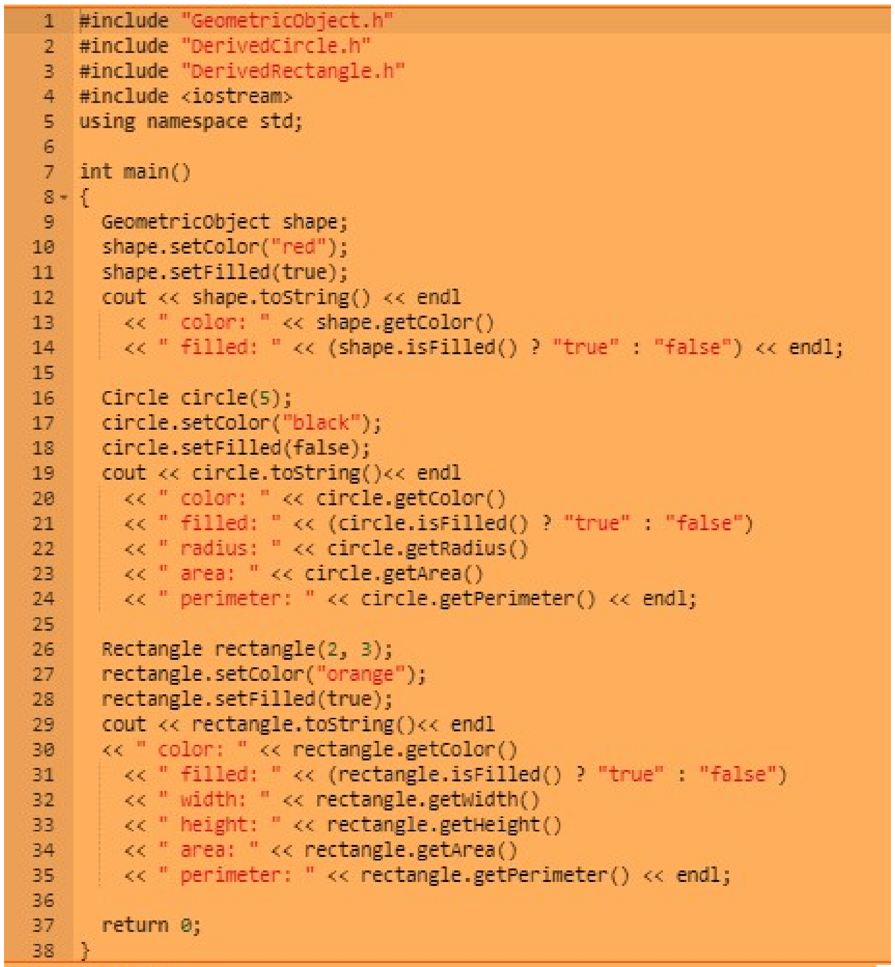
- This syntax tells compiler that the class is derived from the base class, so all public membs in GeometricObject are inherited in Rectangle

• Implementation of Rectangle Class:

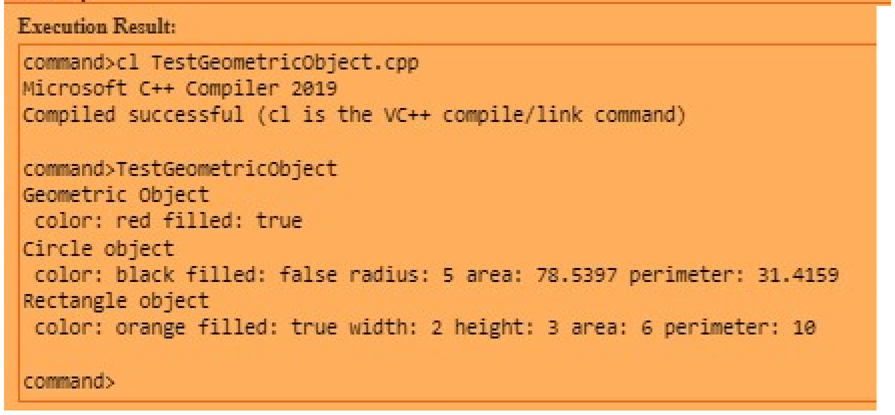


```
#include "DerivedRectangle.h"
// Construct a default rectangle object
Rectangle::Rectangle(){
  width = 1;
  height = 1;
}
// Construct a rectangle object with specified width and height
Rectangle::Rectangle(double width, double height){
  setWidth(width);
  setHeight(height);
}
Rectangle::Rectangle(
  double width, double height, const string& color, bool filled){
  setWidth(width);
  setHeight(height);
  setColor(color);
  setFilled(filled);
}
// Return the width of this rectangle
double Rectangle::getWidth() const{
  return width;
}
// Set a new radius
void Rectangle::setWidth(double width){
  this->width = (width >= 0) ? width : 0;
}
// Return the height of this rectangle
double Rectangle::getHeight() const{
  return height;
}
// Set a new height
void Rectangle::setHeight(double height){
  this->height = (height >= 0) ? height : 0;
}
// Return the area of this rectangle
double Rectangle::getArea() const{
  return width * height;
}
// Return the perimeter of this rectangle
double Rectangle::getPerimeter() const{
  return 2 * (width + height);
}
// Redefine the toString function, to be covered in Section 15.5
string Rectangle::toString() const{
  return "Rectangle object";
}
```

- Test pgrm that uses the three classes- GeometricObject , Circle , Rectangles :

- 

```cpp
#include "GeometricObject.h"
#include "DerivedCircle.h"
#include "DerivedRectangle.h"
#include <iostream>
using namespace std;

int main()
{
  GeometricObject shape;
  shape.setColor("red");
  shape.setFilled(true);
  cout << shape.toString() << endl
    << " color: " << shape.getColor()
    << " filled: " << (shape.isFilled() ? "true" : "false") << endl;

  Circle circle(5);
  circle.setColor("black");
  circle.setFilled(false);
  cout << circle.toString()<< endl
    << " color: " << circle.getColor()
    << " filled: " << (circle.isFilled() ? "true" : "false")
    << " radius: " << circle.getRadius()
    << " area: " << circle.getArea()
    << " perimeter: " << circle.getPerimeter() << endl;

  Rectangle rectangle(2, 3);
  rectangle.setColor("orange");
  rectangle.setFilled(true);
  cout << rectangle.toString()<< endl
  << " color: " << rectangle.getColor()
    << " filled: " << (rectangle.isFilled() ? "true" : "false")
    << " width: " << rectangle.getWidth()
    << " height: " << rectangle.getHeight()
    << " area: " << rectangle.getArea()
    << " perimeter: " << rectangle.getPerimeter() << endl;

  return 0;
}
```

- 

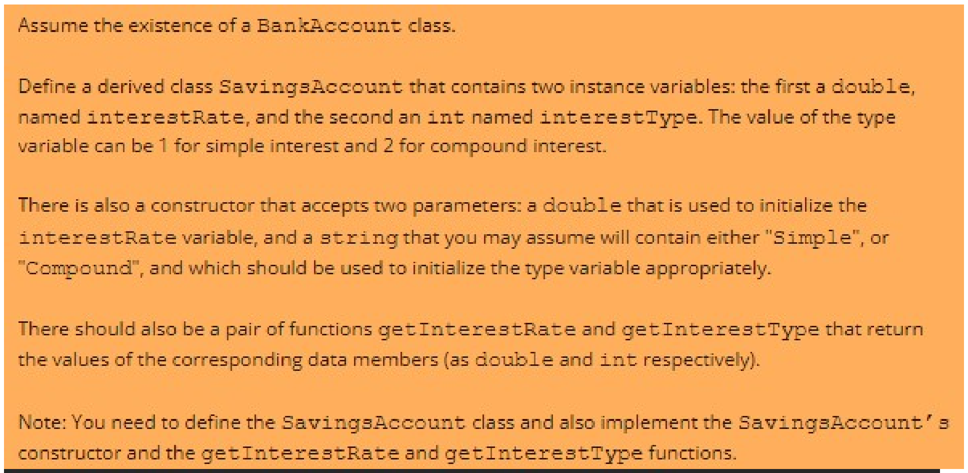
Execution Result:

```
command>cl TestGeometricObject.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>TestGeometricObject
Geometric Object
 color: red filled: true
Circle object
 color: black filled: false radius: 5 area: 78.5397 perimeter: 31.4159
Rectangle object
 color: orange filled: true width: 2 height: 3 area: 6 perimeter: 10

command>
```

- Pgrm makes a GeometricObject and invokes its fns setColor, setFilled, toString, getColor, and isFilled
- Pgrm makes a Circle object, invokes those ^, and also getRadius, getArea, and getPerimeter
- Pgrm makes a Rectangle object and those ^^(2 lns above), and also getHeight, getWidth, getArea, and getPerimeter

• Important notes abt inheritance
  - Private data fields in a base class aren't accessible outside the class, so they can't be used directly in a derived class, can be accessed/mutated thru public accessor/mutator if defined in the base class
  - Not all is-a relationships should be modeled using inheritance, like a square is a rectangle, but should not extend a Rectangle class
  - Inheritance is used to model the is-a relationship, don't rando extend just to reuse fns
  - C++ allow to derive a derived class from many classes, aka multiple inheritance

•

- 

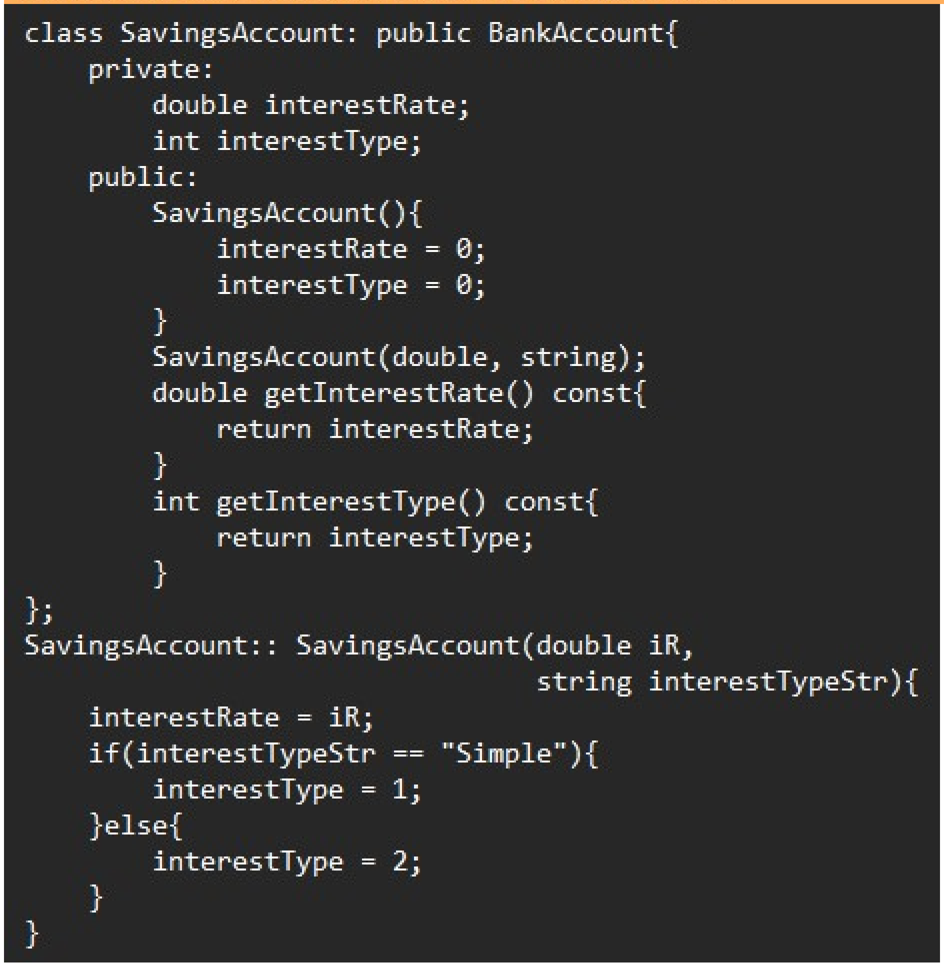
  Assume the existence of a BankAccount class.

  Define a derived class SavingsAccount that contains two instance variables: the first a double, named interestRate, and the second an int named interestType. The value of the type variable can be 1 for simple interest and 2 for compound interest.

  There is also a constructor that accepts two parameters: a double that is used to initialize the interestRate variable, and a string that you may assume will contain either "Simple", or "Compound", and which should be used to initialize the type variable appropriately.

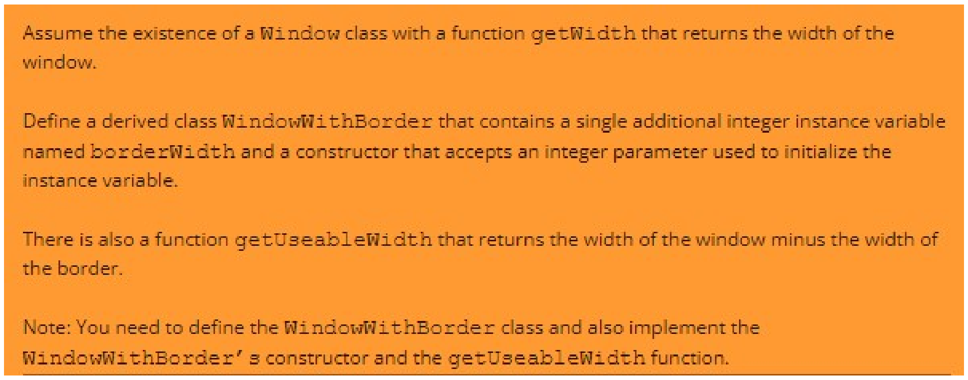
  There should also be a pair of functions getInterestRate and getInterestType that return the values of the corresponding data members (as double and int respectively).

  Note: You need to define the SavingsAccount class and also implement the SavingsAccount's constructor and the getInterestRate and getInterestType functions.

- 

```
class SavingsAccount: public BankAccount{
    private:
        double interestRate;
        int interestType;
    public:
        SavingsAccount(){
            interestRate = 0;
            interestType = 0;
        }
        SavingsAccount(double, string);
        double getInterestRate() const{
            return interestRate;
        }
        int getInterestType() const{
            return interestType;
        }
};
SavingsAccount:: SavingsAccount(double iR,
                                string interestTypeStr){
    interestRate = iR;
    if(interestTypeStr == "Simple"){
        interestType = 1;
    }else{
        interestType = 2;
    }
}
```

-

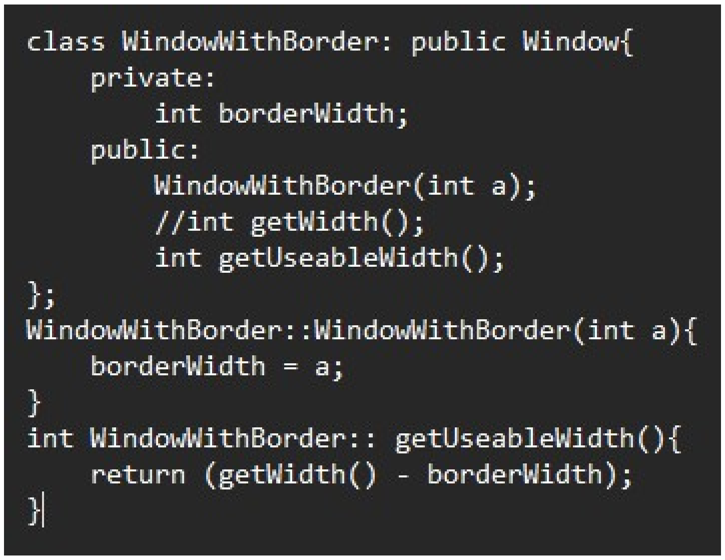
- 

  Assume the existence of a Window class with a function getWidth that returns the width of the window.

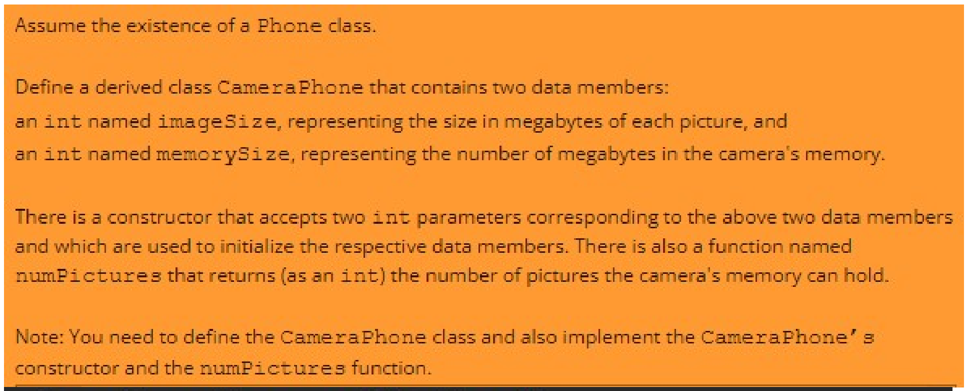
  Define a derived class WindowWithBorder that contains a single additional integer instance variable named borderWidth and a constructor that accepts an integer parameter used to initialize the instance variable.

  There is also a function getUseableWidth that returns the width of the window minus the width of the border.

  Note: You need to define the WindowWithBorder class and also implement the WindowWithBorder's constructor and the getUseableWidth function.

- 

```
class WindowWithBorder: public Window{
    private:
        int borderWidth;
    public:
        WindowWithBorder(int a);
        //int getWidth();
        int getUseableWidth();
};
WindowWithBorder::WindowWithBorder(int a){
    borderWidth = a;
}
int WindowWithBorder:: getUseableWidth(){
    return (getWidth() - borderWidth);
}
```

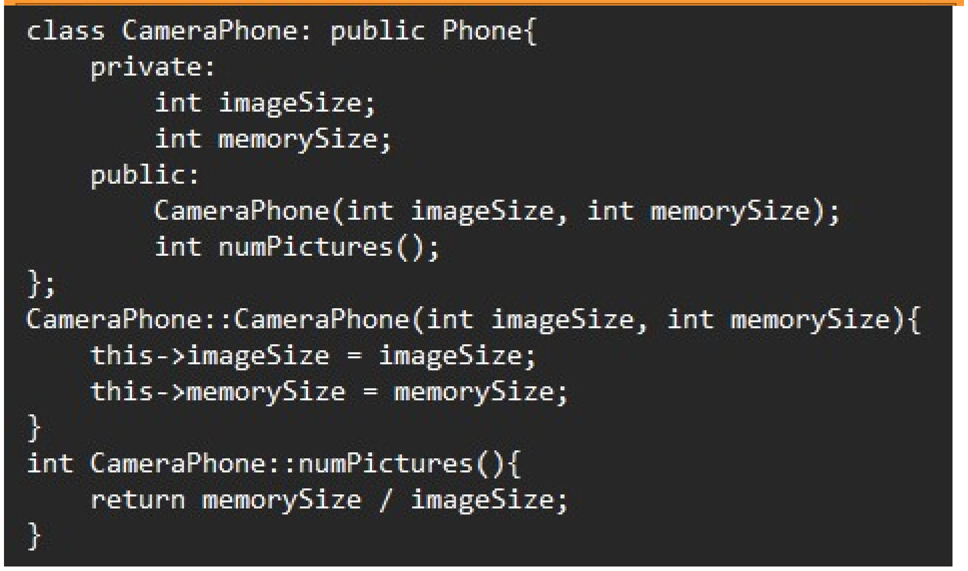
-
- 

Assume the existence of a `Phone` class.

Define a derived class `CameraPhone` that contains two data members:
an `int` named `imageSize`, representing the size in megabytes of each picture, and
an `int` named `memorySize`, representing the number of megabytes in the camera's memory.

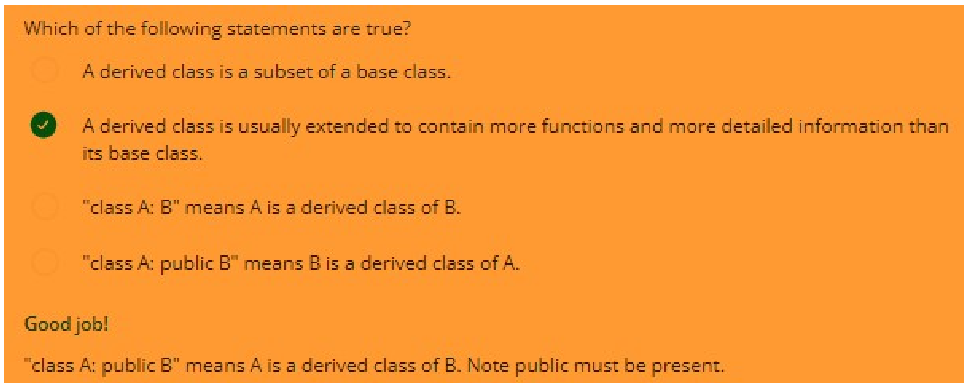
There is a constructor that accepts two `int` parameters corresponding to the above two data members and which are used to initialize the respective data members. There is also a function named `numPictures` that returns (as an `int`) the number of pictures the camera's memory can hold.

Note: You need to define the `CameraPhone` class and also implement the `CameraPhone’s` constructor and the `numPictures` function.

- 

```
class CameraPhone: public Phone{
    private:
        int imageSize;
        int memorySize;
    public:
        CameraPhone(int imageSize, int memorySize);
        int numPictures();
};
CameraPhone::CameraPhone(int imageSize, int memorySize){
    this->imageSize = imageSize;
    this->memorySize = memorySize;
}
int CameraPhone::numPictures(){
    return memorySize / imageSize;
}
```

-

- 
  Which of the following statements are true?

  A derived class is a subset of a base class.

  [✓ A derived class is usually extended to contain more functions and more detailed information than its base class.]

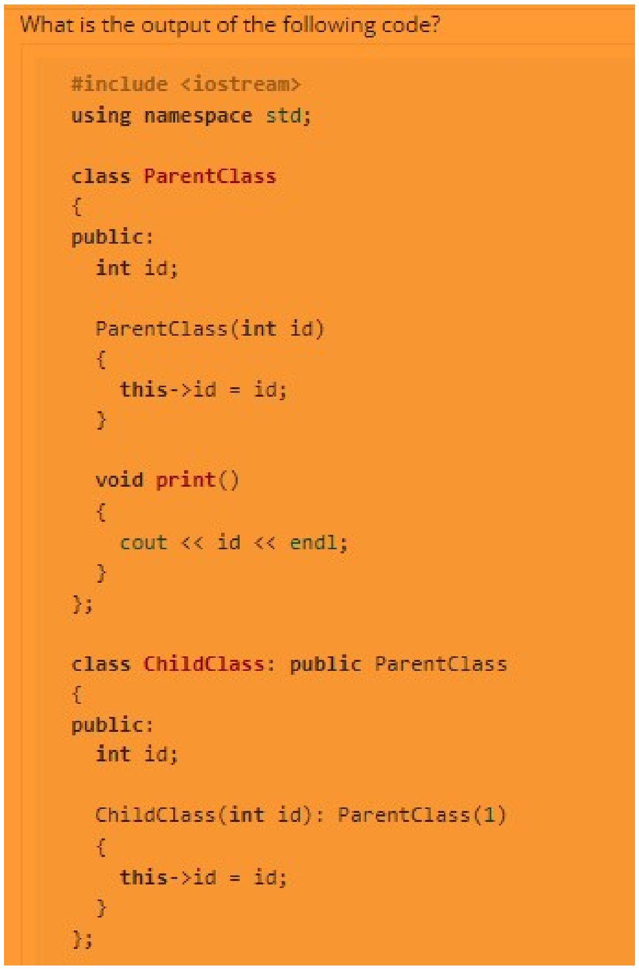
  "class A: B" means A is a derived class of B.

  "class A: public B" means B is a derived class of A.

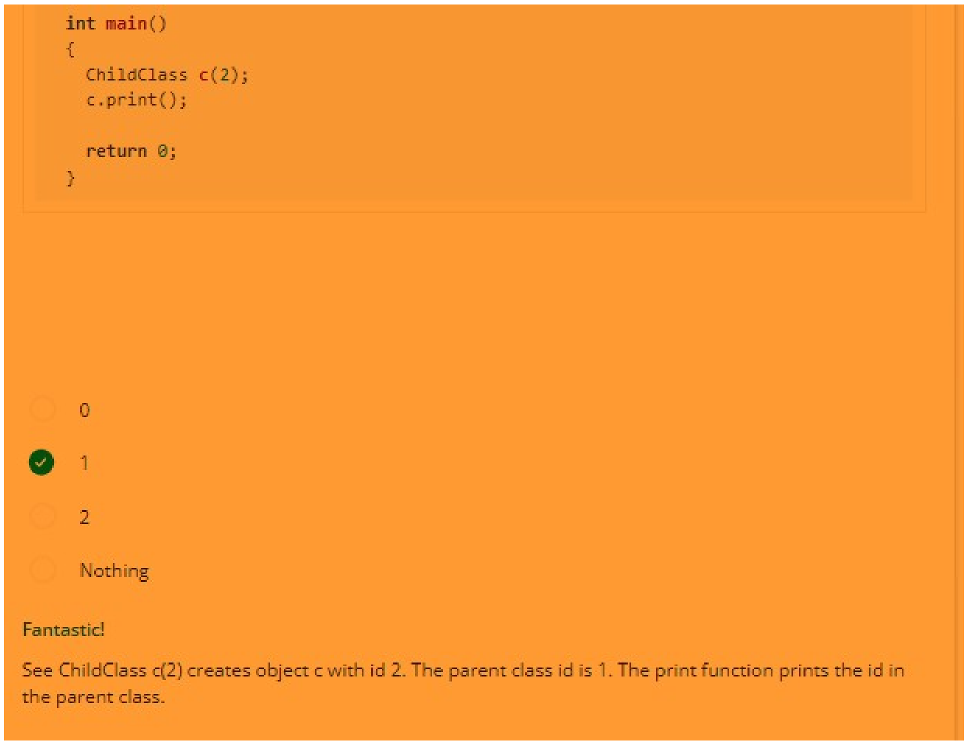
  Good job!

  "class A: public B" means A is a derived class of B. Note public must be present.

-

- 
  What is the output of the following code?

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

class ParentClass
{
public:
  int id;

  ParentClass(int id)
  {
    this->id = id;
  }

  void print()
  {
    cout << id << endl;
  }
};

class ChildClass: public ParentClass
{
public:
  int id;

  ChildClass(int id): ParentClass(1)
  {
    this->id = id;
  }
};
```



```
int main()
{
  ChildClass c(2);
  c.print();

  return 0;
}
```

-

0

[✓ 1]

2

Nothing

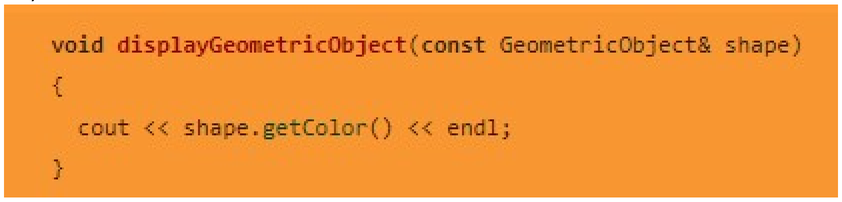
Fantastic!

See ChildClass c(2) creates object c with id 2. The parent class id is 1. The print function prints the id in the parent class.

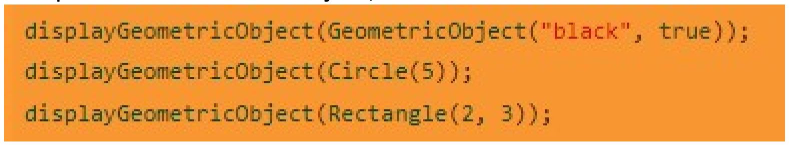
-

15.3 Generic Programming

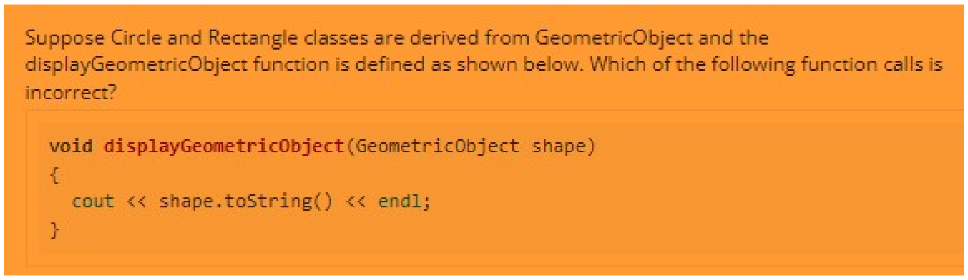
Wednesday, April 19, 2023 10:12 PM

- an object of a derived class can be passed wherever an object of a base type param is required, so a fn can be used generically for a wide range of object args, aka generic pgrming
- If a fn's param type is a base class (like GeometricObject) u can pass an object to this fn of any of the param's derived classes
- Aka generic pgrming
- Ex, if have fn
- 

```
void displayGeometricObject(const GeometricObject& shape)
{
  cout << shape.getColor() << endl;
}
```

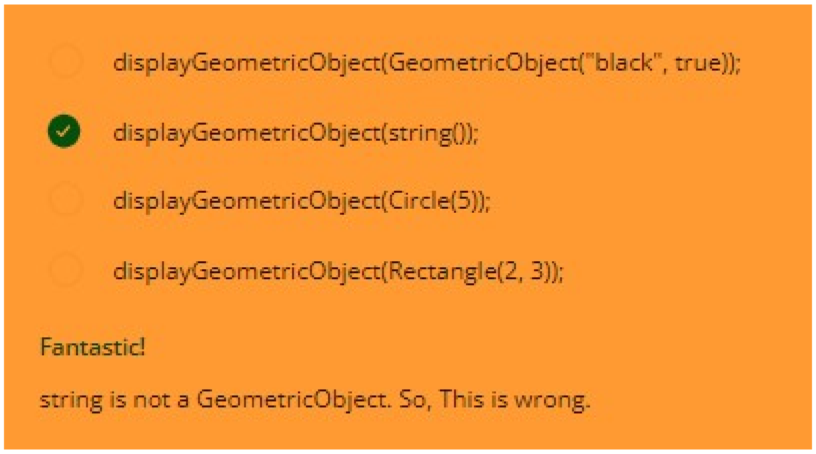
- The param is GeometricObject, can invoke this fn in this code
- 

```
displayGeometricObject(GeometricObject("black", true));
displayGeometricObject(Circle(5));
displayGeometricObject(Rectangle(2, 3));
```

- Each statement makes an anonymous object and passes it to invoke displayGeometricObject
- Since Circle and Rectangle are derived from GeometricObject, can pass a Circle/Rectangle object to the GeometricObject param type in the displayGeometricObject fn
-
- 

Suppose Circle and Rectangle classes are derived from GeometricObject and the displayGeometricObject function is defined as shown below. Which of the following function calls is incorrect?

```
void displayGeometricObject(GeometricObject shape)
{
  cout << shape.toString() << endl;
}
```

-
- 

displayGeometricObject(GeometricObject("black", true));

[✓ displayGeometricObject(string());]

displayGeometricObject(Circle(5));

displayGeometricObject(Rectangle(2, 3));

Fantastic!

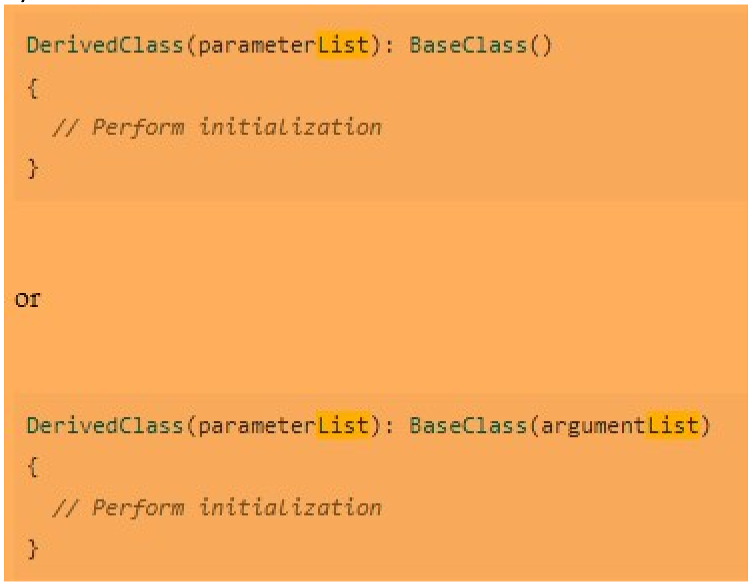
string is not a GeometricObject. So, This is wrong.

-

15.4 Constructors and Destructors

Thursday, April 20, 2023 4:41 PM

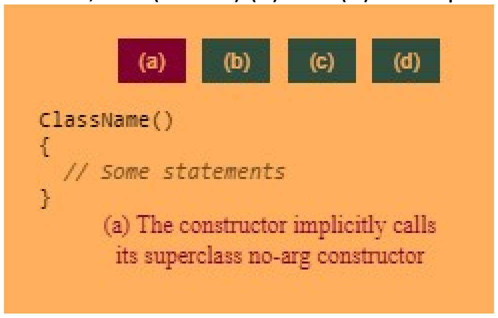
- The constructor of a derived class first calls its base class's constructor b4 it executes its own code, the destructor of a derived class executes its own code then auto calls its base class's destructor
- A derived class inherits accessible data fields and fns from its base class
- 15.4.1 Calling Base Class Constructors
  - A constructor is used to construct an instance of a class
  - Constructors of a base class not inherited in derived class, unlike data fields and fns
  - Can only be invoked from the constructors of the derived classes to initialize the data fields in the base class
  - Can invoke the base class's constructor from the constructor initializer list of the derived class
  - Syntax:



```
DerivedClass(parameterList): BaseClass()
{
  // Perform initialization
}
```

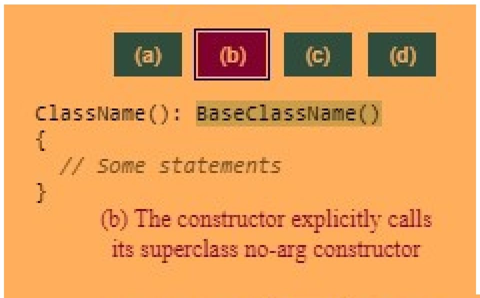
  - or

```
DerivedClass(parameterList): BaseClass(argumentList)
{
  // Perform initialization
}
```

  - First one invokes the no-arg constructor of its base class, last one invokes base class constructor w/ specified arg
  - Constructor in a derived class always invokes a constructor in its base class explicitly/implicitly
  - If a base constructor isn't invoked explicitly, the base class's no-arg constructor is invoked by default, like (below) (a) and (b) are equivalent, and (c) and (d) are equivalent
  - 

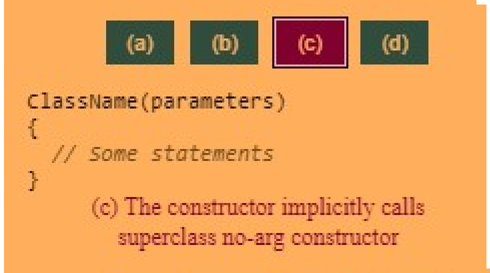
```
ClassName()
{
  // Some statements
}
```

(a) The constructor implicitly calls its superclass no-arg constructor

- 

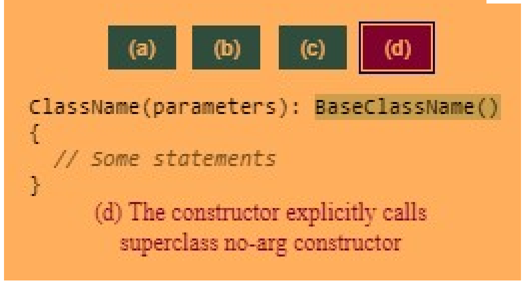
```
ClassName(): BaseClassName()
{
  // Some statements
}
```

(b) The constructor explicitly calls its superclass no-arg constructor

- 

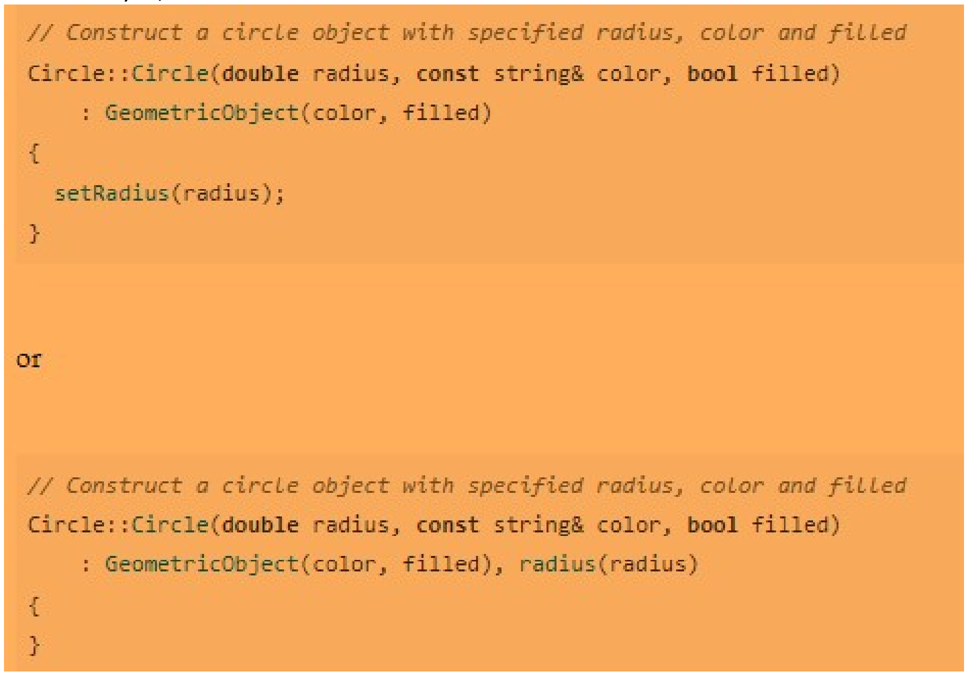
```
ClassName(parameters)
{
  // Some statements
}
```

(c) The constructor implicitly calls superclass no-arg constructor

- 

```
ClassName(parameters): BaseClassName()
{
  // Some statements
}
```

(d) The constructor explicitly calls superclass no-arg constructor

- The Circle(double radius, const string& color, bool filled) constructor (below) can also be implemented by invoking the base class's constructor: GeometricObject(const strint& color, bool filled) , like:
- 

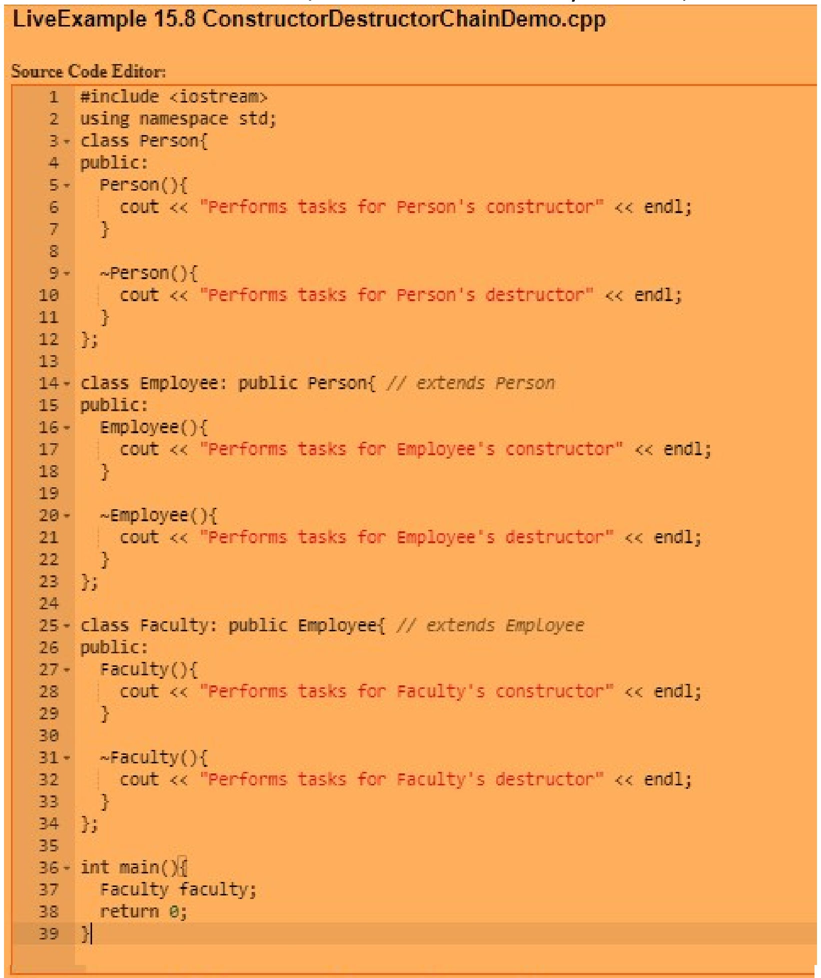
```
// Construct a circle object with specified radius, color and filled
Circle::Circle(double radius, const string& color, bool filled)
    : GeometricObject(color, filled)
{
  setRadius(radius);
}
```

or

```
// Construct a circle object with specified radius, color and filled
Circle::Circle(double radius, const string& color, bool filled)
    : GeometricObject(color, filled), radius(radius)
{
}
```

- 2nd also initializes the data field radius in the constructor initializer, radius is a data field defined in the Circle class

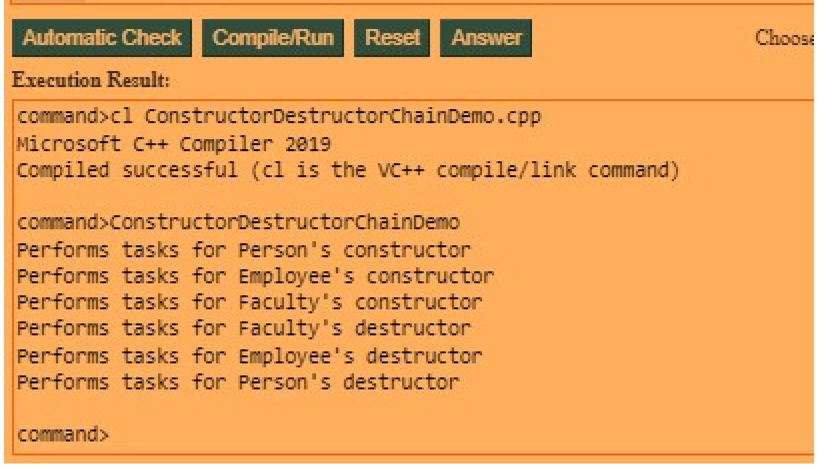
• 15.4.2 Constructor and Destructor Chaining
  - Constructing an instance of a class invokes the constructors of all the base classes along the inheritance chain
  - When constructing an object of a derived class, the derived class constructor first invokes its base class constructor b4 doing own tasks, if base class derived from another class, base class constructor invokes its parent class constructor b4 doing own tasks

- It goes till last constructor along inheritance hierarchy is called, aka constructor chaining
- Destructors auto invoked in reverse order
- When object of derived class is destroyed, the derived class destructor is called
- After fin its tasks, it invokes base class destructor
- Goes till last destructor w/ inheritance hierarchy is called, aka destructor chaining
- 

LiveExample 15.8 ConstructorDestructorChainDemo.cpp

Source Code Editor:

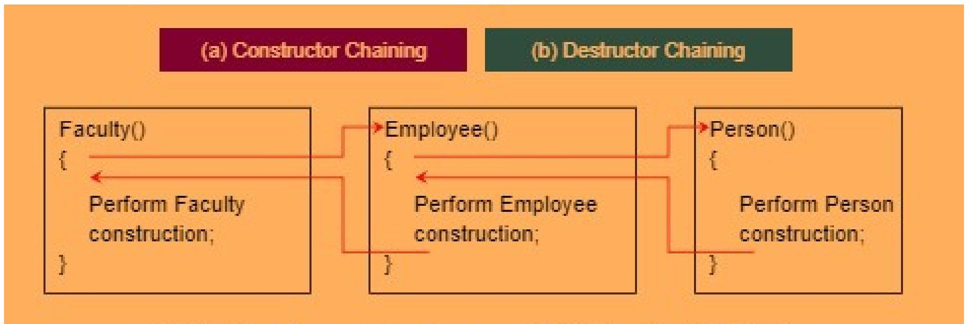
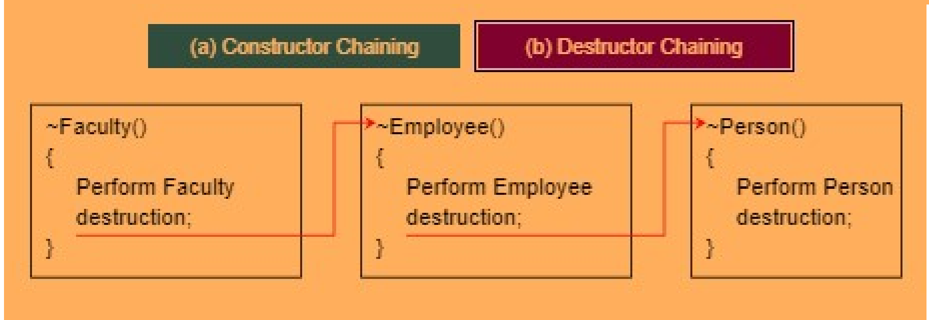
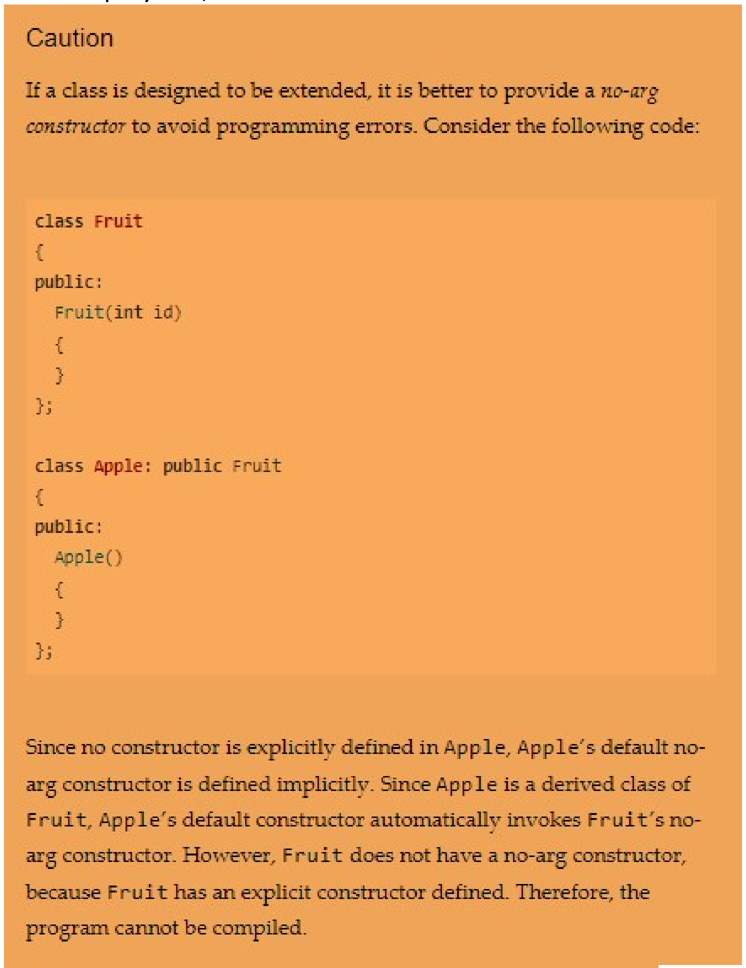
```cpp
#include <iostream>
using namespace std;
class Person{
public:
  Person(){
    cout << "Performs tasks for Person's constructor" << endl;
  }

  ~Person(){
    cout << "Performs tasks for Person's destructor" << endl;
  }
};

class Employee: public Person{ // extends Person
public:
  Employee(){
    cout << "Performs tasks for Employee's constructor" << endl;
  }

  ~Employee(){
    cout << "Performs tasks for Employee's destructor" << endl;
  }
};

class Faculty: public Employee{ // extends Employee
public:
  Faculty(){
    cout << "Performs tasks for Faculty's constructor" << endl;
  }

  ~Faculty(){
    cout << "Performs tasks for Faculty's destructor" << endl;
  }
};

int main(){
  Faculty faculty;
  return 0;
}
```

- 

Execution Result:

```
command>cl ConstructorDestructorChainDemo.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>ConstructorDestructorChainDemo
Performs tasks for Person's constructor
Performs tasks for Employee's constructor
Performs tasks for Faculty's constructor
Performs tasks for Faculty's destructor
Performs tasks for Employee's destructor
Performs tasks for Person's destructor

command>
```

- Pgrm makes an instance of Faculty (ln 48), which is derived from Employee, which is derived from Person
- So Faculty's constructor invokes Employee's constructor b4 doing its own tasks, and same for Employee's constructor invoking

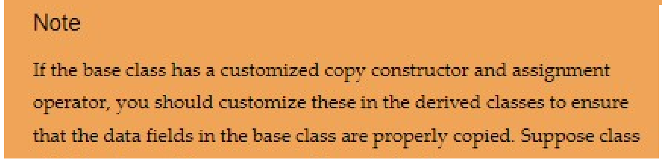
- 
(a) Constructor Chaining
(b) Destructor Chaining
Faculty() { Perform Faculty construction; }
Employee() { Perform Employee construction; }
Person() { Perform Person construction; }
(a) The base class constructors are called before the child class's
- 
(a) Constructor Chaining
(b) Destructor Chaining
~Faculty() { Perform Faculty destruction; }
~Employee() { Perform Employee destruction; }
~Person() { Perform Person destruction; }
(b) The base class destructors are called after the child class's
- When the pgrm exits, the Faculty object is destroyed, so the Facult's destructor is called, then Employee's, and then Person's
- 

Caution

If a class is designed to be extended, it is better to provide a *no-arg constructor* to avoid programming errors. Consider the following code:

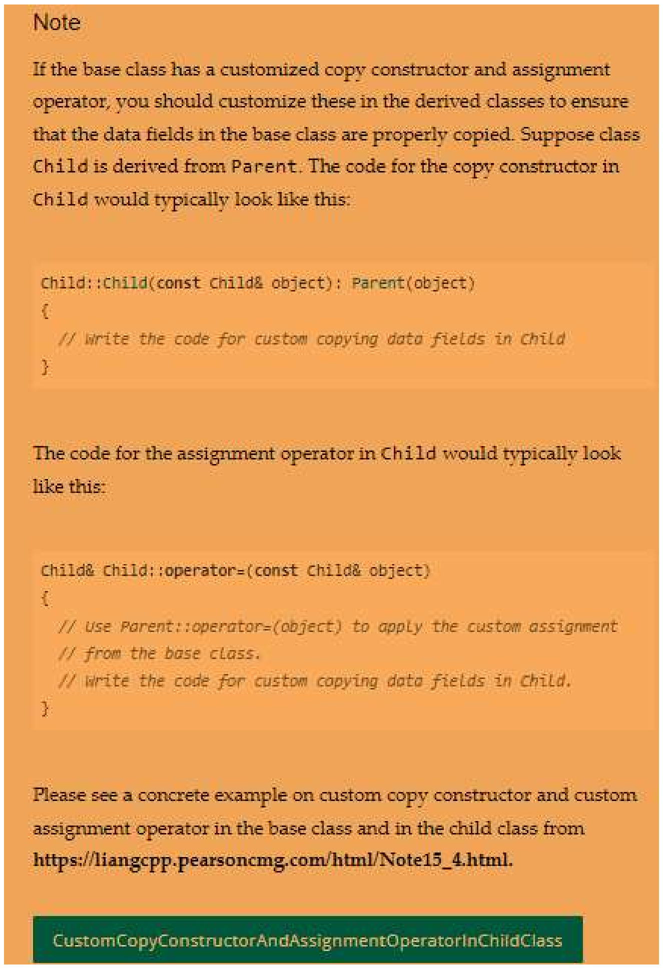
```
class Fruit
{
public:
  Fruit(int id)
  {
  }
};

class Apple: public Fruit
{
public:
  Apple()
  {
  }
};
```

Since no constructor is explicitly defined in `Apple`, `Apple`'s default no-arg constructor is defined implicitly. Since `Apple` is a derived class of `Fruit`, `Apple`'s default constructor automatically invokes `Fruit`'s no-arg constructor. However, `Fruit` does not have a no-arg constructor, because `Fruit` has an explicit constructor defined. Therefore, the program cannot be compiled.



Note

If the base class has a customized copy constructor and assignment operator, you should customize these in the derived classes to ensure that the data fields in the base class are properly copied. Suppose class

- 

Note

If the base class has a customized copy constructor and assignment operator, you should customize these in the derived classes to ensure that the data fields in the base class are properly copied. Suppose class `Child` is derived from `Parent`. The code for the copy constructor in `Child` would typically look like this:

```
Child::Child(const Child& object): Parent(object)
{
  // Write the code for custom copying data fields in Child
}
```

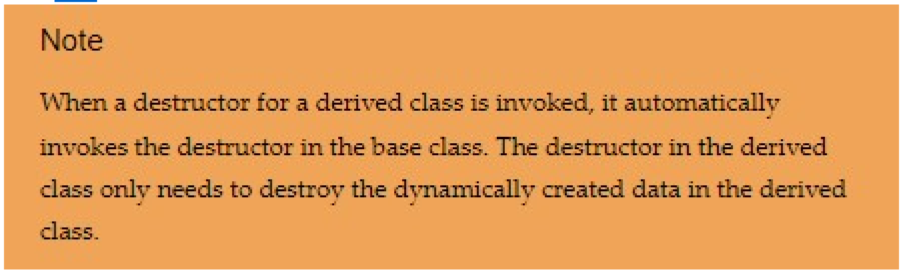
The code for the assignment operator in `Child` would typically look like this:

```
Child& Child::operator=(const Child& object)
{
  // Use Parent::operator=(object) to apply the custom assignment
  // from the base class.
  // Write the code for custom copying data fields in Child.
}
```

Please see a concrete example on custom copy constructor and custom assignment operator in the base class and in the child class from **https://liangcpp.pearsoncmg.com/html/Note15_4.html.**

CustomCopyConstructorAndAssignmentOperatorInChildClass

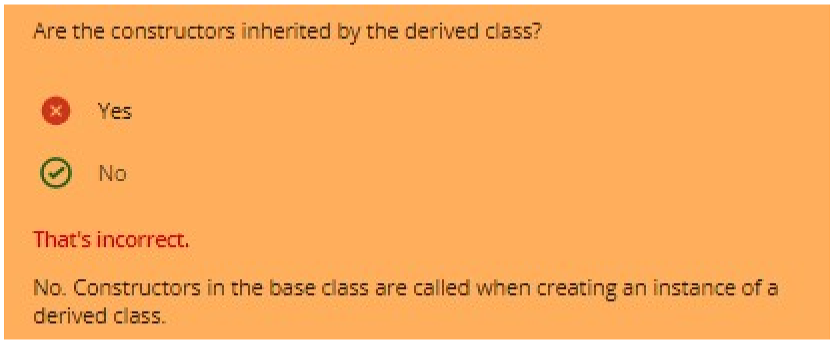
- The link from the bottom



Note

When a destructor for a derived class is invoked, it automatically invokes the destructor in the base class. The destructor in the derived class only needs to destroy the dynamically created data in the derived class.

-



Are the constructors inherited by the derived class?

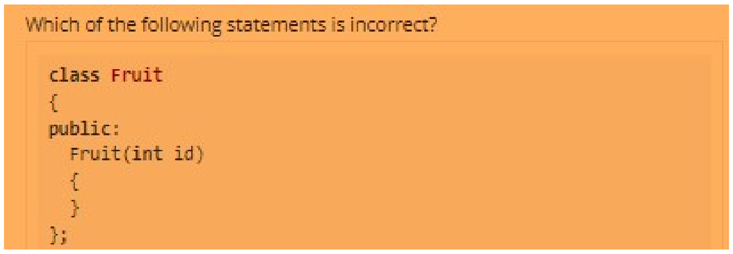
✗ Yes

✓ No

That's incorrect.

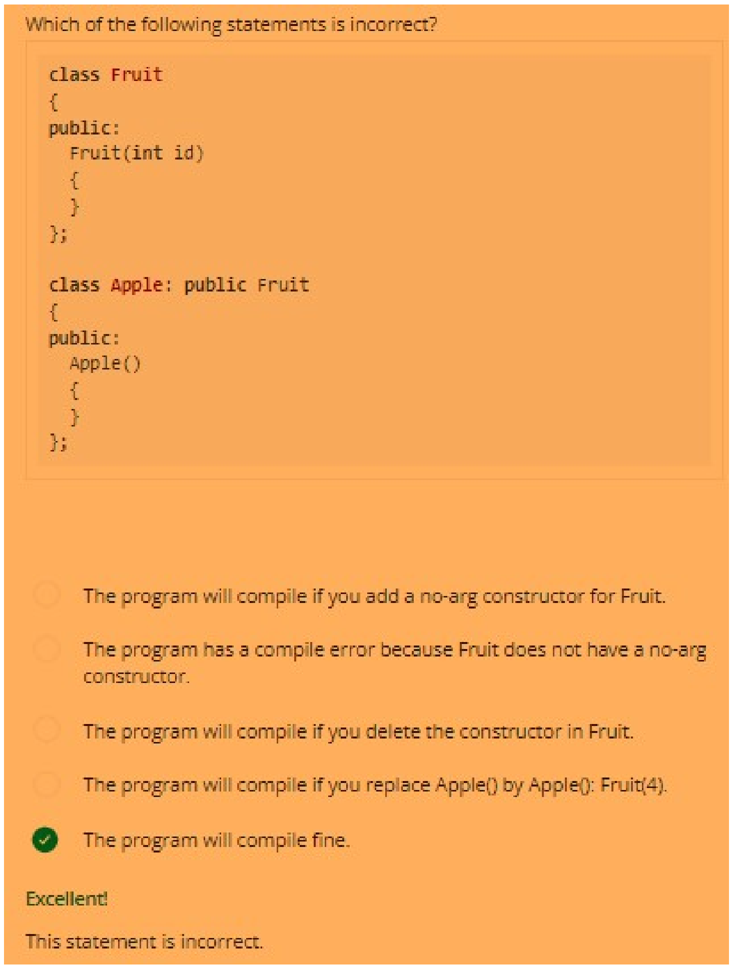
No. Constructors in the base class are called when creating an instance of a derived class.

-



Which of the following statements is incorrect?

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



Which of the following statements is incorrect?

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

class Apple: public Fruit
{
public:
  Apple()
  {
  }
};
```

○

The program will compile if you add a no-arg constructor for Fruit.

The program has a compile error because Fruit does not have a no-arg constructor.

The program will compile if you delete the constructor in Fruit.

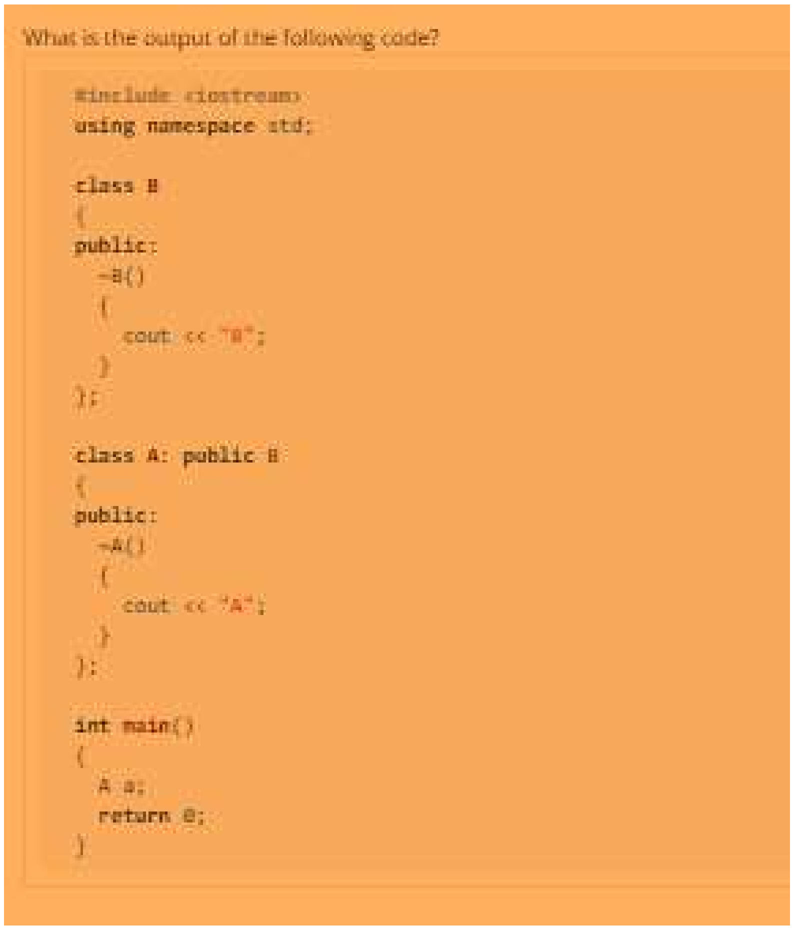
The program will compile if you replace Apple() by Apple(): Fruit(4).

[✓ The program will compile fine.]

Excellent!

This statement is incorrect.

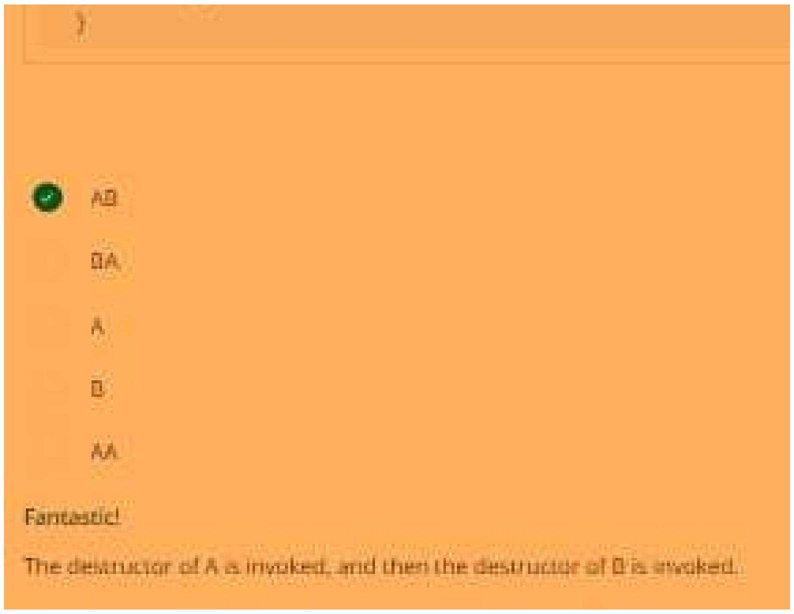
○



What is the output of the following code?

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

class B
{
public:
  ~B()
  {
    cout << "B";
  }
};

class A: public B
{
public:
  ~A()
  {
    cout << "A";
  }
};

int main()
{
  A a;
  return 0;
}
```

○



```
}
```

[✓ AB]

BA

A

B

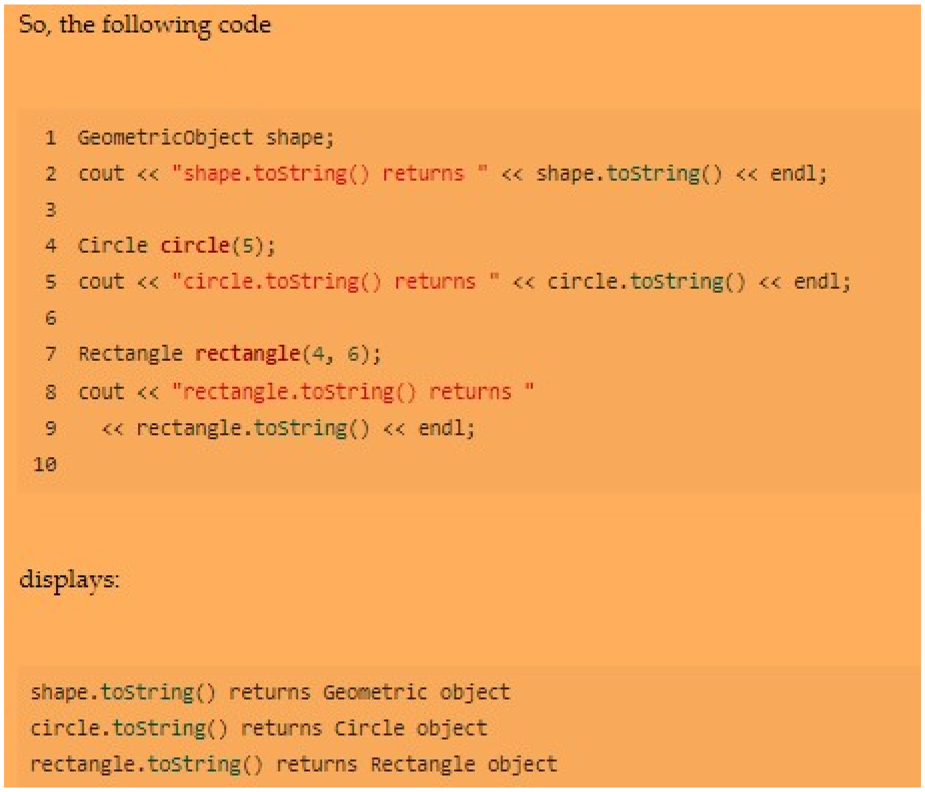
AA

Fantastic!

The destructor of A is invoked, and then the destructor of B is invoked.

15.5 Redefining Functions

Thursday, April 20, 2023 5:35 PM

- A fn defined in the base class can be redefined in the derived classes
- The toString() fn is defined in the GeometricObject class to return a string "Geometric object", like
-
```
string GeometricObject::toString() const
{
  return "Geometric object";
}
```
- To redefine a base class's fn in the derived class, need to add the fn's prototype in the derived class's header file, and give a new implementation for the fn in the derived class's implementation file
- The toString() fn is redefined in the Circle class like
-
```
string Circle::toString() const
{
  return "Circle object";
}
```
- And in the Rectangle class like
-
```
string Rectangle::toString() const
{
  return "Rectangle object";
}
```
-
- 

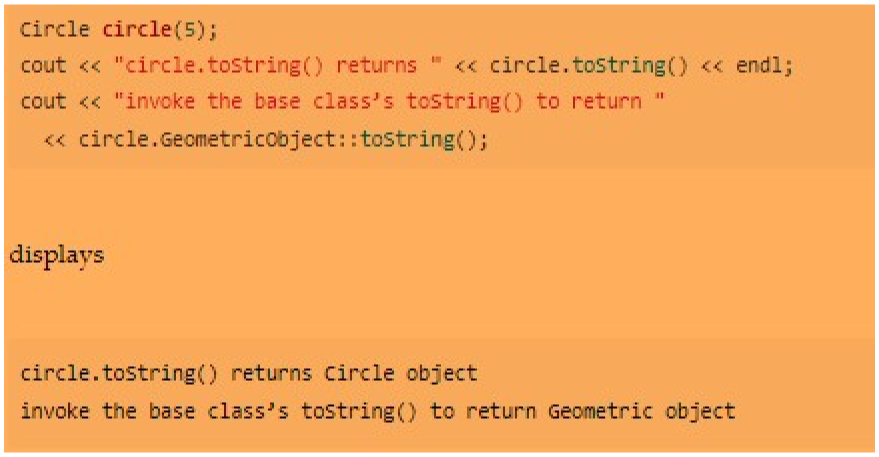
So, the following code

```
GeometricObject shape;
cout << "shape.toString() returns " << shape.toString() << endl;

Circle circle(5);
cout << "circle.toString() returns " << circle.toString() << endl;

Rectangle rectangle(4, 6);
cout << "rectangle.toString() returns "
  << rectangle.toString() << endl;

```

displays:

```
shape.toString() returns Geometric object
circle.toString() returns Circle object
rectangle.toString() returns Rectangle object
```

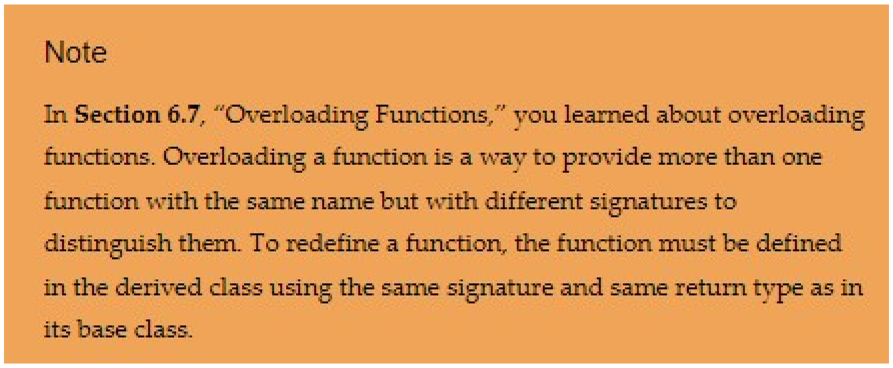
- If want to invoke the toString fn defined in the GeometricObject class on the calling object circle, use the scope resolution operator(::) w/ the base class name, like



```
Circle circle(5);
cout << "circle.toString() returns " << circle.toString() << endl;
cout << "invoke the base class's toString() to return "
  << circle.GeometricObject::toString();
```

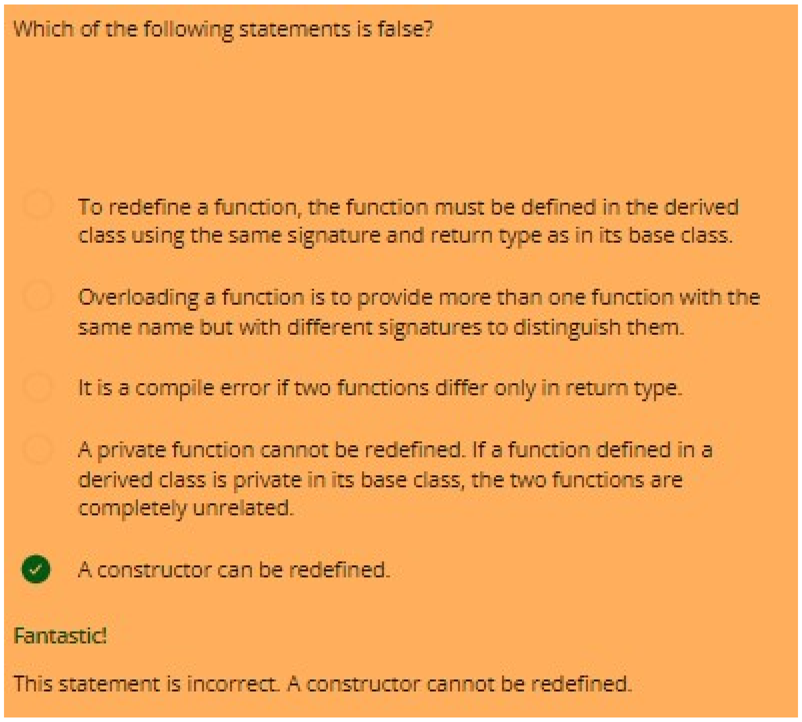
displays

```
circle.toString() returns Circle object
invoke the base class's toString() to return Geometric object
```



Note

In **Section 6.7**, "Overloading Functions," you learned about overloading functions. Overloading a function is a way to provide more than one function with the same name but with different signatures to distinguish them. To redefine a function, the function must be defined in the derived class using the same signature and same return type as in its base class.

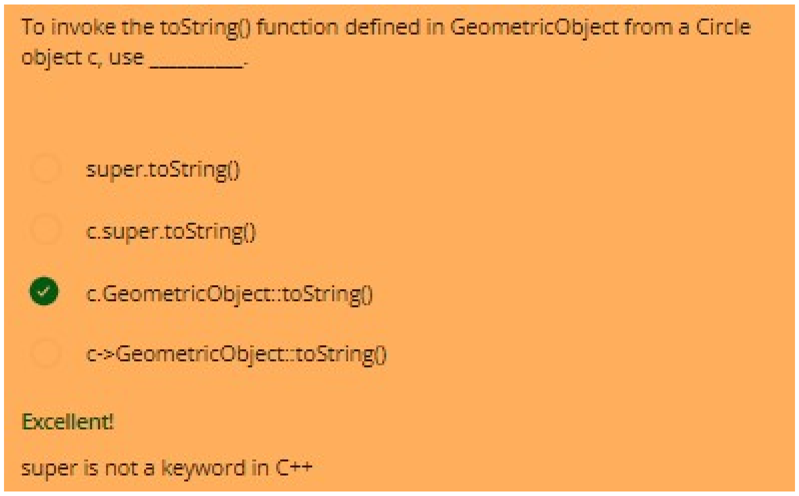


Which of the following statements is false?

- To redefine a function, the function must be defined in the derived class using the same signature and return type as in its base class.
- Overloading a function is to provide more than one function with the same name but with different signatures to distinguish them.
- It is a compile error if two functions differ only in return type.
- A private function cannot be redefined. If a function defined in a derived class is private in its base class, the two functions are completely unrelated.
- [✓ A constructor can be redefined.]

Fantastic!

This statement is incorrect. A constructor cannot be redefined.

- 
To invoke the toString() function defined in GeometricObject from a Circle object c, use _________.

super.toString()

c.super.toString()

[✓ c.GeometricObject::toString()]

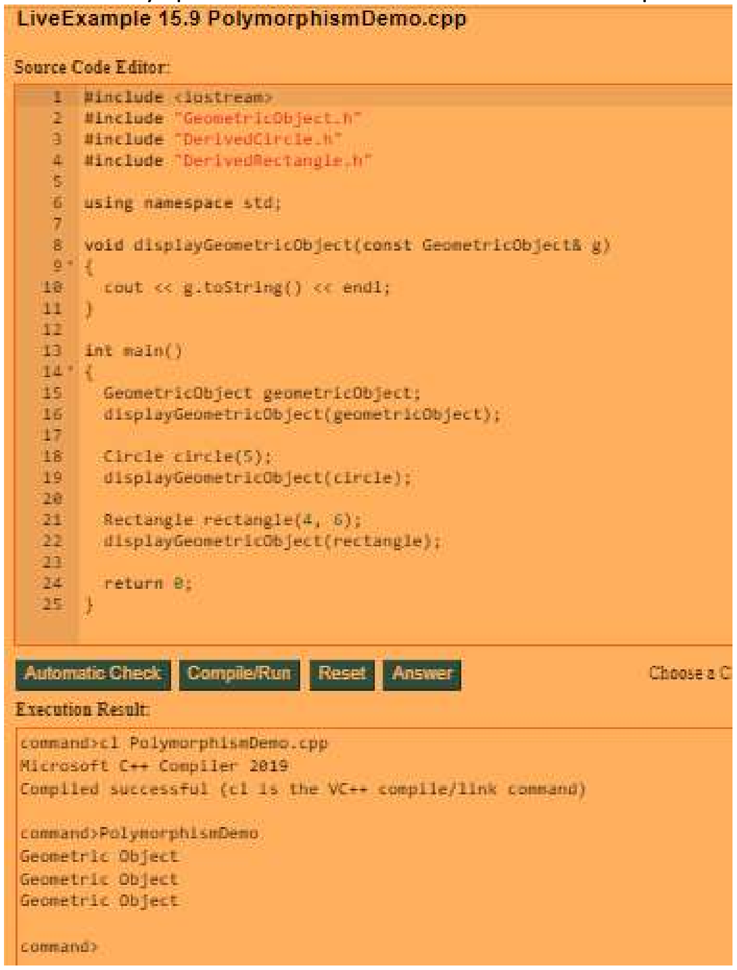
c->GeometricObject::toString()

Excellent!

super is not a keyword in C++
-
-

15.6 Polymorphism

Thursday, April 20, 2023 5:51 PM

- Polymorphism = var of a supertype can refer to a subtype object
- 3 pillars of oop: encapsulation, inheritance, and polymorphism
- Subtype = type defined by a derived class
- Supertype = type defined by its base class
  - Ex: So Circle is a subtype of GeometricObject and GeometricObject is a supertype for Circle
- Inheritance relationship lets a derived class inherit features from its base class w/ new features
- Derived class is specialization of its base class, every instance of a derived class is also an instance of its base class, not opp tho
  - Ex: every circle is a geometric object, not every geometric object is a circle
- So can always pass an instance of a derived class to a param of its base class type
- 

LiveExample 15.9 PolymorphismDemo.cpp

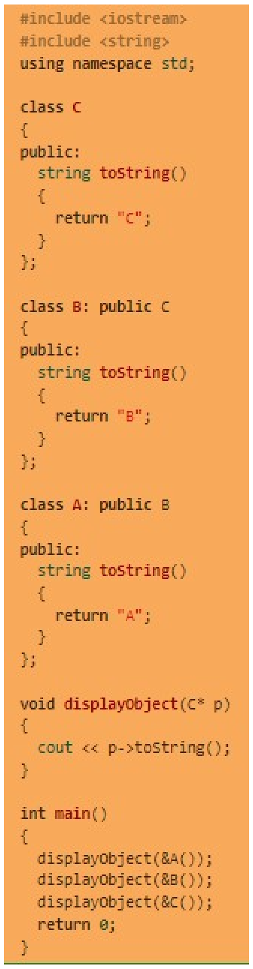
Source Code Editor:

```cpp
#include <iostream>
#include "GeometricObject.h"
#include "DerivedCircle.h"
#include "DerivedRectangle.h"

using namespace std;

void displayGeometricObject(const GeometricObject& g)
{
  cout << g.toString() << endl;
}

int main()
{
  GeometricObject geometricObject;
  displayGeometricObject(geometricObject);

  Circle circle(5);
  displayGeometricObject(circle);

  Rectangle rectangle(4, 6);
  displayGeometricObject(rectangle);

  return 0;
}
```

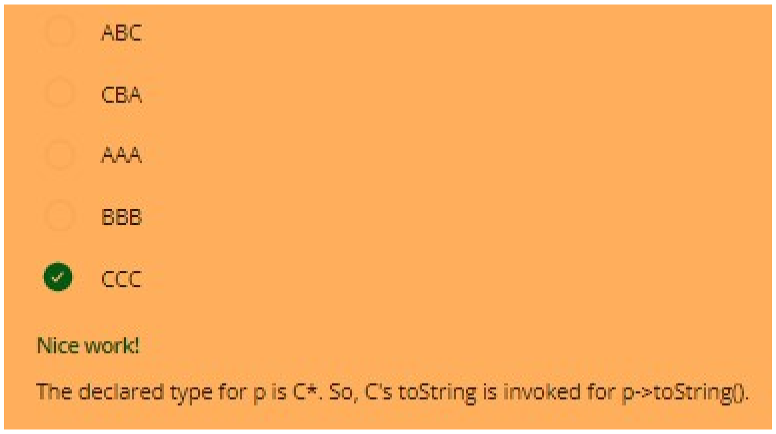
Execution Result:

```
command>cl PolymorphismDemo.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>PolymorphismDemo
Geometric Object
Geometric Object
Geometric Object

command>
```

- Fn displayGeometricObject takes a param of the GeometricObject type
- Can invoke displayGeometricObject by passing any instance of GeometricObject, Circle, and Rectangle
- An object of a derived class can be used wherever its base class object is used
- Aka polymorphism (greek, "many forms")
- So basically, means that a var of a supertype can refer to a subtype object
-

- 

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

class C
{
public:
  string toString()
  {
    return "C";
  }
};

class B: public C
{
public:
  string toString()
  {
    return "B";
  }
};

class A: public B
{
public:
  string toString()
  {
    return "A";
  }
};

void displayObject(C* p)
{
  cout << p->toString();
}

int main()
{
  displayObject(&A());
  displayObject(&B());
  displayObject(&C());
  return 0;
}
```

-
- 

ABC
CBA
AAA
BBB
[✓ CCC]

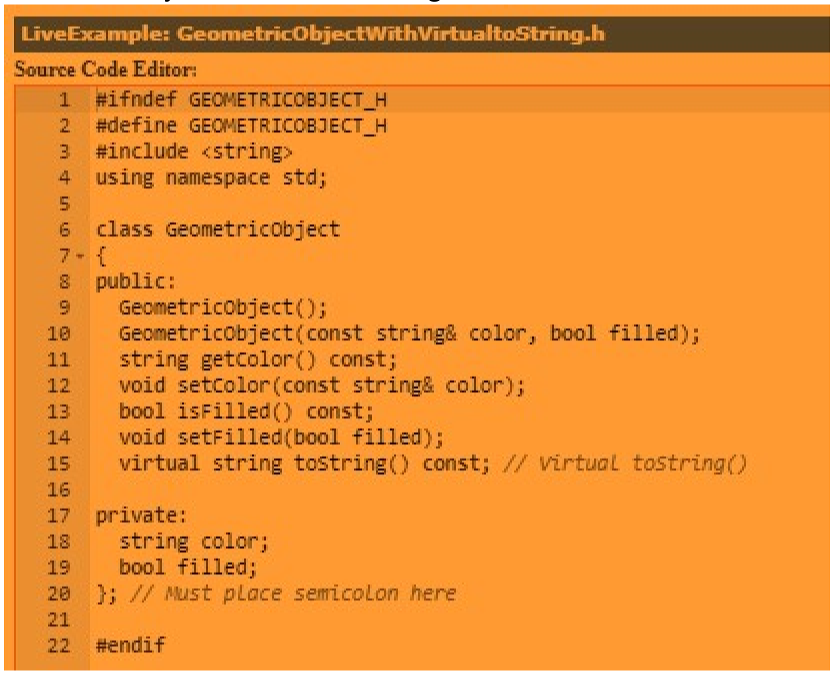
Nice work!

The declared type for p is C*. So, C's toString is invoked for p->toString().

-

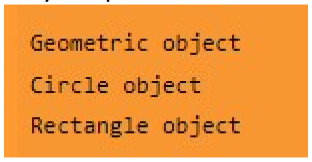
15.7 Virtual Functions and Dynamic Binding

Thursday, April 20, 2023 9:42 PM

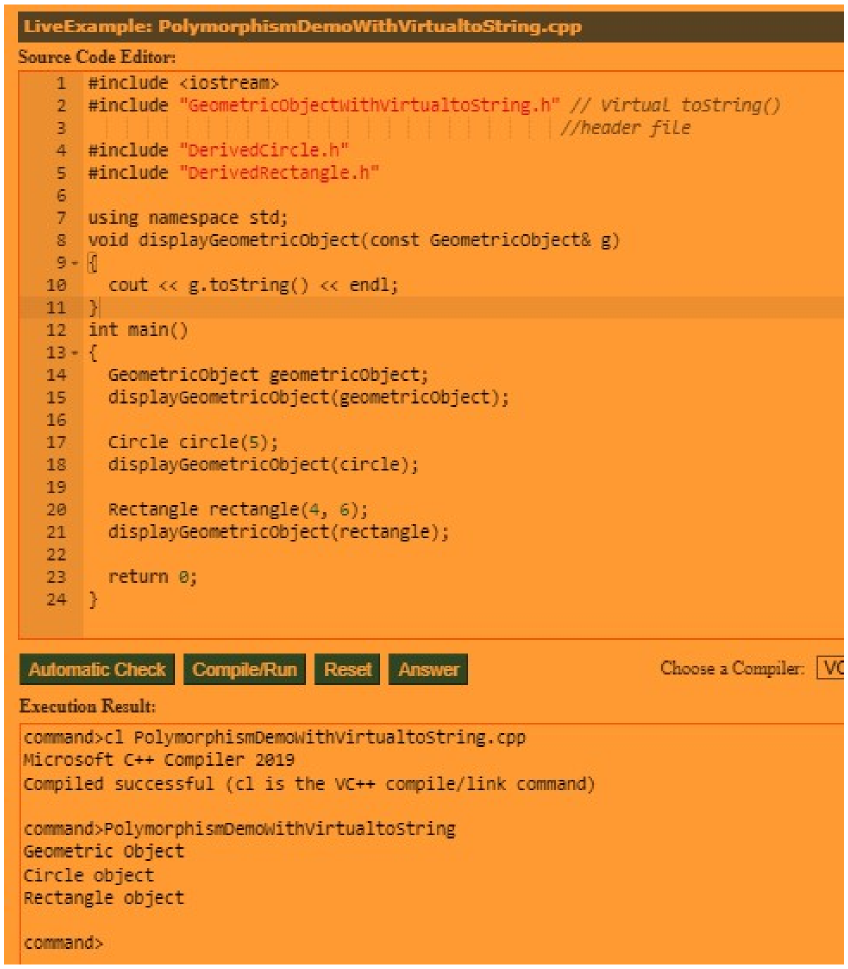
- A fn can be implemented in several classes along the inheritance chain, virtual fns let the sys decide which fn is invoked @ runtime based on the actual type of the object
- B4 code, the displayGeometricObject fn that invokes the toString fn on a GeometricObject
- The displayGeometricObject fn is invoked by passing an object of GeometricObject, Circle, and Rectangle (respectively)
- Output shows the toString() fn defined in class GeometricObject is invoked
- Can do all this by just declaring toString as virtual fn in the base class GeometricObject
    - Invoke the toString(0 fn defined in Circle when executing displayGeometricObject(circle)
    - The toString(0 fn defined in Rectangle when executing displayGeometricObject(rectangle)
    - And toString(0 fn defined in GeometricObject when executing displayGeometricObject(geometricObject)
- B4 code, what if replace a line w/ this (and make a new file called GeometricObjectWithVirtualtoString.h:
- 

  LiveExample: GeometricObjectWithVirtualtoString.h

  Source Code Editor:

```cpp
#ifndef GEOMETRICOBJECT_H
#define GEOMETRICOBJECT_H
#include <string>
using namespace std;

class GeometricObject
{
public:
  GeometricObject();
  GeometricObject(const string& color, bool filled);
  string getColor() const;
  void setColor(const string& color);
  bool isFilled() const;
  void setFilled(bool filled);
  virtual string toString() const; // Virtual toString()

private:
  string color;
  bool filled;
}; // Must place semicolon here

#endif
```
- Now if do diff b4 code w/ virtual toString() in PolymorphismDemoWithVirtualtoString.cpp, see this output:
- 

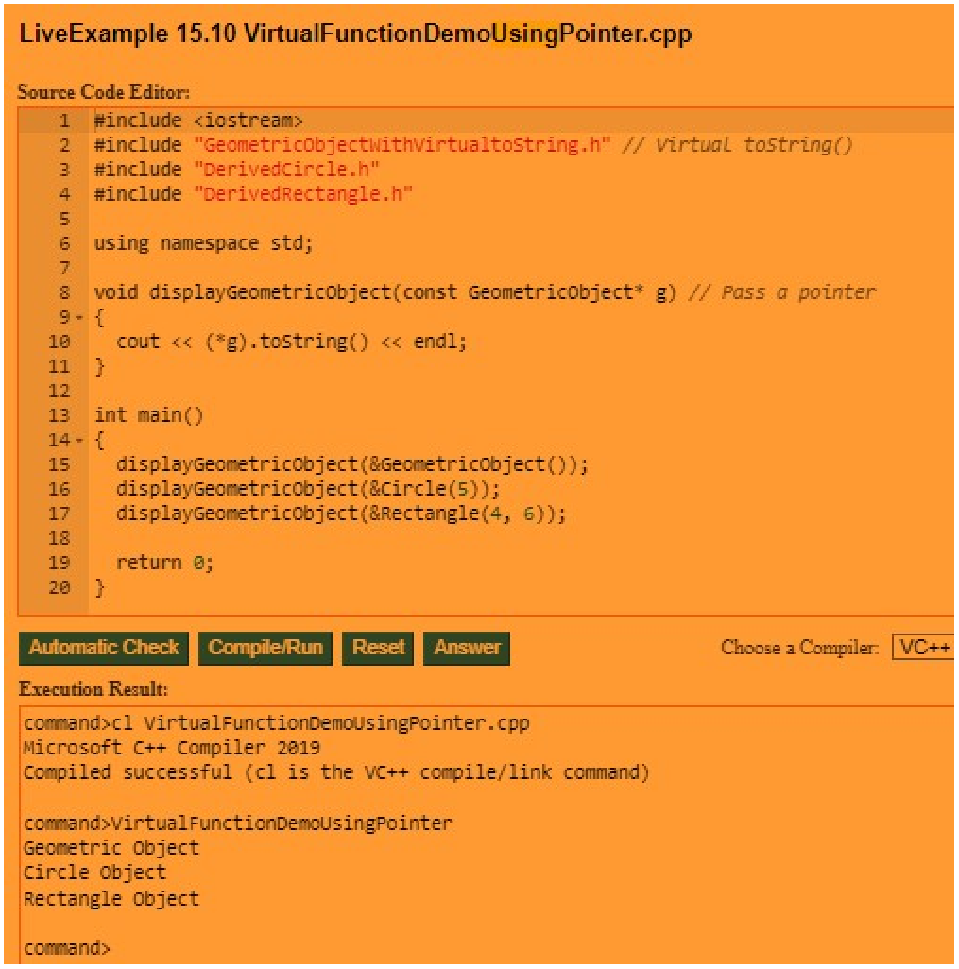
```
Geometric object
Circle object
Rectangle object
```
- For this code:

- 

```cpp
#include <iostream>
#include "GeometricObjectWithVirtualtoString.h" // Virtual toString()
                                                //header file
#include "DerivedCircle.h"
#include "DerivedRectangle.h"

using namespace std;
void displayGeometricObject(const GeometricObject& g)
{
  cout << g.toString() << endl;
}
int main()
{
  GeometricObject geometricObject;
  displayGeometricObject(geometricObject);

  Circle circle(5);
  displayGeometricObject(circle);

  Rectangle rectangle(4, 6);
  displayGeometricObject(rectangle);

  return 0;
}
```

```
command>cl PolymorphismDemoWithVirtualtoString.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>PolymorphismDemoWithVirtualtoString
Geometric Object
Circle object
Rectangle object

command>
```

- W/ the toString() fn defined as virtual in the base class, C++ dynamically determines which toString() fn to invoke at runtime
- When invoking displayGeometricObject(circle), a Circle object is passed to g by reference, since g referes to an object of the Circpe type, the toString fn defined in class Circle is invoked
- The capability of determining which fn to invoke at runtime is known as dynamic binding
- In C++, redefining a virtual fn in a derived class is called overriding a fn
- To let dynamic binding for a fn, need 2 things
    - The fn must be defined virtual in the base class
    - The var that references the object must be passed by reference or passed as a pointer in the virtual fn
- B4 code passes the object to a param by reference, but can rewrite some lines by passing a pointer, like this:

- 

LiveExample 15.10 VirtualFunctionDemoUsingPointer.cpp

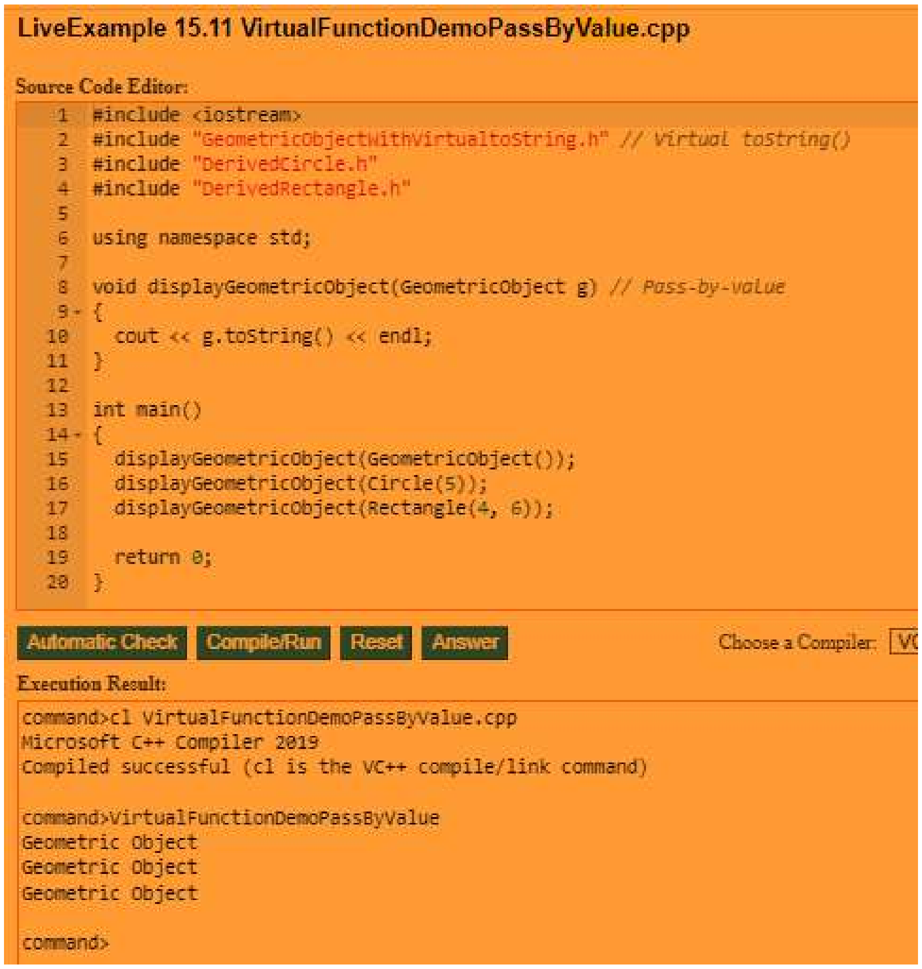
Source Code Editor:

```cpp
#include <iostream>
#include "GeometricObjectWithVirtualtoString.h" // Virtual toString()
#include "DerivedCircle.h"
#include "DerivedRectangle.h"

using namespace std;

void displayGeometricObject(const GeometricObject* g) // Pass a pointer
{
  cout << (*g).toString() << endl;
}

int main()
{
  displayGeometricObject(&GeometricObject());
  displayGeometricObject(&Circle(5));
  displayGeometricObject(&Rectangle(4, 6));

  return 0;
}
```

Execution Result:

```
command>cl VirtualFunctionDemoUsingPointer.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>VirtualFunctionDemoUsingPointer
Geometric Object
Circle Object
Rectangle Object

command>
```

- But if the object arg is passed by val, virtual fns are not bound dynamically
- Shown below, even though fn is defined to be virtual, the output is the same as it would be w/out using the virtual fn

- 

LiveExample 15.11 VirtualFunctionDemoPassByValue.cpp

Source Code Editor:

```cpp
#include <iostream>
#include "GeometricObjectWithVirtualtoString.h" // Virtual toString()
#include "DerivedCircle.h"
#include "DerivedRectangle.h"

using namespace std;

void displayGeometricObject(GeometricObject g) // Pass-by-value
{
  cout << g.toString() << endl;
}

int main()
{
  displayGeometricObject(GeometricObject());
  displayGeometricObject(Circle(5));
  displayGeometricObject(Rectangle(4, 6));

  return 0;
}
```

Execution Result:

```
command>cl VirtualFunctionDemoPassByValue.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>VirtualFunctionDemoPassByValue
Geometric Object
Geometric Object
Geometric Object

command>
```

- Points abt virtual fns
  - If a fn is defined virtual in a base class, it auto virtual in all its derived classes, the keyword virtual need not be added in fn declaration in the derived class
  - Matching a fn signature and binding a fn implementation are 2 sep issues, the declared type of the var decides which fn to match at compile time, aka static binding, the compiler finds a matching fn according to param type, numb of params, and order of the params @ compile time, a virtual fn can be implemented in several derived classes, C++ dynamically binds the implementation of fn at runtime, decided by actual class of the object referenced by the var, aka dynamic binding
  - Dynamic binding works like this:
    - If an object o is an instance of classes C1, C2, …, Cn-1, and Cn, where C1 is a subclass of C2, and C2 is a subclass of C3, …
    - Cn is most general class, and C1 is the most specific class, if o invokes a fn p, C++ searches implementation for the fn p in C1, C2, …, Cn, in this order until its found
    - Once implementaion found, search stops and first-found implementation is invoked

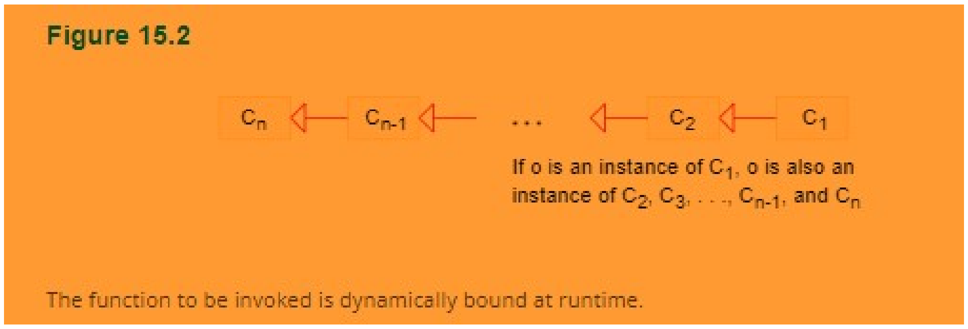
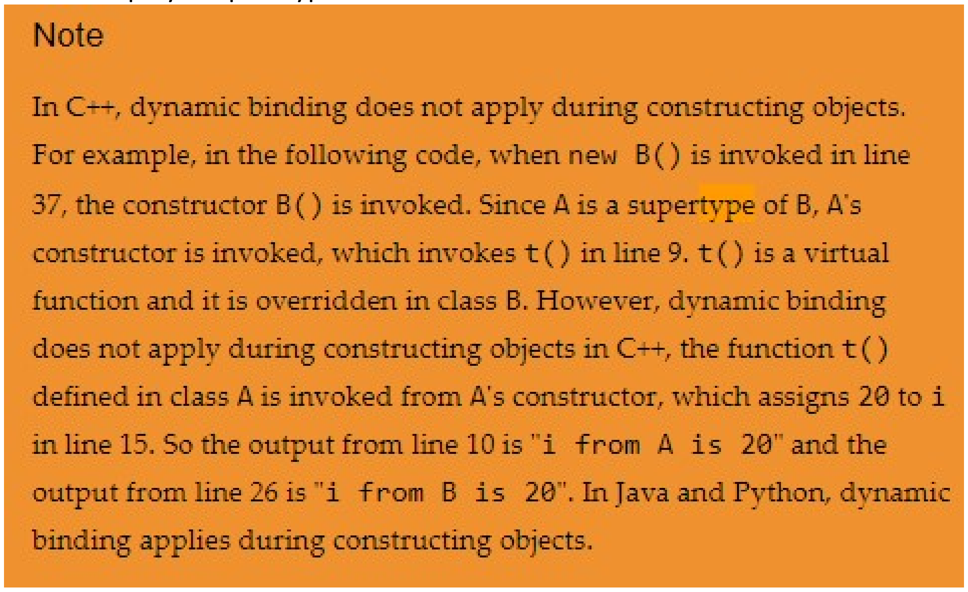
- 

  Figure 15.2

  C_n
C_{n-1}
...
C_2
C_1
If o is an instance of C_1, o is also an instance of C_2, C_3, . . ., C_{n-1}, and C_n

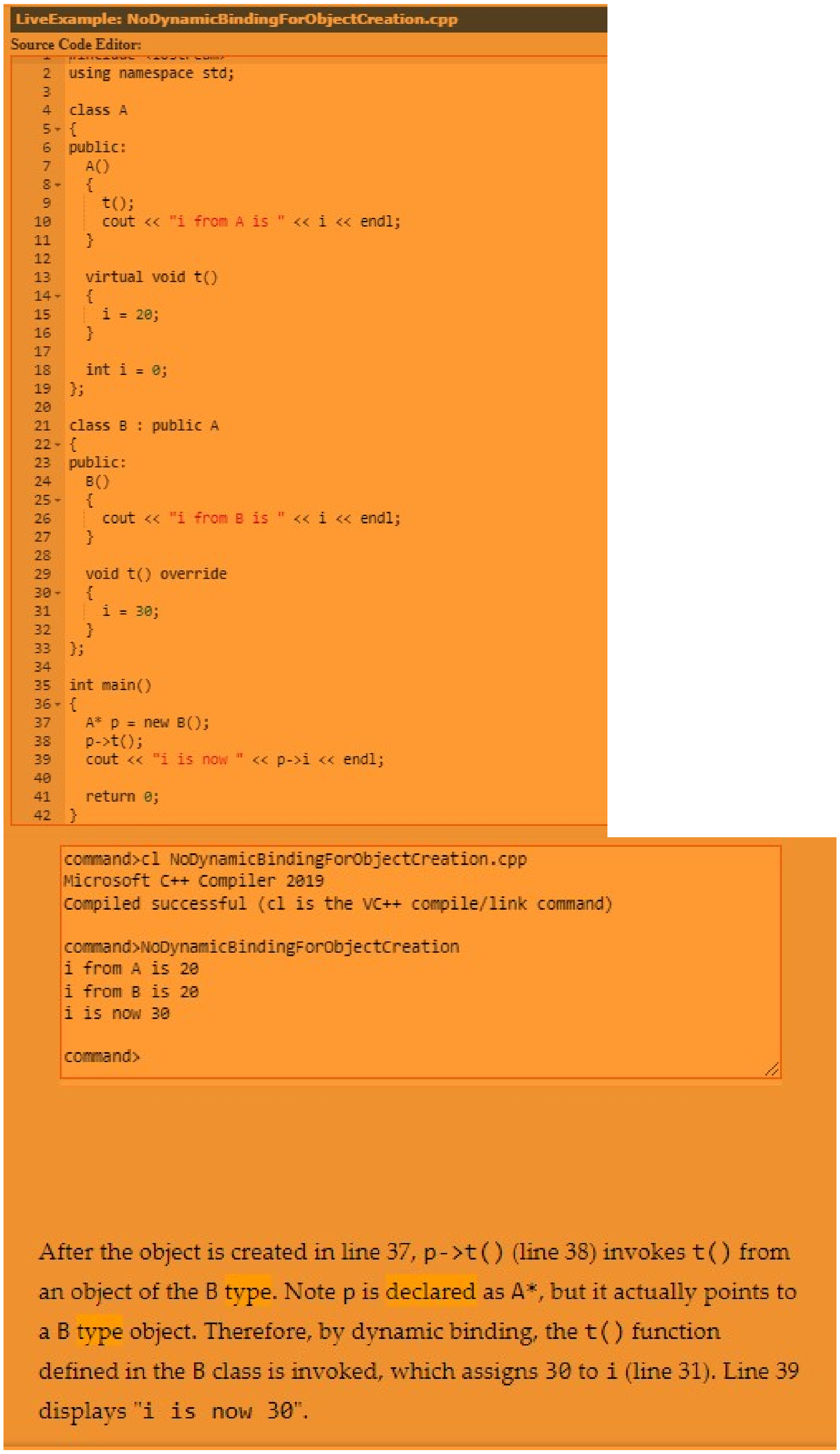
  The function to be invoked is dynamically bound at runtime.

- If a fn defined in a base class needs to be redefined in its derived classes, u should define it virtual to avoid confusion and mistakes, but if a fn will not be redefined, its more efficient not to declare it virtual bc more time and sys resource are required to bind virtual fns dynamically @ runttime, can call a classw/ a virtual fn a polymorphic type



Note

In C++, dynamic binding does not apply during constructing objects. For example, in the following code, when `new B()` is invoked in line 37, the constructor `B()` is invoked. Since A is a supertype of B, A's constructor is invoked, which invokes `t()` in line 9. `t()` is a virtual function and it is overridden in class B. However, dynamic binding does not apply during constructing objects in C++, the function `t()` defined in class A is invoked from A's constructor, which assigns `20` to `i` in line 15. So the output from line 10 is "`i from A is 20`" and the output from line 26 is "`i from B is 20`". In Java and Python, dynamic binding applies during constructing objects.



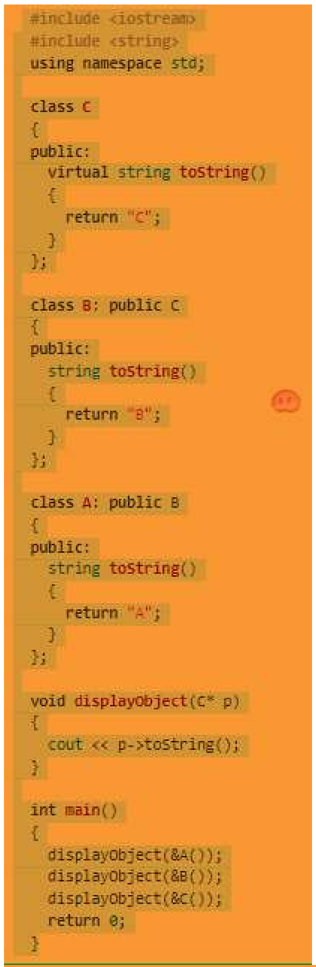
LiveExample: NoDynamicBindingForObjectCreation.cpp

Source Code Editor:

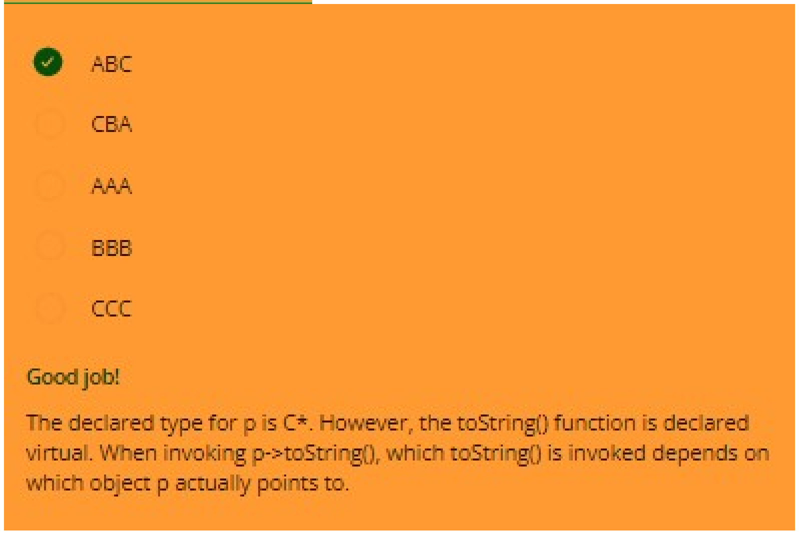
```
 1  #include <iostream>
 2  using namespace std;
 3
 4  class A
 5  {
 6  public:
 7    A()
 8    {
 9      t();
10      cout << "i from A is " << i << endl;
11    }
12
13    virtual void t()
14    {
15      i = 20;
16    }
17
18    int i = 0;
19  };
20
21  class B : public A
22  {
23  public:
24    B()
25    {
26      cout << "i from B is " << i << endl;
27    }
28
29    void t() override
30    {
31      i = 30;
32    }
33  };
34
35  int main()
36  {
37    A* p = new B();
38    p->t();
39    cout << "i is now " << p->i << endl;
40
41    return 0;
42  }
```

```
command>cl NoDynamicBindingForObjectCreation.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>NoDynamicBindingForObjectCreation
i from A is 20
i from B is 20
i is now 30

command>
```

After the object is created in line 37, `p->t()` (line 38) invokes `t()` from an object of the B type. Note p is declared as A*, but it actually points to a B type object. Therefore, by dynamic binding, the `t()` function defined in the B class is invoked, which assigns `30` to `i` (line 31). Line 39 displays "`i is now 30`".

- 

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

class C
{
public:
  virtual string toString()
  {
    return "C";
  }
};

class B: public C
{
public:
  string toString()
  {
    return "B";
  }
};

class A: public B
{
public:
  string toString()
  {
    return "A";
  }
};

void displayObject(C* p)
{
  cout << p->toString();
}

int main()
{
  displayObject(&A());
  displayObject(&B());
  displayObject(&C());
  return 0;
}
```

- 

  [✓ ABC]

  CBA

  AAA

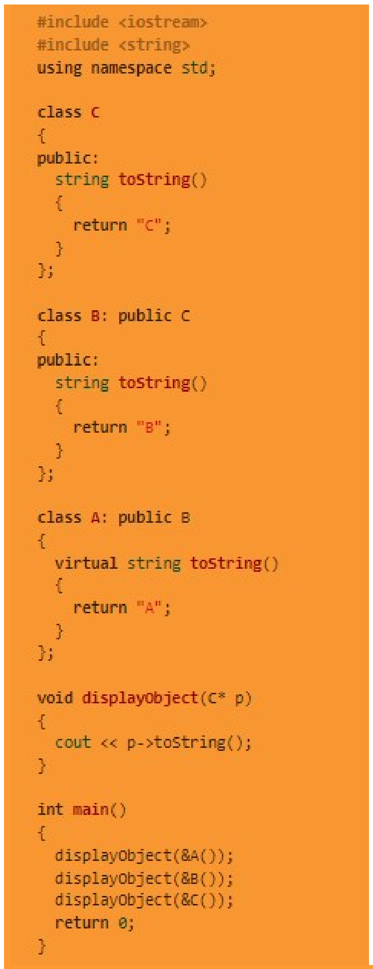
  BBB

  CCC

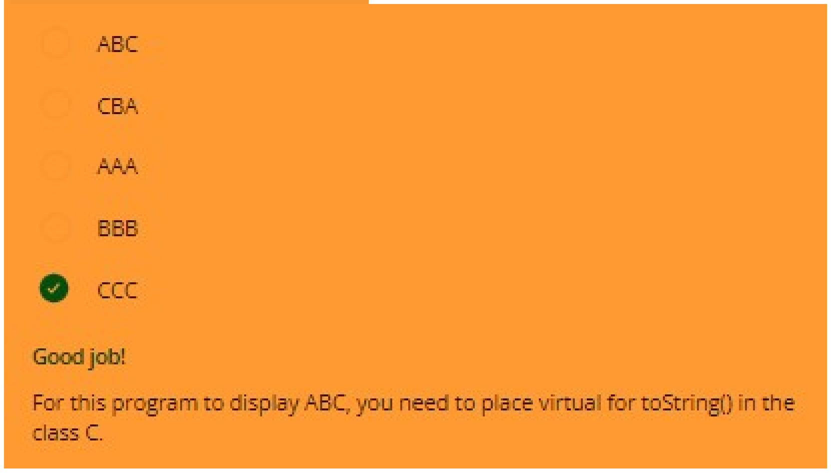
  Good job!

  The declared type for p is C*. However, the toString() function is declared virtual. When invoking p->toString(), which toString() is invoked depends on which object p actually points to.

-

- 

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

class C
{
public:
  string toString()
  {
    return "C";
  }
};

class B: public C
{
public:
  string toString()
  {
    return "B";
  }
};

class A: public B
{
  virtual string toString()
  {
    return "A";
  }
};

void displayObject(C* p)
{
  cout << p->toString();
}

int main()
{
  displayObject(&A());
  displayObject(&B());
  displayObject(&C());
  return 0;
}
```

- 

  ○ ABC

  ○ CBA

  ○ AAA

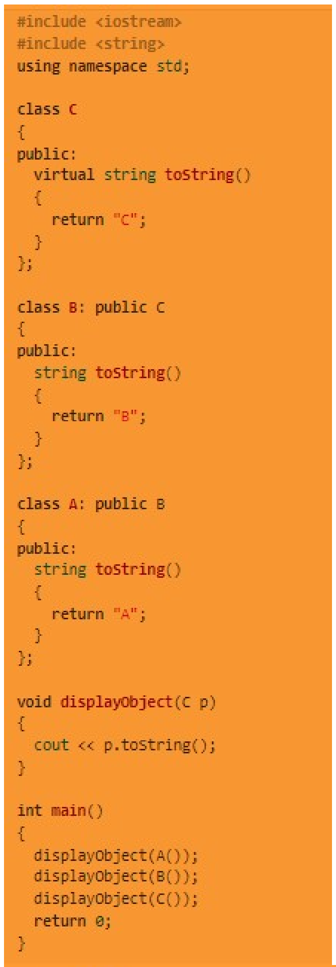
  ○ BBB

  ✓ CCC

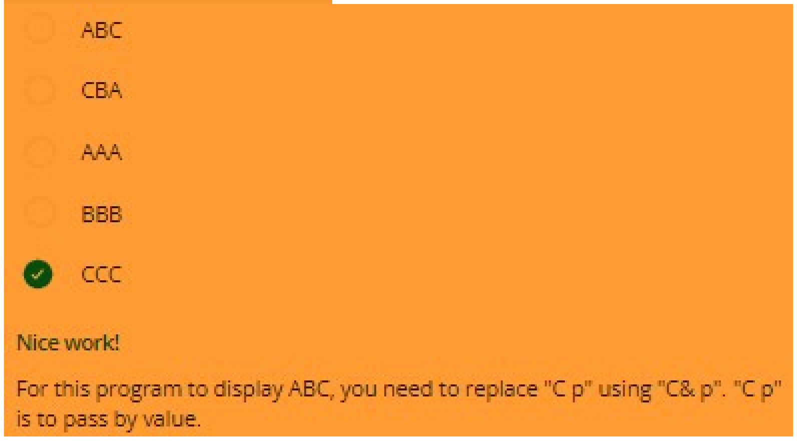
  Good job!

  For this program to display ABC, you need to place virtual for toString() in the class C.

-

- 

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

class C
{
public:
  virtual string toString()
  {
    return "C";
  }
};

class B: public C
{
public:
  string toString()
  {
    return "B";
  }
};

class A: public B
{
public:
  string toString()
  {
    return "A";
  }
};

void displayObject(C p)
{
  cout << p.toString();
}

int main()
{
  displayObject(A());
  displayObject(B());
  displayObject(C());
  return 0;
}
```

- 

ABC

CBA

AAA

BBB

[✓ CCC]

Nice work!

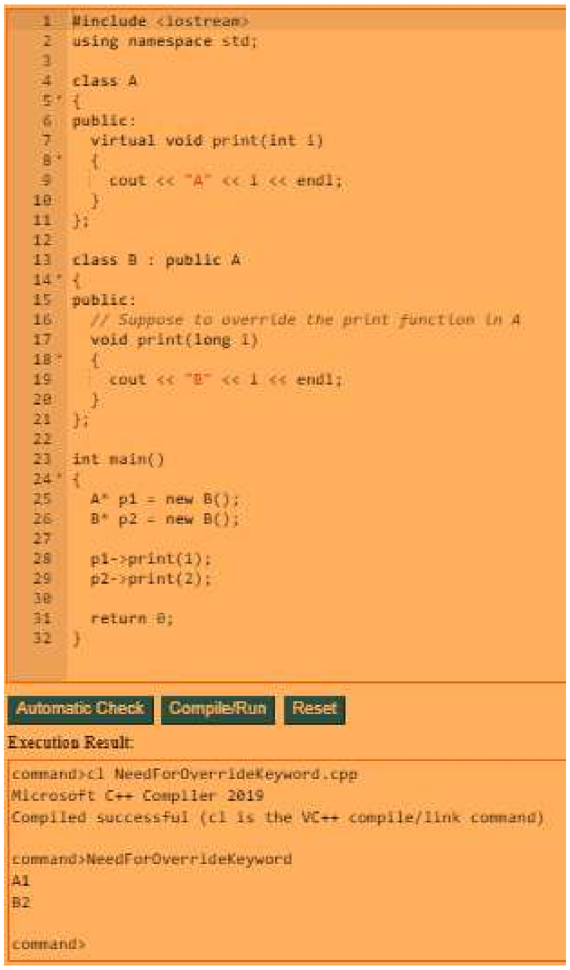
For this program to display ABC, you need to replace "C p" using "C& p". "C p" is to pass by value.

-

•

15.8 The C++11 override and final Keywords

Friday, April 21, 2023 12:00 PM

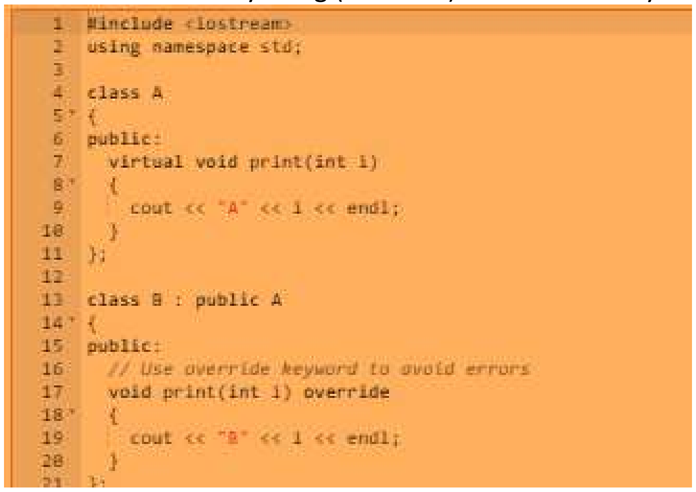
- The override keyword ensures that a fn is overridden and the final keyword prevents a fn from being overridden
- C++11 intro the override keyword to avoid pgrming errors and make sure that a fn in a derived class overrides a virtual fn in the base class
- 

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

class A
{
public:
  virtual void print(int i)
  {
    cout << "A" << i << endl;
  }
};

class B : public A
{
public:
  // Suppose to override the print function in A
  void print(long i)
  {
    cout << "B" << i << endl;
  }
};

int main()
{
  A* p1 = new B();
  B* p2 = new B();

  p1->print(1);
  p2->print(2);

  return 0;
}
```

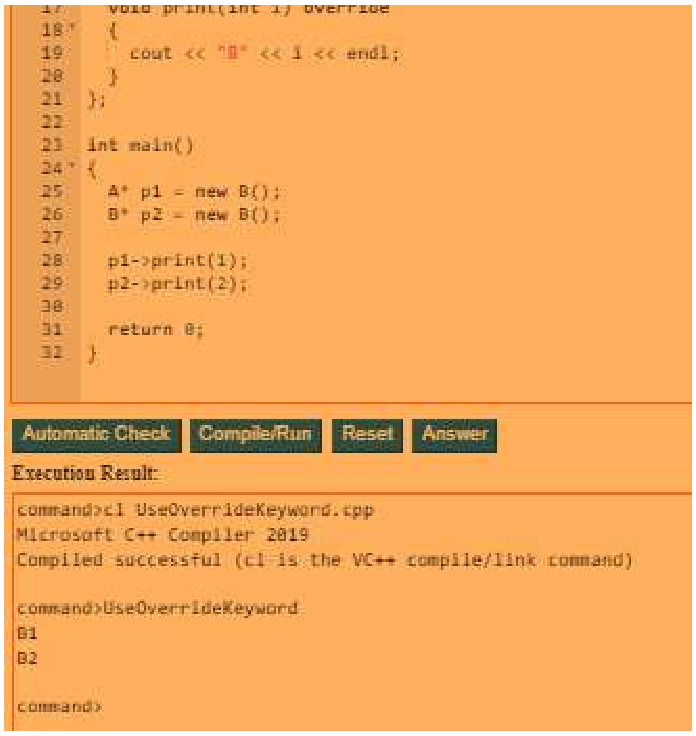
Execution Result:

```
command>cl NeedForOverrideKeyword.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>NeedForOverrideKeyword
A1
B2

command>
```

- A virtual print fn defined in the base class A is supposed to be overridden in the derived class B
- But its done wrong bc these 2 print fns have diff signatures
- The param I in the print fn in B is of the long type, but is of the int type in A
- Avoid these errors by using (in C++11) the override keyword



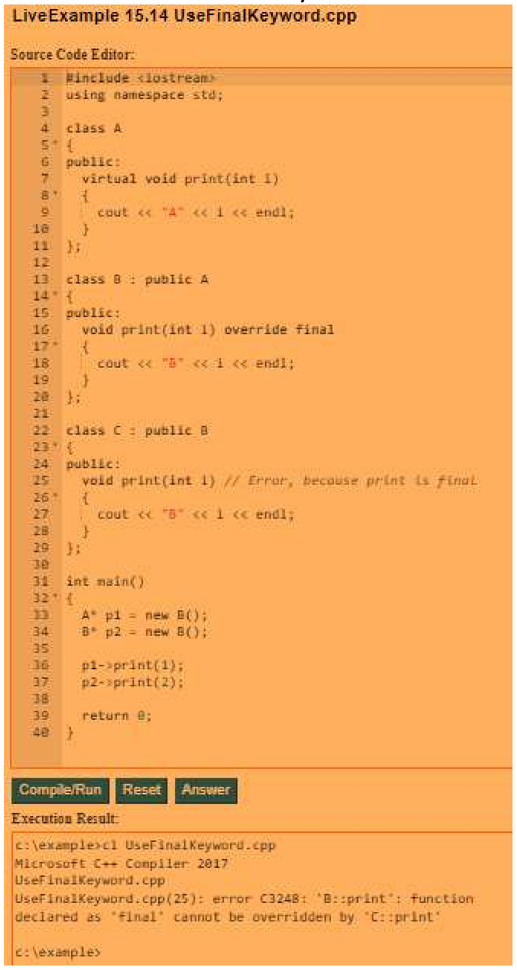
```cpp
#include <iostream>
using namespace std;

class A
{
public:
  virtual void print(int i)
  {
    cout << "A" << i << endl;
  }
};

class B : public A
{
public:
  // Use override keyword to avoid errors
  void print(int i) override
  {
    cout << "B" << i << endl;
  }
};
```

- 

```
    void print(int i) override
    {
      cout << "B" << i << endl;
    }
};

int main()
{
  A* p1 = new B();
  B* p2 = new B();

  p1->print(1);
  p2->print(2);

  return 0;
}
```

Execution Result:

```
command>cl UseOverrideKeyword.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>UseOverrideKeyword
B1
B2

command>
```

- The override keyword tells the compiler to check that the fn in the derived class is overriding a virtual fn in the base class
- If the overridden fn in the derived class doesn't match the virtual fn defined in the base class, the compiler would report the error
- C++ also into the final keyword that can be used to prevent a virtual fn from being overridden
- 

LiveExample 15.14 UseFinalKeyword.cpp

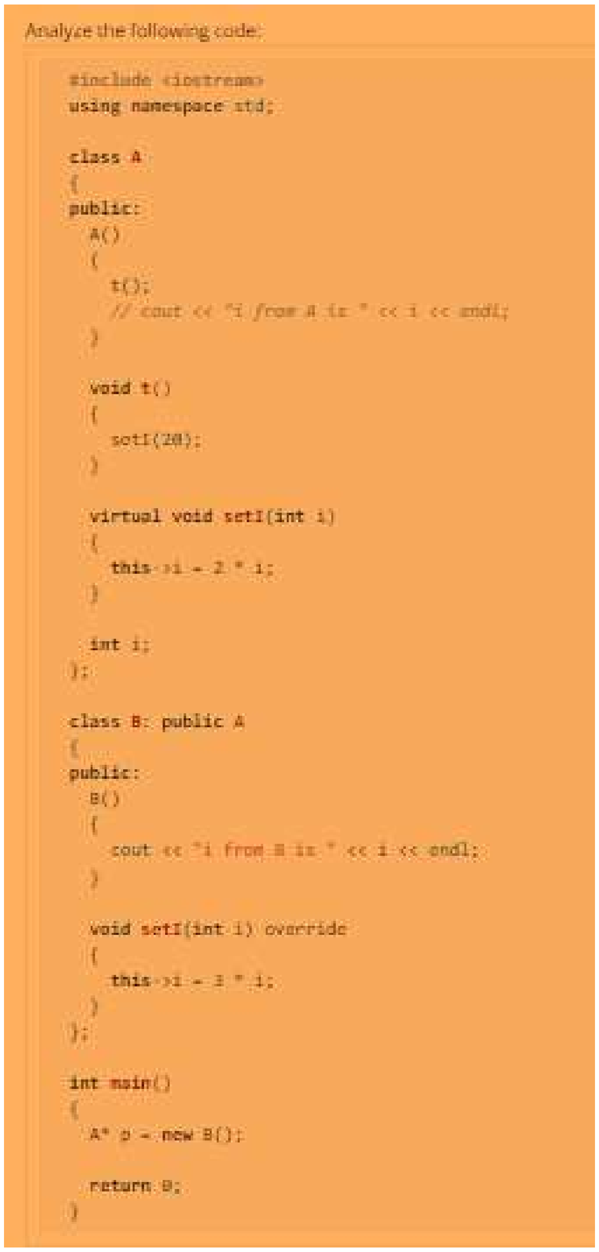
Source Code Editor:

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  class A
 5  {
 6  public:
 7    virtual void print(int i)
 8    {
 9      cout << "A" << i << endl;
10    }
11  };
12
13  class B : public A
14  {
15  public:
16    void print(int i) override final
17    {
18      cout << "B" << i << endl;
19    }
20  };
21
22  class C : public B
23  {
24  public:
25    void print(int i) // Error, because print is final
26    {
27      cout << "B" << i << endl;
28    }
29  };
30
31  int main()
32  {
33    A* p1 = new B();
34    B* p2 = new B();
35
36    p1->print(1);
37    p2->print(2);
38
39    return 0;
40  }
```

Execution Result:

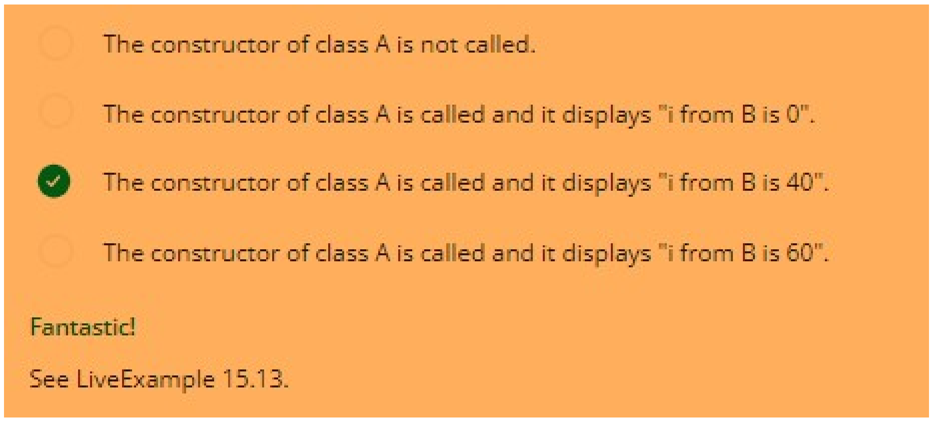
```
c:\example>cl UseFinalKeyword.cpp
Microsoft C++ Compiler 2017
UseFinalKeyword.cpp
UseFinalKeyword.cpp(25): error C3248: 'B::print': function
declared as 'final' cannot be overridden by 'C::print'

c:\example>
```

- The pgrm has a syntax error in ln 25 bc the print fn is declared final in line 16

- The final keyword is only used w/ virtual fns, print is a virtual fn og defined in class A
- 

Analyze the following code:

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

class A
{
public:
  A()
  {
    t();
    // cout << "i from A is " << i << endl;
  }

  void t()
  {
    setI(20);
  }

  virtual void setI(int i)
  {
    this->i = 2 * i;
  }

  int i;
};

class B: public A
{
public:
  B()
  {
    cout << "i from B is " << i << endl;
  }

  void setI(int i) override
  {
    this->i = 3 * i;
  }
};

int main()
{
  A* p = new B();

  return 0;
}
```

- 

The constructor of class A is not called.

The constructor of class A is called and it displays "i from B is 0".

✓ The constructor of class A is called and it displays "i from B is 40".

The constructor of class A is called and it displays "i from B is 60".

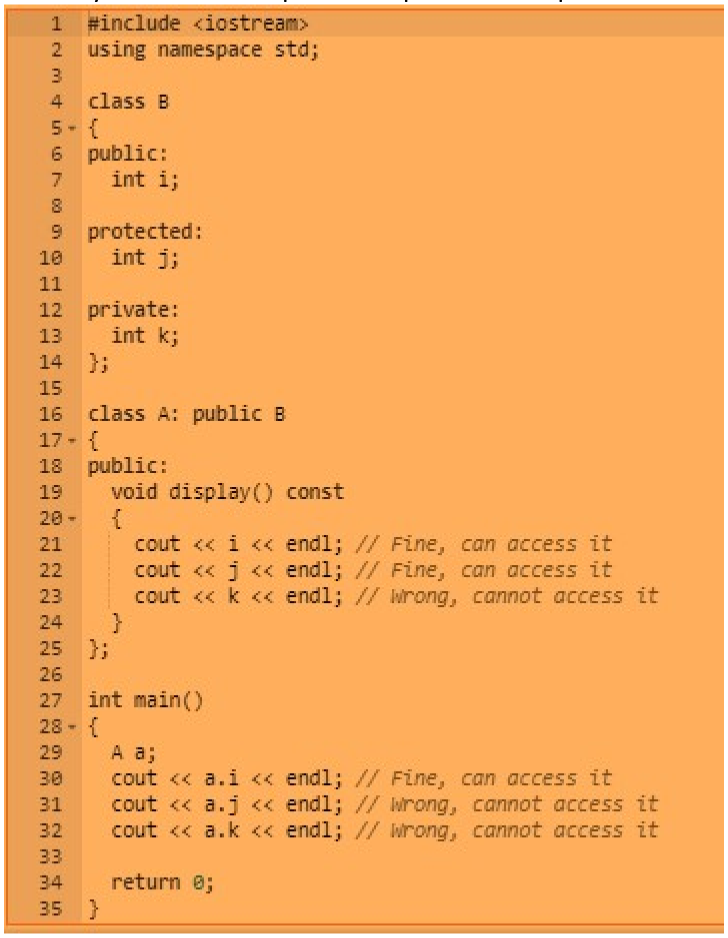
Fantastic!

See LiveExample 15.13.

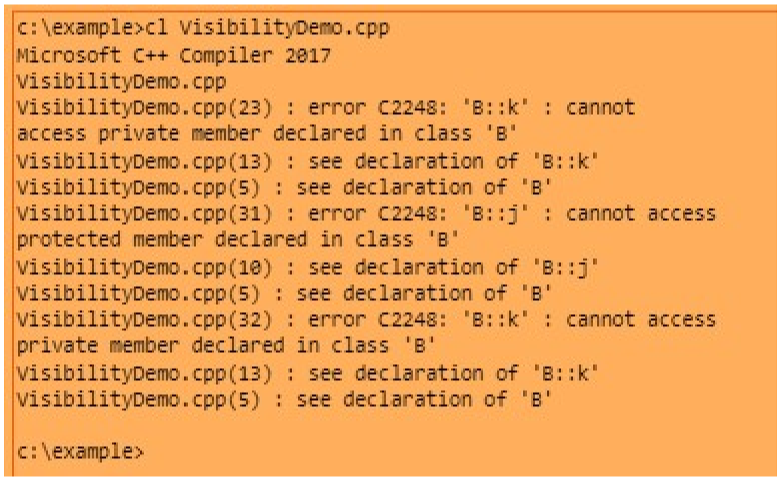
-

15.9 The protected Keyword

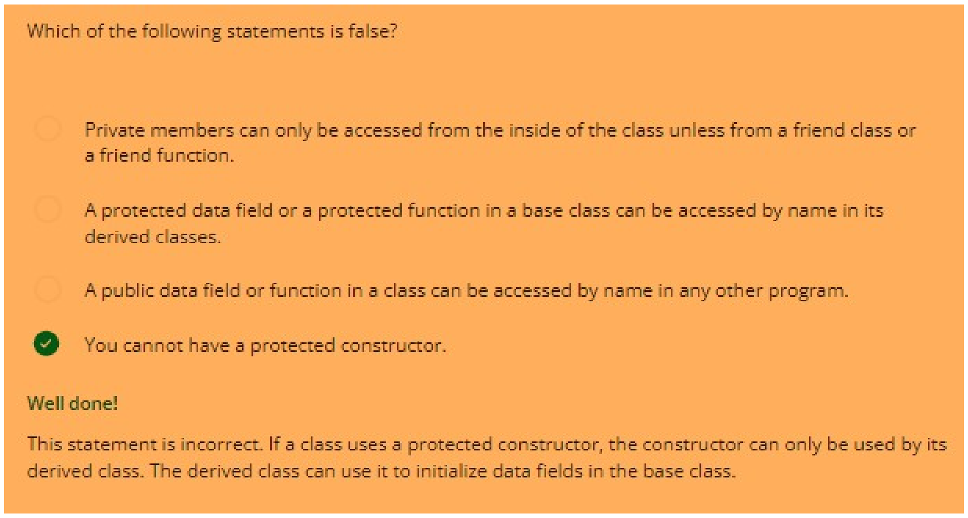
Friday, April 21, 2023 12:24 PM

- The protected member of a class can be accessed from a derived class
- Have used private and public keywords to specify data fields and fns can be accessed from outside the class
- Private membs can be accessed only from inside the class / from friend fns and friend classes
- Public membs can be accessed from any other classes
- Usually good to allow derived classes to access data fields / fns defined in the base class but not allow nonderived classes to do so
- Can use protected keyword
- Protected data field / protected fn in a base class can be accessed in its derived classes
- Keywords private, protected, and public aka visibility / accessibility keywords bc specify how class and class membs are accessed
- Visibility increase from private -> protected -> public
- 

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

class B
{
public:
  int i;

protected:
  int j;

private:
  int k;
};

class A: public B
{
public:
  void display() const
  {
    cout << i << endl; // Fine, can access it
    cout << j << endl; // Fine, can access it
    cout << k << endl; // Wrong, cannot access it
  }
};

int main()
{
  A a;
  cout << a.i << endl; // Fine, can access it
  cout << a.j << endl; // Wrong, cannot access it
  cout << a.k << endl; // Wrong, cannot access it

  return 0;
}
```

- 

```
c:\example>cl VisibilityDemo.cpp
Microsoft C++ Compiler 2017
VisibilityDemo.cpp
VisibilityDemo.cpp(23) : error C2248: 'B::k' : cannot
access private member declared in class 'B'
VisibilityDemo.cpp(13) : see declaration of 'B::k'
VisibilityDemo.cpp(5) : see declaration of 'B'
VisibilityDemo.cpp(31) : error C2248: 'B::j' : cannot access
protected member declared in class 'B'
VisibilityDemo.cpp(10) : see declaration of 'B::j'
VisibilityDemo.cpp(5) : see declaration of 'B'
VisibilityDemo.cpp(32) : error C2248: 'B::k' : cannot access
private member declared in class 'B'
VisibilityDemo.cpp(13) : see declaration of 'B::k'
VisibilityDemo.cpp(5) : see declaration of 'B'

c:\example>
```

- Since A is derived from B and j is protected, j can be accessed from class A in ln 22
- since k is private, k cannot be accessed from class A in ln 23
- Since i is public, i can be accessed from a.i in ln 30
- Since j and k aren't public, cannot be accessed from the object a in lns 31-32
-
-
- 

Which of the following statements is false?

Private members can only be accessed from the inside of the class unless from a friend class or a friend function.

A protected data field or a protected function in a base class can be accessed by name in its derived classes.

A public data field or function in a class can be accessed by name in any other program.

[✓ You cannot have a protected constructor.]

Well done!

This statement is incorrect. If a class uses a protected constructor, the constructor can only be used by its derived class. The derived class can use it to initialize data fields in the base class.

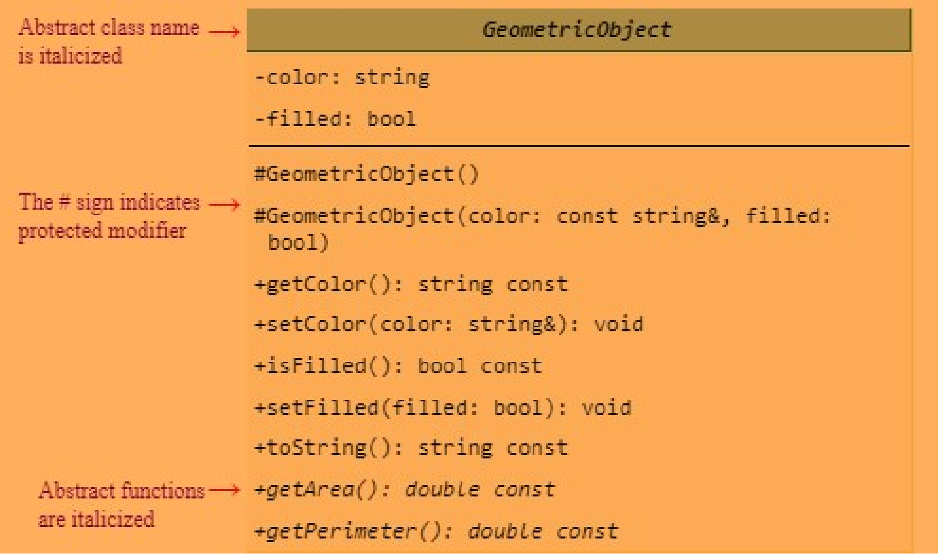
-

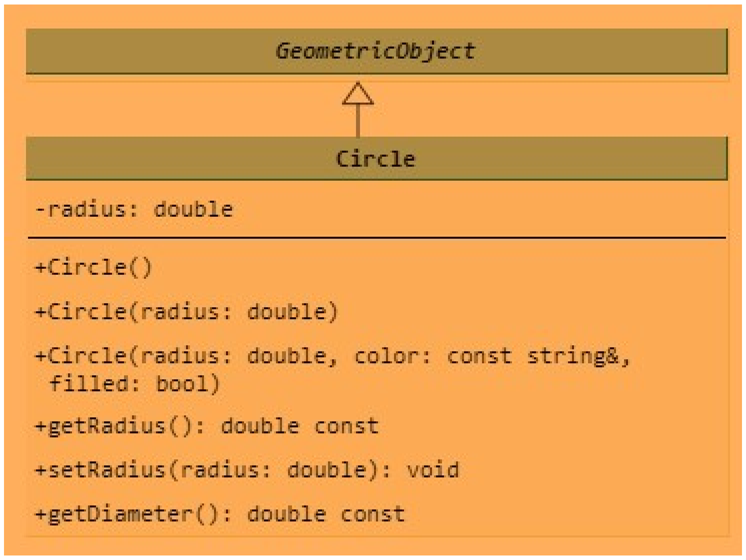
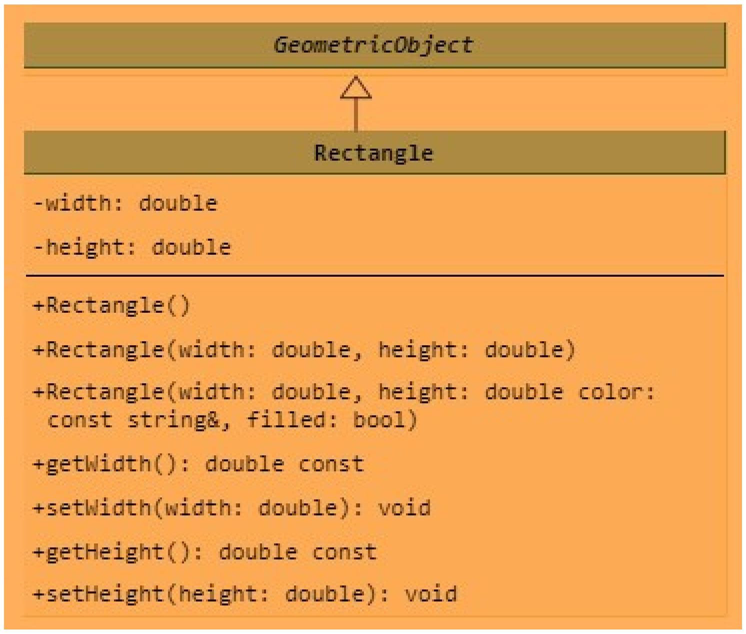
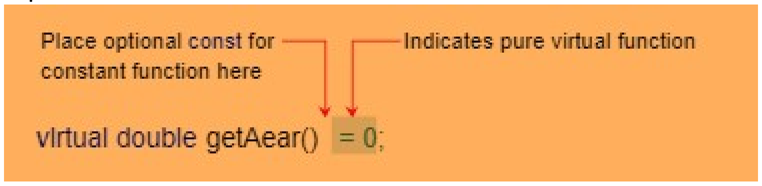
15.10 Abstract Classes and Pure Virtual Functions

Friday, April 21, 2023 12:35 PM

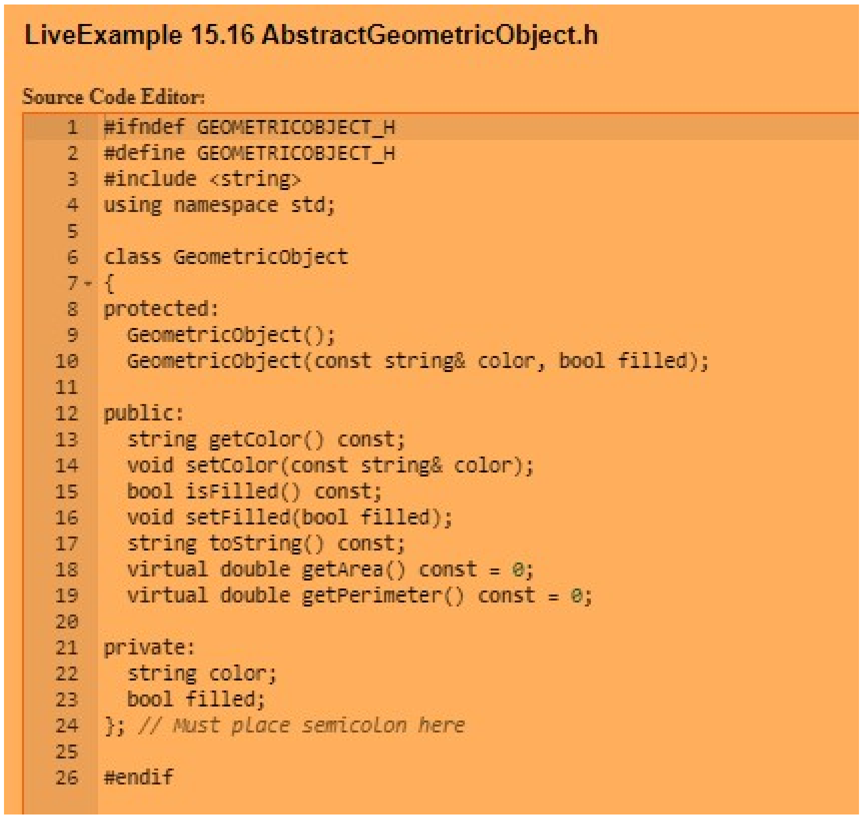
- An abstract class cannot be used to make objects, an abstract class can have abstract fns, which are implemented in concrete derived classes
- In the inheritance hierarchy, classes become more specific and concrete w/ each new derived class
- If move from a derived class back up to parent and ancestor classes, the classes become more general and less specific
- Class design should ensure that a base class has common features of its derived classes, sometimes a base class is so abstract that it cannot have any specific instances, aka abstract class
- In the pgrm w/ GeometricObject defined, it's the base class for Circle and Rectangle
- GeometricObject models common features of geometric objects
- Both Circle and Rectangle have the getArea() and getPerimeter() fns for computing the area and perimeter of a circle and a rectangle
- Since can compute areas and perimeters for all geometric objects, better to define the getArea() and getPerimeter() fns in the GeometricObject class
- But, these fns cannot be implemented in the GeometricObject class bc their implementation is dependent on the specific type of geometric object, aka abstract fns
- After define the abstract fns in GeometricObject, GeometricObject becomes an abstract class
- New GeometricObject class is below, UML names of abstract class and abstract fns are italicized

Figure 15.3

- 
Abstract class name is italicized
GeometricObject
-color: string
-filled: bool
#GeometricObject()
The # sign indicates protected modifier
#GeometricObject(color: const string&, filled: bool)
+getColor(): string const
+setColor(color: string&): void
+isFilled(): bool const
+setFilled(filled: bool): void
+toString(): string const
Abstract functions are italicized
+getArea(): double const
+getPerimeter(): double const
-

- 
GeometricObject
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: const string&, filled: bool)
+getRadius(): double const
+setRadius(radius: double): void
+getDiameter(): double const
-
- 
GeometricObject
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: const string&, filled: bool)
+getWidth(): double const
+setWidth(width: double): void
+getHeight(): double const
+setHeight(height: double): void
- In C++, abstract fns called pure virtual fns, a class that has pure virtual fns becomes an abstract class
- A pure virtual fn defined like
- 
Place optional const for constant function here
Indicates pure virtual function

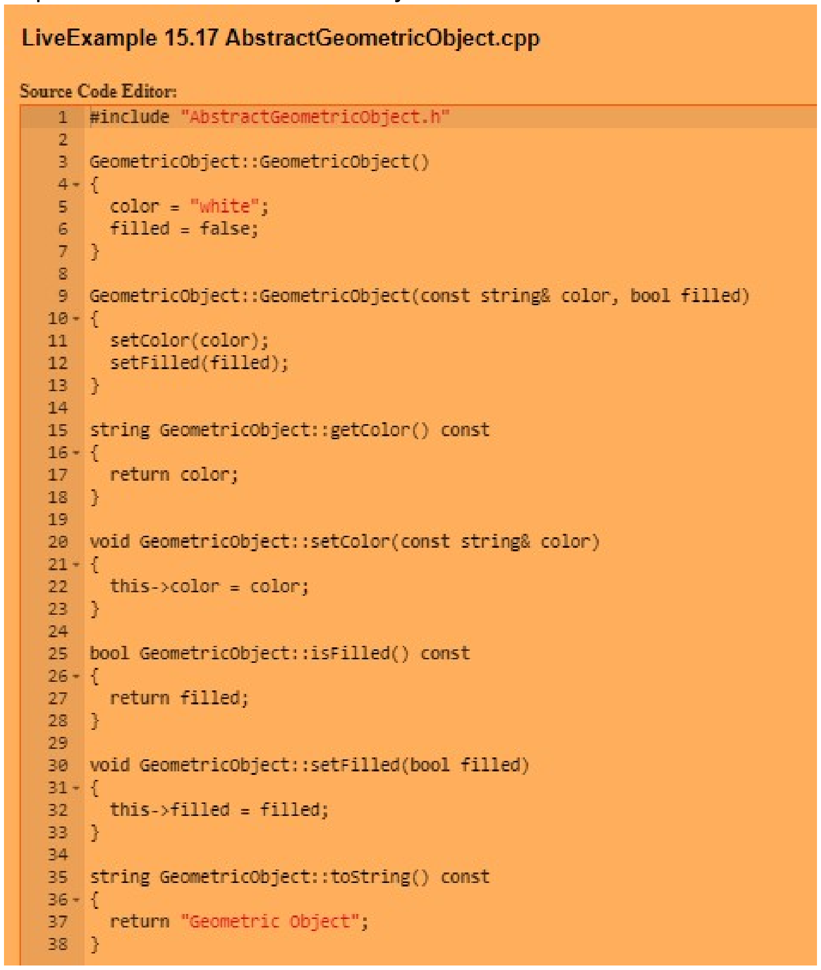
```
virtual double getAear() = 0;
```
- The =0 notation indicates the getArea is a pure virtual fn, a pure virtual fn does not have a body / implementation in the base class
- Defined new abstract GeometricObject class w/ 2 pure virtual fns (lns 18-19)

- 

LiveExample 15.16 AbstractGeometricObject.h

Source Code Editor:

```
#ifndef GEOMETRICOBJECT_H
#define GEOMETRICOBJECT_H
#include <string>
using namespace std;

class GeometricObject
{
protected:
  GeometricObject();
  GeometricObject(const string& color, bool filled);

public:
  string getColor() const;
  void setColor(const string& color);
  bool isFilled() const;
  void setFilled(bool filled);
  string toString() const;
  virtual double getArea() const = 0;
  virtual double getPerimeter() const = 0;

private:
  string color;
  bool filled;
}; // Must place semicolon here

#endif
```

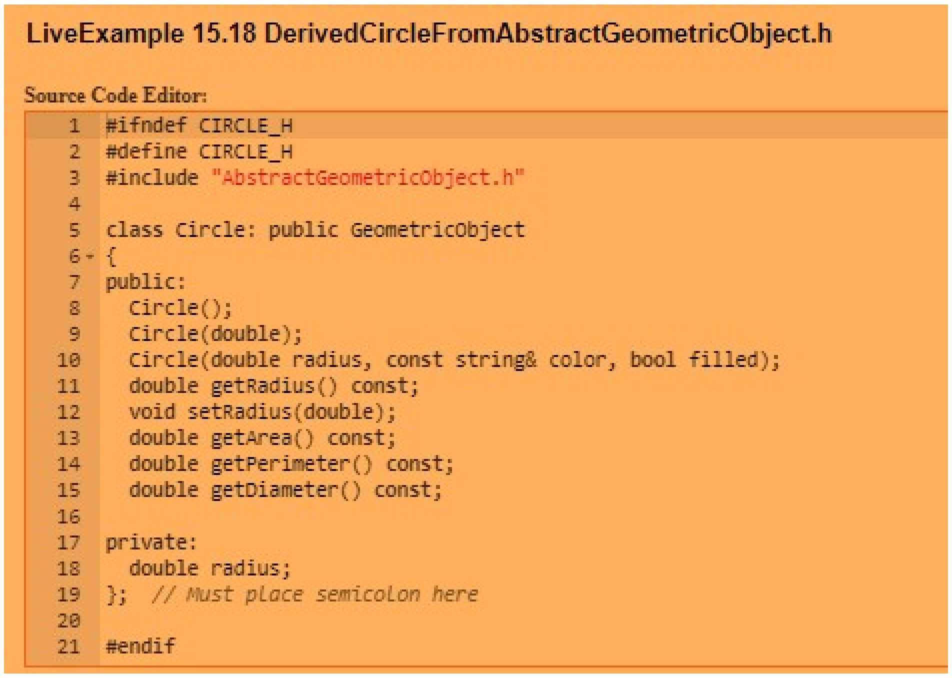
- GeometricObject is just like reg class, except cannot make objects from it bc its an abstract class
- If attempt to make an object from GeometricObject, compiler reports an error
- Implementation of GeometricObject class
- 

LiveExample 15.17 AbstractGeometricObject.cpp

Source Code Editor:

```
#include "AbstractGeometricObject.h"

GeometricObject::GeometricObject()
{
  color = "white";
  filled = false;
}

GeometricObject::GeometricObject(const string& color, bool filled)
{
  setColor(color);
  setFilled(filled);
}

string GeometricObject::getColor() const
{
  return color;
}

void GeometricObject::setColor(const string& color)
{
  this->color = color;
}

bool GeometricObject::isFilled() const
{
  return filled;
}

void GeometricObject::setFilled(bool filled)
{
  this->filled = filled;
}

string GeometricObject::toString() const
{
  return "Geometric Object";
}
```

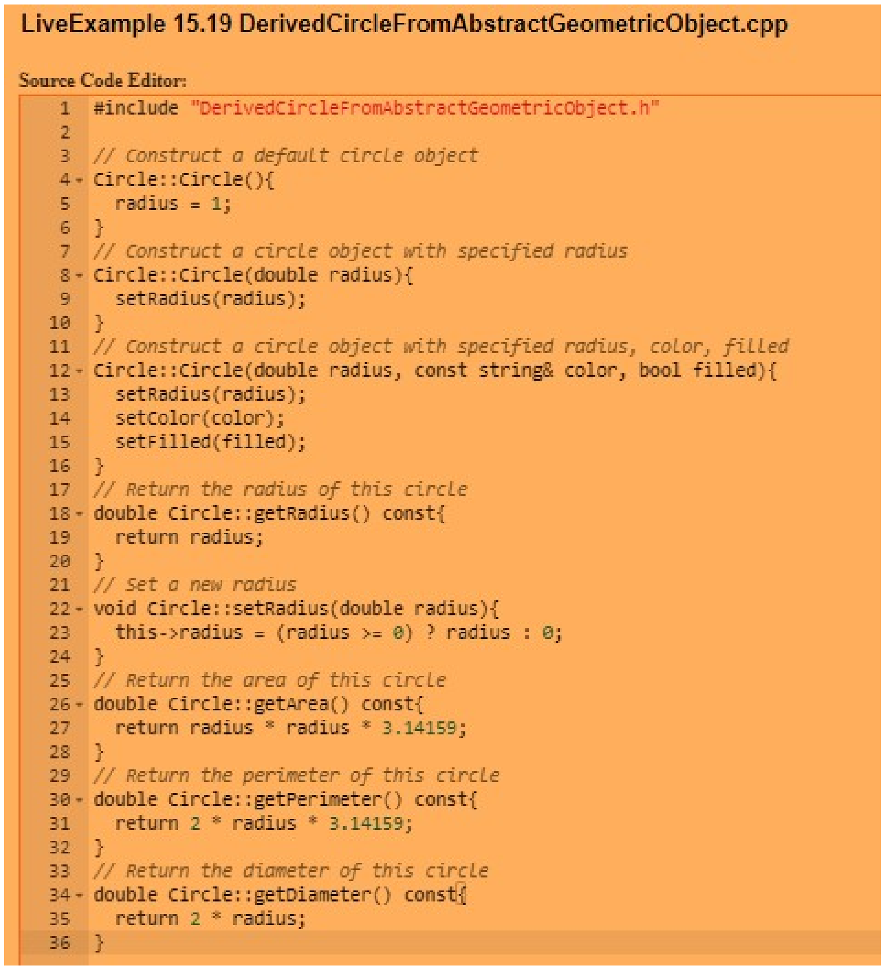
- Now for the Circle and rectangle classes derived from the abstract GeometricObject

- 

LiveExample 15.18 DerivedCircleFromAbstractGeometricObject.h

Source Code Editor:

```cpp
#ifndef CIRCLE_H
#define CIRCLE_H
#include "AbstractGeometricObject.h"

class Circle: public GeometricObject
{
public:
  Circle();
  Circle(double);
  Circle(double radius, const string& color, bool filled);
  double getRadius() const;
  void setRadius(double);
  double getArea() const;
  double getPerimeter() const;
  double getDiameter() const;

private:
  double radius;
};  // Must place semicolon here

#endif
```

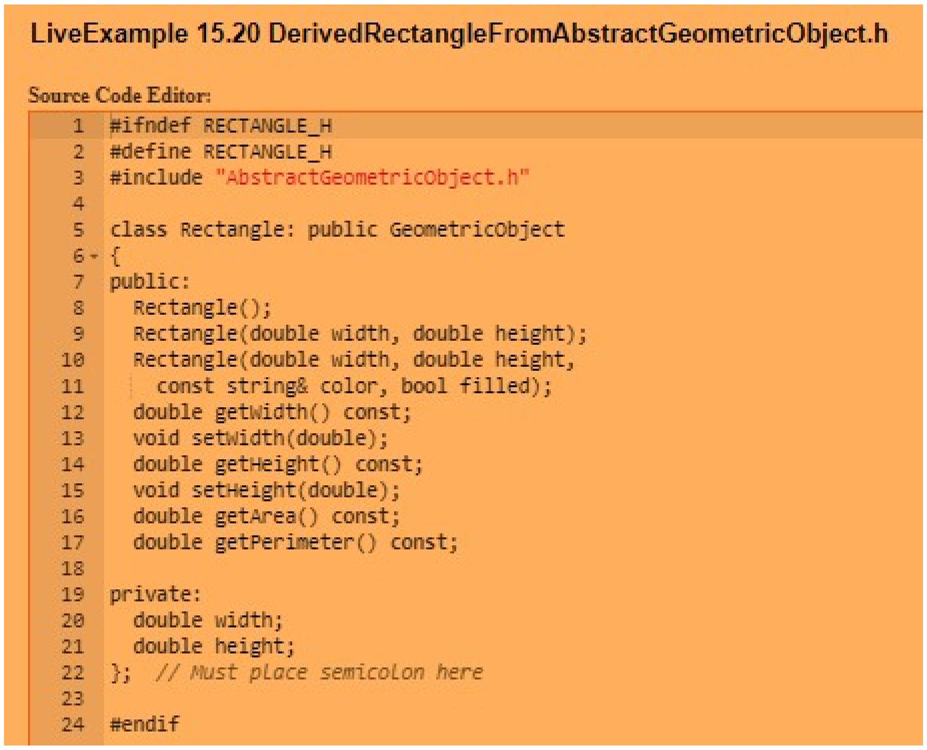
- 

LiveExample 15.19 DerivedCircleFromAbstractGeometricObject.cpp

Source Code Editor:

```cpp
#include "DerivedCircleFromAbstractGeometricObject.h"

// Construct a default circle object
Circle::Circle(){
  radius = 1;
}
// Construct a circle object with specified radius
Circle::Circle(double radius){
  setRadius(radius);
}
// Construct a circle object with specified radius, color, filled
Circle::Circle(double radius, const string& color, bool filled){
  setRadius(radius);
  setColor(color);
  setFilled(filled);
}
// Return the radius of this circle
double Circle::getRadius() const{
  return radius;
}
// Set a new radius
void Circle::setRadius(double radius){
  this->radius = (radius >= 0) ? radius : 0;
}
// Return the area of this circle
double Circle::getArea() const{
  return radius * radius * 3.14159;
}
// Return the perimeter of this circle
double Circle::getPerimeter() const{
  return 2 * radius * 3.14159;
}
// Return the diameter of this circle
double Circle::getDiameter() const{
  return 2 * radius;
}
```

-

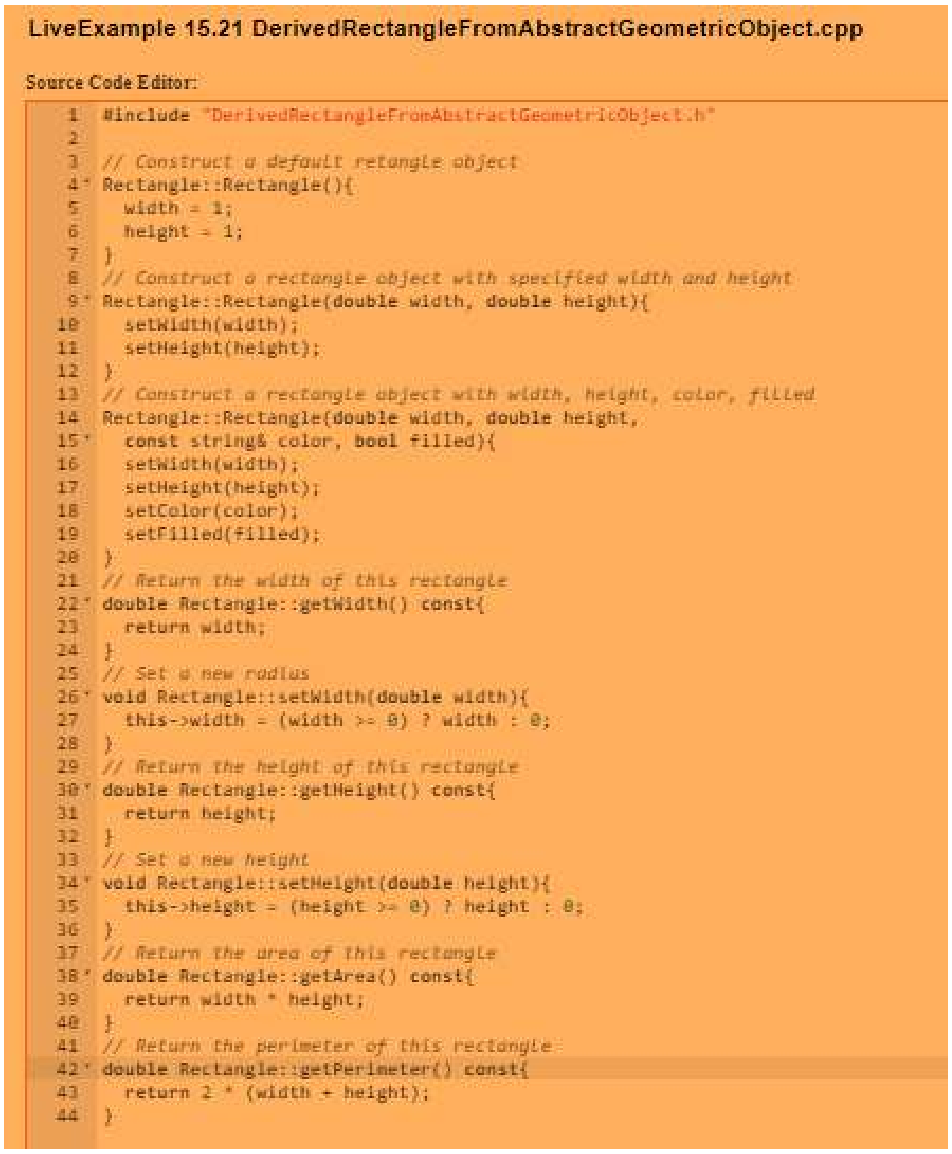
- 

LiveExample 15.20 DerivedRectangleFromAbstractGeometricObject.h

Source Code Editor:

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
#include "AbstractGeometricObject.h"

class Rectangle: public GeometricObject
{
public:
  Rectangle();
  Rectangle(double width, double height);
  Rectangle(double width, double height,
    const string& color, bool filled);
  double getWidth() const;
  void setWidth(double);
  double getHeight() const;
  void setHeight(double);
  double getArea() const;
  double getPerimeter() const;

private:
  double width;
  double height;
};  // Must place semicolon here

#endif
```

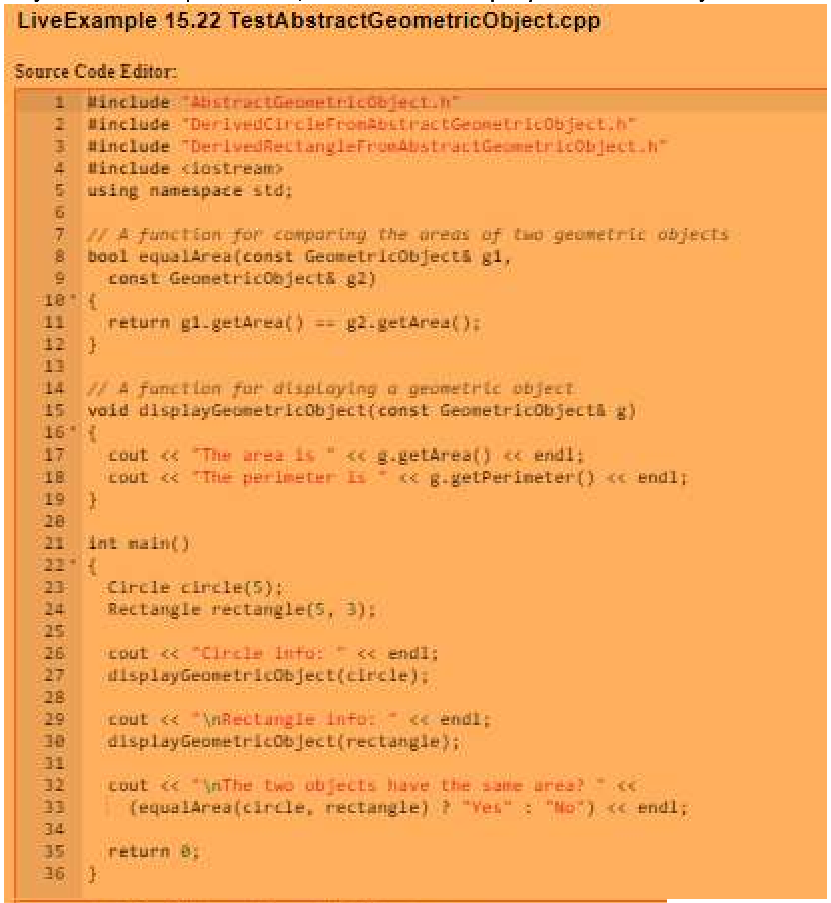
-

- 

LiveExample 15.21 DerivedRectangleFromAbstractGeometricObject.cpp

Source Code Editor:

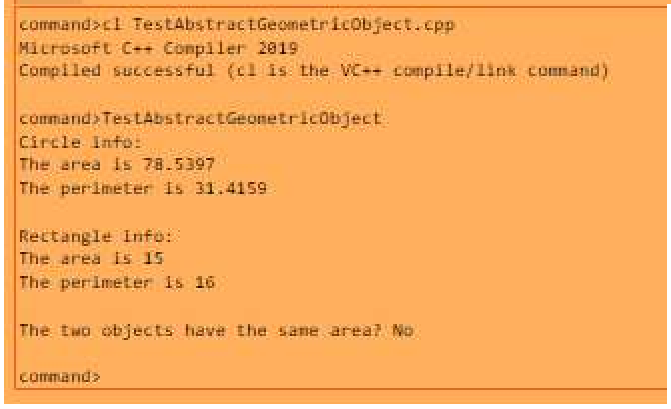
```
#include "DerivedRectangleFromAbstractGeometricObject.h"

// Construct a default retangle object
Rectangle::Rectangle(){
  width = 1;
  height = 1;
}
// Construct a rectangle object with specified width and height
Rectangle::Rectangle(double width, double height){
  setWidth(width);
  setHeight(height);
}
// Construct a rectangle object with width, height, color, filled
Rectangle::Rectangle(double width, double height,
  const string& color, bool filled){
  setWidth(width);
  setHeight(height);
  setColor(color);
  setFilled(filled);
}
// Return the width of this rectangle
double Rectangle::getWidth() const{
  return width;
}
// Set a new radius
void Rectangle::setWidth(double width){
  this->width = (width >= 0) ? width : 0;
}
// Return the height of this rectangle
double Rectangle::getHeight() const{
  return height;
}
// Set a new height
void Rectangle::setHeight(double height){
  this->height = (height >= 0) ? height : 0;
}
// Return the area of this rectangle
double Rectangle::getArea() const{
  return width * height;
}
// Return the perimeter of this rectangle
double Rectangle::getPerimeter() const{
  return 2 * (width + height);
}
```

- ? Abstract fns getArea and getPerimeter should be removed from the GeometricOBject class?
- Benefit of defining them in the GeometricObject class
- Ex pgrm makes 2 geometric objects (circle and a rectangle), invokes the equalArea fn to check if 2 objects have equal areas, invokes the displayGeometricObject fn to display the objects
- 

LiveExample 15.22 TestAbstractGeometricObject.cpp

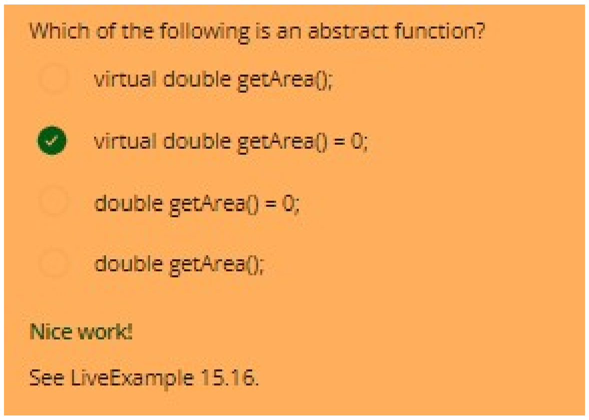
Source Code Editor:

```
 1  #include "AbstractGeometricObject.h"
 2  #include "DerivedCircleFromAbstractGeometricObject.h"
 3  #include "DerivedRectangleFromAbstractGeometricObject.h"
 4  #include <iostream>
 5  using namespace std;
 6
 7  // A function for comparing the areas of two geometric objects
 8  bool equalArea(const GeometricObject& g1,
 9    const GeometricObject& g2)
10  {
11    return g1.getArea() == g2.getArea();
12  }
13
14  // A function for displaying a geometric object
15  void displayGeometricObject(const GeometricObject& g)
16  {
17    cout << "The area is " << g.getArea() << endl;
18    cout << "The perimeter is " << g.getPerimeter() << endl;
19  }
20
21  int main()
22  {
23    Circle circle(5);
24    Rectangle rectangle(5, 3);
25
26    cout << "Circle info: " << endl;
27    displayGeometricObject(circle);
28
29    cout << "\nRectangle info: " << endl;
30    displayGeometricObject(rectangle);
31
32    cout << "\nThe two objects have the same area? " <<
33      (equalArea(circle, rectangle) ? "Yes" : "No") << endl;
34
35    return 0;
36  }
```

- 

```
command>cl TestAbstractGeometricObject.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>TestAbstractGeometricObject
Circle info:
The area is 78.5397
The perimeter is 31.4159

Rectangle info:
The area is 15
The perimeter is 16

The two objects have the same area? No

command>
```

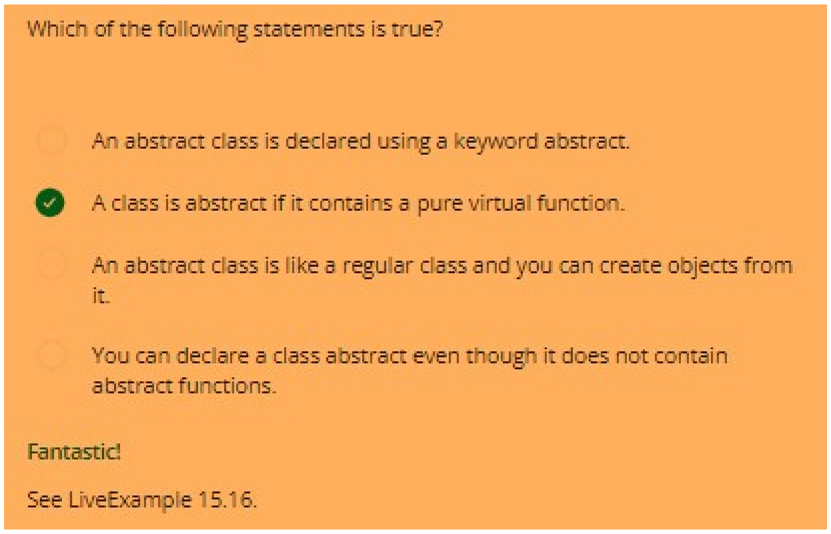
- The pgrm makes a Circle object and a Rectangle object (lns 23-24)
- The pure virtual fns getArea() and getPerimeter(0 defined in the GeometricObject class are overridden in the Circle class and the Rectangle class
- When invoking displayGeometricObject(circle) (ln 27), the fns getArea and getPerimeter defined in the Circle class are used, and when invoking displayGeometricObject(rectangle) (ln 30), the fns getArea and getPerimeter defined in the Rectangle class are used
- C++ dynamically determines which of these fns to invoke at runtime, depending on type of object
- Same when invoking equalArea(circle, rectangle) (ln 33), the getArea fn defined in the Circle class used for g1.getArea(), since g1 is a circle
- Also, the getArea fn defined in the Rectangle class is used for g2.getArea(), since g2 is a rectangle
- Note that if the getArea and getPerimeter fns were not defined in GeometricObject, u cannot define the equalArea and displayGeometricObject fns in this pgrm
- So yeah, benefit for defining abstract fns in GeometricObject

-

Which of the following is an abstract function?

- virtual double getArea();
- [✓ virtual double getArea() = 0;]
- double getArea() = 0;
- double getArea();

Nice work!

See LiveExample 15.16.

-

Which of the following statements is true?

- An abstract class is declared using a keyword abstract.
- [✓ A class is abstract if it contains a pure virtual function.]
- An abstract class is like a regular class and you can create objects from it.
- You can declare a class abstract even though it does not contain abstract functions.

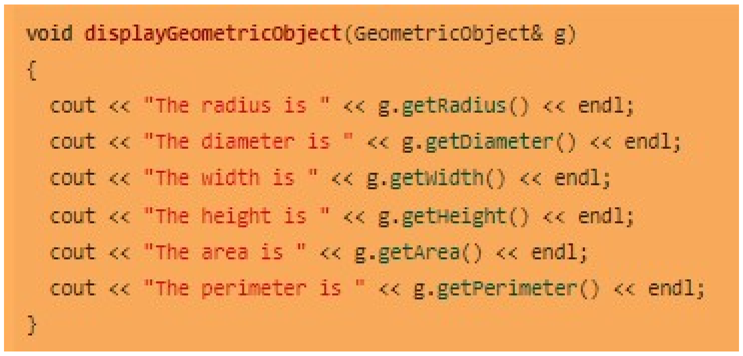
Fantastic!

See LiveExample 15.16.

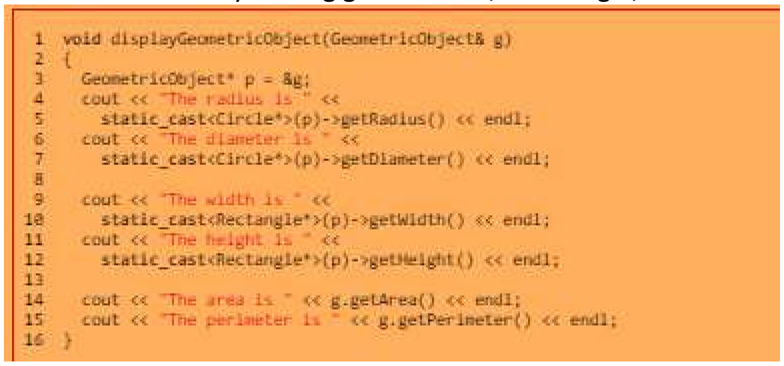
-

15.11 Casting: static_cast VS dynamic_cast

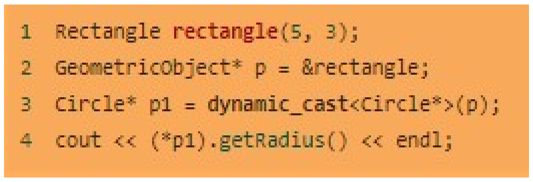
Saturday, April 22, 2023 5:27 PM

- The dynamic_cast operator can be used to cast an object to its actual type at runtime
- If want to rewrite the displayGeometricObject fn from b4, TestAbstractGeometricObject.cpp, to display the radius and diameter for a circle object and the width and height for a rectangle object, don't do this:
- 

```
void displayGeometricObject(GeometricObject& g)
{
  cout << "The radius 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;
}
```

- Bc 2 probs
  - First, code cannot be compiled bc g's type is GeometricObject, but the GeometricObject class does not contain the getRadius(), getDiameter(), getWidth(), and getHeight() fns
  - Second, code should detect whether the geometric object is a circle or a rectangle and then display radius and diameter for a circle and width and height for a rectangle
- Prob can be fixed by casting g into Circle / Rectangle, like
- 

```
1  void displayGeometricObject(GeometricObject& g)
2  {
3    GeometricObject* p = &g;
4    cout << "The radius is " <<
5      static_cast<Circle*>(p)->getRadius() << endl;
6    cout << "The diameter is " <<
7      static_cast<Circle*>(p)->getDiameter() << endl;
8
9    cout << "The width is " <<
10     static_cast<Rectangle*>(p)->getWidth() << endl;
11   cout << "The height is " <<
12     static_cast<Rectangle*>(p)->getHeight() << endl;
13
14   cout << "The area is " << g.getArea() << endl;
15   cout << "The perimeter is " << g.getPerimeter() << endl;
16 }
```

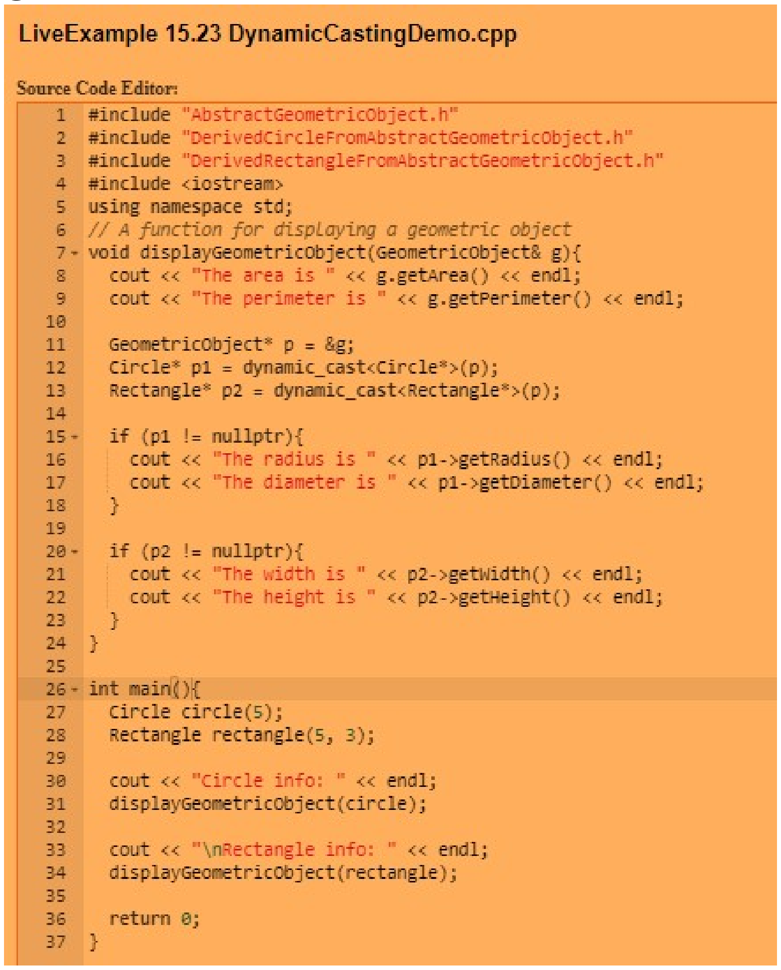
- Static casting is performed on p that points to a GeometricObject g (ln 3), this new fn can compiler, but its still wrong
- A Circle object can be cast to Rectangle to invoke getWidth() (ln 10), and Rectangle object can be cast to Circle to invoke getRadius() (ln 5)
- Need to ensure that object is really a Circle object b4 invoking getRadius(), can be done by dynamic_cast
- The dynamic_cast works like static_cast, and also does runtime checking to make sure casting is successful, if fails, it returns nullptr
- So if run
- 

```
1  Rectangle rectangle(5, 3);
2  GeometricObject* p = &rectangle;
3  Circle* p1 = dynamic_cast<Circle*>(p);
4  cout << (*p1).getRadius() << endl;
```

- The p1 will be nullptr, a runtime error will happen when running the code in ln 4
- Assigning a pointer of a derived class type to a pointer of its base class type is called upcasting and assigning a pointer of a base class type to a pointer of its derived class type is called downcasting
- Upcasting can be performed implicitly w/out using the static_cast or dynamic_cast op

- Like this is correct:

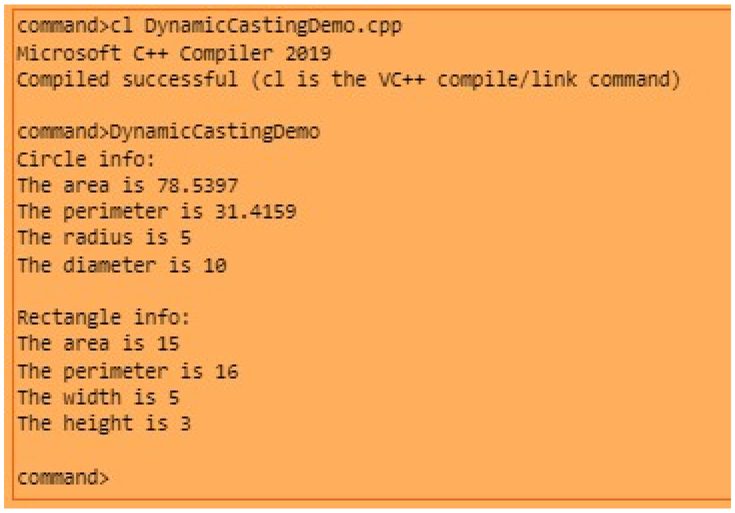
-
  ```
  GeometricObject* p = new Circle(1);
  Circle* p1 = new Circle(2);
  p = p1;
  ```
- But downcasting must be performed explicitly, like to assign p to p1, have to use

-
  ```
  p1 = static_cast<Circle*>(p);
  or p1 = dynamic_cast<Circle*>(p);
  ```
- The dynamic_cast can be done only on the pointer / a reference of a polymorphic type, like the type having a virtual fn
- The dynamic_cast can be used to check if casting is performed successfully @ runtime
- The static_cast is done @ compile time
- Now, to rewrite the displayGeometricObject fn using dynamic casting, check if casting is successful @ runtime
- 

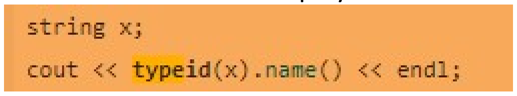
LiveExample 15.23 DynamicCastingDemo.cpp

Source Code Editor:

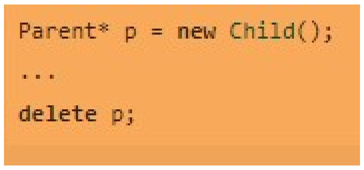
```
#include "AbstractGeometricObject.h"
#include "DerivedCircleFromAbstractGeometricObject.h"
#include "DerivedRectangleFromAbstractGeometricObject.h"
#include <iostream>
using namespace std;
// A function for displaying a geometric object
void displayGeometricObject(GeometricObject& g){
  cout << "The area is " << g.getArea() << endl;
  cout << "The perimeter is " << g.getPerimeter() << endl;

  GeometricObject* p = &g;
  Circle* p1 = dynamic_cast<Circle*>(p);
  Rectangle* p2 = dynamic_cast<Rectangle*>(p);

  if (p1 != nullptr){
    cout << "The radius is " << p1->getRadius() << endl;
    cout << "The diameter is " << p1->getDiameter() << endl;
  }

  if (p2 != nullptr){
    cout << "The width is " << p2->getWidth() << endl;
    cout << "The height is " << p2->getHeight() << endl;
  }
}

int main(){
  Circle circle(5);
  Rectangle rectangle(5, 3);

  cout << "Circle info: " << endl;
  displayGeometricObject(circle);

  cout << "\nRectangle info: " << endl;
  displayGeometricObject(rectangle);

  return 0;
}
```

- 

```
command>cl DynamicCastingDemo.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>DynamicCastingDemo
Circle info:
The area is 78.5397
The perimeter is 31.4159
The radius is 5
The diameter is 10

Rectangle info:
The area is 15
The perimeter is 16
The width is 5
The height is 3

command>
```

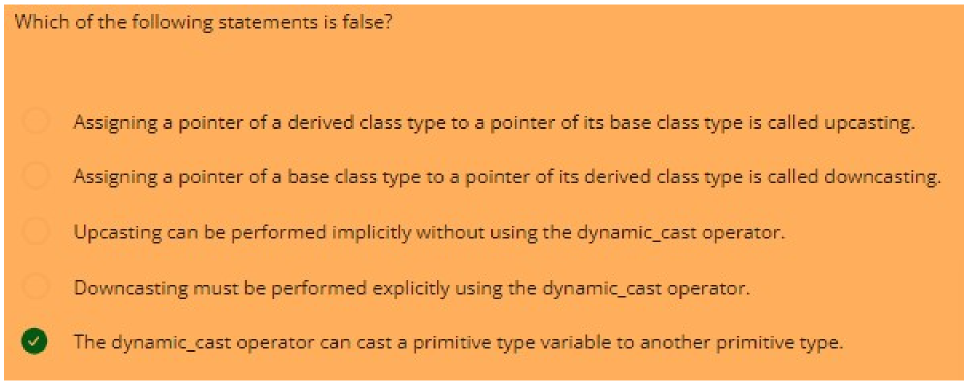
- Analyzed
  - @ ln 13 – pointer for a GeometricObject g
  - @ ln 14 – the dynamic_cast op checks if pointer p points to a Circle object, if yes, the object's address is assigned to p1, else p1 is nullptr
  - @ ln 19-20 – if p1 aint nullptr, the getRadius() and getDiameter() fns of the Circle object are invoked
  - @ ln 25-26 – same as ^ if object is rectangle and w/ width and height fns
  - @ ln 36 – pgrm invokes the displayGeometricObject fn to display a Circle object
  - @ ln 39 - ^ but for Rectangle object
  - @ ln 14 – fn casts the param g into a Circle pointer p1
  - @ ln 15 - ^ but for Rectangle pointer
  - @ ln 19-20 – If it's a Circle object, the object's getRadius() and getDiameter() fns invoked
  - @ ln 25-26 – if its a Rectangle object, the object's getWidth() and getHeight(0 fns are invoked
  - @ ln 10-11 – fns also invokesGeometricObject's getArea() and getPerimeter() fns, since both these defined in GeometricObject clss, no need to downcast object param to Circle / Rectangle to invoke them
- Occasionally, usefule to get info abt the class of object, can use the typeid operator to return a reference to an object of class type_info
  - Ex: if want use this to display the class name for object x:
  - 

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

  - It displays string bc x is an object of the string class
  - To use the typeid op, the pgrm must include the <typeinfo> header file
- If a class is used polymorphic w/ dynamic mem allocations, its destructor should be defined virtual
- If class Child is derived from class Parent and destructors are not virtual
- So like
- 

  ```
  Parent* p = new Child();
  ...
  delete p;
  ```

- When delete is invoked w/ p, Parent's destructor is classed, since p is declared a pointer for Parent
- p actually pts to an object of Child, but Child's destructor is never called
- To fix prob, define the destructor virtual in class Parent, no when delete is invoked w/ p, Child's destructor is called and the Parent's destructor is called, bc destructors are virtual
-

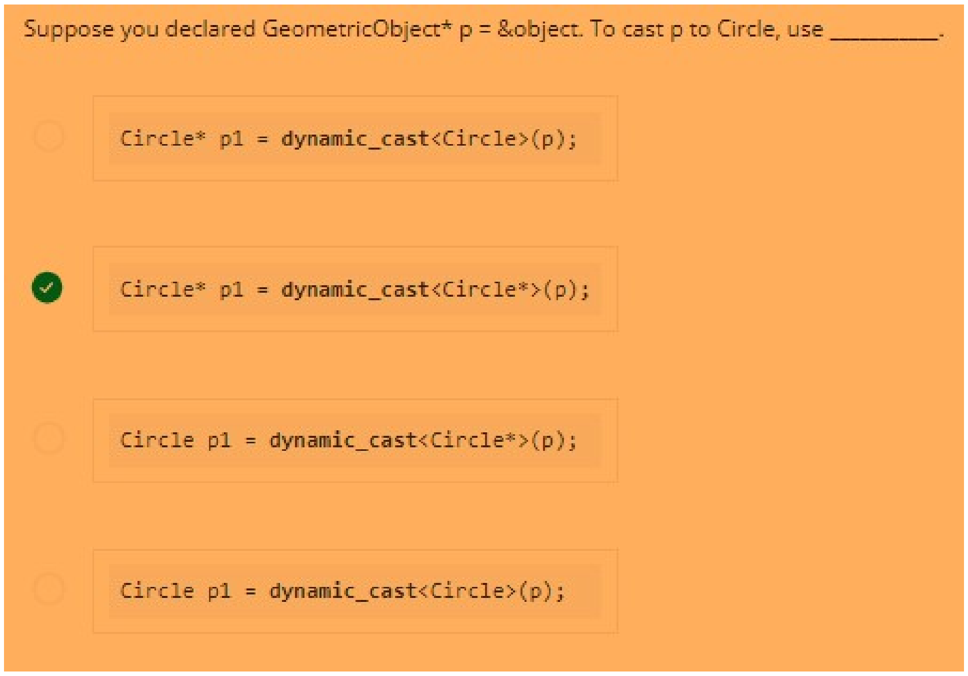
- 
  Which of the following statements is false?

  Assigning a pointer of a derived class type to a pointer of its base class type is called upcasting.

  Assigning a pointer of a base class type to a pointer of its derived class type is called downcasting.

  Upcasting can be performed implicitly without using the dynamic_cast operator.

  Downcasting must be performed explicitly using the dynamic_cast operator.

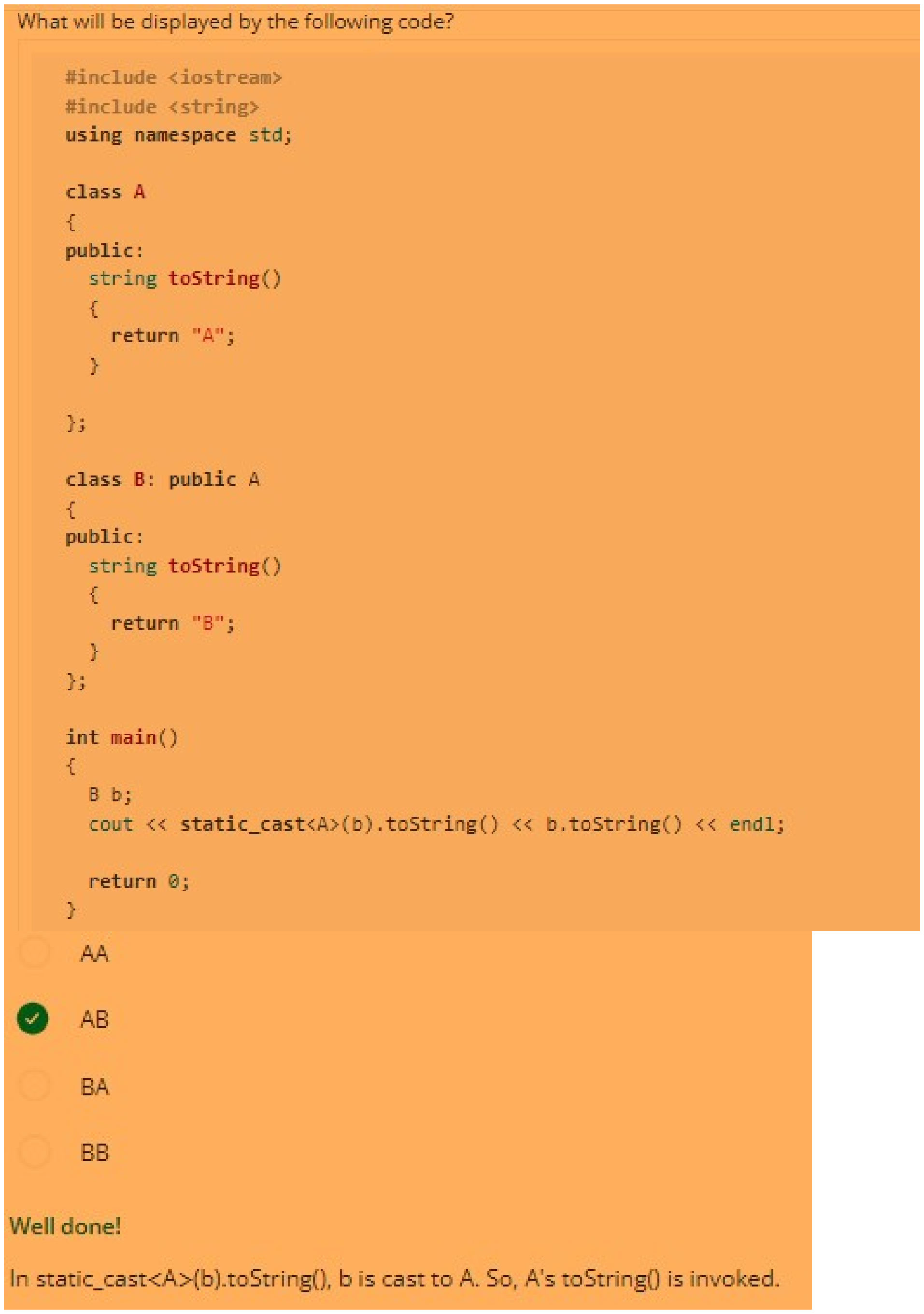
  [✓ The dynamic_cast operator can cast a primitive type variable to another primitive type.]
-
- 
  Suppose you declared GeometricObject* p = &object. To cast p to Circle, use __________.

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

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

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

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



- What will be displayed by the following code?

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

class A
{
public:
  string toString()
  {
    return "A";
  }

};

class B: public A
{
public:
  string toString()
  {
    return "B";
  }
};

int main()
{
  B b;
  cout << static_cast<A>(b).toString() << b.toString() << endl;

  return 0;
}
```

- AA

  [✓ AB]

  BA

  BB

  Well done!

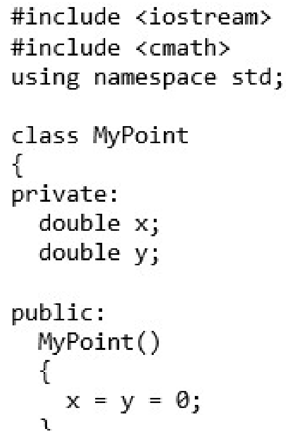
  In static_cast<A>(b).toString(), b is cast to A. So, A's toString() is invoked.

-

Ch 15 Summary

Saturday, April 22, 2023 6:18 PM

1. You can derive a new class from an existing class. This is known as class inheritance. The new class is called a derived class, child class, or subclass. The existing class is called a base class or parent class.
2. An object of a derived class can be passed wherever an object of a base type parameter is required. Then a function can be used generically for a wide range of object arguments. This is known as generic programming.
3. A constructor is used to construct an instance of a class. Unlike data fields and functions, the constructors of a base class are not inherited in the derived class. They can only be invoked from the constructors of the derived classes to initialize the data fields in the base class.
4. A derived class constructor always invokes its base class constructor. If a base constructor is not invoked explicitly, the base class no-arg constructor is invoked by default.
5. Constructing an instance of a class invokes the constructors of all the base classes along the inheritance chain.
6. A base class constructor is called from a derived class constructor. Conversely, the destructors are automatically invoked in reverse order, with the derived class's destructor invoked first. This is called constructor and destructor chaining.
7. A function defined in the base class may be redefined in the derived class. A redefined function must match the signature and return type of the function in the base class.
8. A virtual function enables dynamic binding. A virtual function is often redefined in the derived classes. The compiler decides which function implementation to use dynamically at runtime.
9. If a function defined in a base class needs to be redefined in its derived classes, you should define it virtual to avoid confusions and mistakes. On the other hand, if a function will not be redefined, it is more efficient not to declare it virtual, because more time and system resource are required to bind virtual functions dynamically at runtime.
10. A protected data field or a protected function in a base class can be accessed in its derived classes.
11. A pure virtual function is also called an abstract function.
12. If a class contains a pure virtual function, the class is called an abstract class.
13. You cannot create instances from an abstract class, but abstract classes can be used as data types for parameters in a function to enable generic programming.
14. You can use the static_cast and dynamic_cast operators to cast an object of a base class type to a derived class type. static_cast is performed at compile time and dynamic_cast is performed at runtime for runtime type checking. The dynamic_cast operator can only be performed on a polymorphic type (i.e., the type with virtual functions).



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

class MyPoint
{
private:
  double x;
  double y;

public:
  MyPoint()
  {
    x = y = 0;
  MyPoint()
  {
    x = y = 0;
  }

  MyPoint(double x, double y)
  {
    this->x = x;
    this->y = y;
  }

  double distance(const MyPoint& p2) const
  {
    return sqrt((x - p2.x) * (x - p2.x) + (y - p2.y) * (y - p2.y));
  }

  double getX() const
  {
    return x;
  }

  double getY() const
  {
    return y;
  }
};

class ThreeDPoint: public MyPoint
{
private:
  double z;

public:
  ThreeDPoint();
  ThreeDPoint(double x, double y, double z);
  double getZ() const;
  double distance(const ThreeDPoint& p2);
};
```

```
// Write the code to implement the constructors and functions in ThreePoint

ThreeDPoint::ThreeDPoint():MyPoint(){
    //x = 0;
    //y = 0;
    //MyPoint();
    z = 0;
}
ThreeDPoint::ThreeDPoint(double x, double y, double z):MyPoint(x, y){
    //double x1 = x;
    //double y1 = y;
    //MyPoint(x, y);
    this->z = z;
}
double ThreeDPoint::getZ() const{
    return z;
}
double ThreeDPoint::distance(const ThreeDPoint& p2){
    //d(P1,P2) = sqrt((x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2)
    double x1 = getX();
    double y1 = getY();
    double z1 = getZ();
    double x2 = p2.getX();
    double y2 = p2.getY();
    double z2 = p2.getZ();


    /*double x1 = 5;
    double y1 = 4;
    double z1 = 3;
    double x2 = 2;
    double y2 = 1;
    double z2 = 0;*/
    //double 2d distance =
    //cout<<x2<<" "<<x1<<endl;
    double distance = sqrt( (pow((x2-x1), 2)) +
                                (pow((y2-y1), 2)) +
                                (pow((z2-z1), 2)));

    return distance;
}

int main()
{
  // Write code here in the main function
  ThreeDPoint first;
  ThreeDPoint second = ThreeDPoint(10, 30, 25.5);
  //cout<<pow(7-2, 2)<<endl;
  cout<<"distance is "<<second.distance(first);
  return 0;
}
```