CSE 1325

Week of 11/23/2020

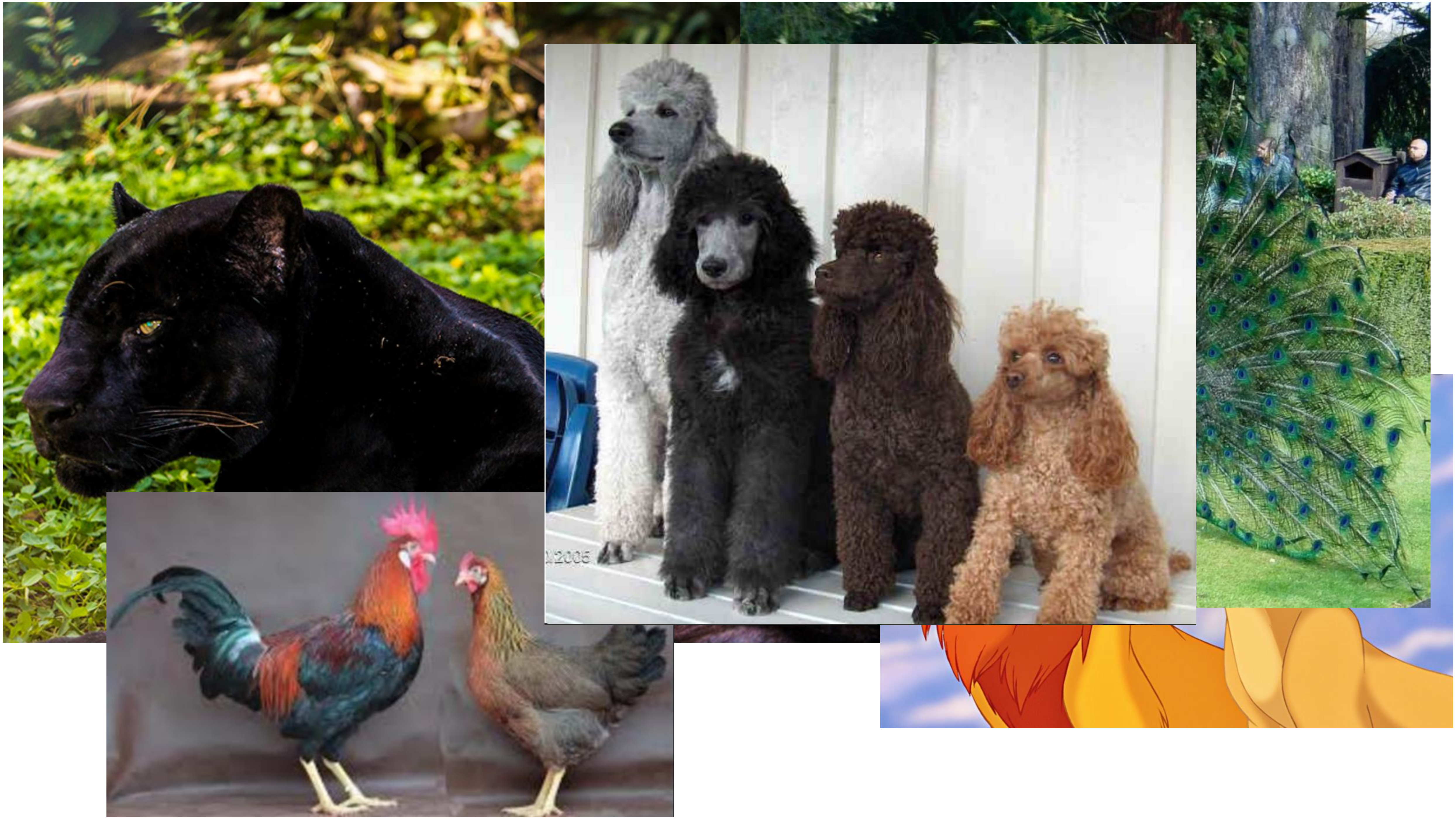
Instructor : Donna French

Polymorphism

Polymorphism occurs in biology.

Polymorphism in biology and zoology is the occurrence of two or more clearly different morphs or forms, also referred to as alternative phenotypes, in the population of a species.

To be classified as such, morphs must occupy the same habitat at the same time and belong to a panmictic population



Polymorphism

The word **polymorphism** means having many forms.

Typically, **polymorphism** occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

```
class Shape
{
  public :
    Shape(std::string name="BaseShape") : ShapeName{name}
    {
      std::cout << "SHAPE!" << std::endl;
    }
    std::string getName()
    {
      return ShapeName;
    }

    float dim1;
    float dim2;
    std::string ShapeName;
};
```

```
class Rectangle : public Shape
{
    public:
        Rectangle(std::string name, float height=0, float width=0
        : Shape(name)
        {
            dim1 = height;
            dim2 = width;
        }

        float getarea()
        {
            return dim1 * dim2;
        }
};
```

```
class Square : public Rectangle
{
    public:
        Square(std::string name, float size)
        : Rectangle(name, size)
        {
            dim1 = size;
            dim2 = size;
        }
    private:
        std::string location{"Line 68"};
};
```

```
class Circle : public Shape
{
    public:
        Circle(std::string name, float radius=0)
        : Shape()
        {
            dim1 = dim2 = radius;
        }

        float getarea()
        {
            return dim1 * dim2 * M_PI;
        }

    private :
        std::string color;
};
```

Seems like it would be a good idea for `getarea()` to live in the base class and be inherited?

```
class Shape
{
  public :
     Shape(std::string name="BaseShape") : ShapeName{name}
     {
       std::cout << "SHAPE!" << std::endl;
     }
     std::string getName()
     {
       return ShapeName;
     }
     float getarea()
     {
     }

     float dim1;
     float dim2;
     std::string ShapeName;
};
```

What would I put in here? "Shape" doesn't have an area!?

We want all of our derived classes to have/use the version of `getarea()` that does not take in any parameters and returns a `float`.

We don't want to allow a derived class to create a `getarea()` function that requires a parameter or returns an `int` for example.

Virtual Function

- Declaring a base class function as `virtual` allows derived classes to override that function's behavior which enables polymorphic behavior.
- An overridden function in a derived class has the same signature and return type (prototype) as the function it overrides in its base class.
- With `virtual` functions, the type of the object determines which version of a `virtual` function to invoke.
- If a base class function is not `virtual`, then the derived class could redefine the function.

```
class Shape
{
   public :
      Shape(std::string name="BaseShape") : ShapeName{name}
      {
         std::cout << "SHAPE!" << std::endl;
      }
      std::string getName()
      {
         return ShapeName;
      }
      virtual float getarea()
      {
      }

      float dim1;
      float dim2;
      std::string ShapeName;
};
```

`getarea()` is now a virtual function

shapeDemo.cpp

```
class Circle : public Shape
{
  public:
    Circle(float radius=0)
    {
      dim1 = dim2 = radius;
    }

    float getarea()
    {
      return dim1 * dim2 * M_PI;
    }
};
```

Circle inherits publicly from Shape
float dim1
float dim2

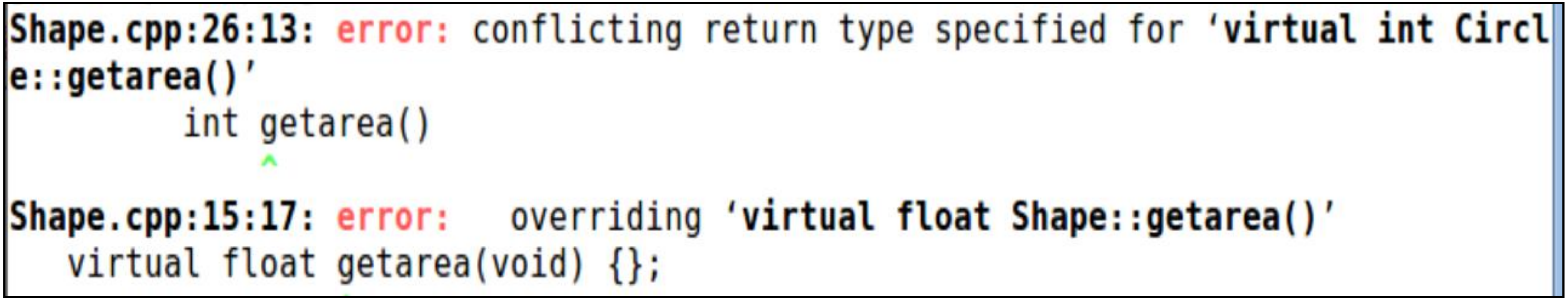
dim1 and dim2 are public members of Shape; therefore, Circle can set them directly

M_PI is define in <cmath>

Shape declared getarea() to be virtual which allows Circle to override the inherited behavior.

shapeDemo.cpp

```
class Circle : public Shape
{
    public:
        Circle(float radius=0)
        {
            dim1 = dim2 = radius;
        }

        float getarea()
        {
            return dim1 * dim2 * M_PI;
        }
};
```



```
Shape.cpp:26:13: error: conflicting return type specified for ‘virtual int Circl
e::getarea()’
        int getarea()
            ^

Shape.cpp:15:17: error:   overriding ‘virtual float Shape::getarea()’
   virtual float getarea(void) {};
```

Virtual Function

- Once a function is declared `virtual`, it remains `virtual` all the way down the inheritance hierarchy from that point, even if that function is not explicitly declared `virtual` when a derived class overrides it.

- When a derived class chooses not to override a `virtual` function from its base class, the derived class simply inherits its base class's `virtual` function implementation.

```
class Rectangle : public Shape
{
  public:
    Rectangle(float height=0, float width=0)
    {
      dim1 = height;
      dim2 = width;
    }

    float getarea()
    {
      return dim1 * dim2;
    }
};
```

Rectangle inherits publicly from Shape
float dim1
float dim2

Constructor with default parameters

dim1 and dim2 are public members of Shape; therefore, Rectangle can set them directly

Shape declared getarea() to be virtual which allows Rectangle to override the inherited behavior.
Note that this version of getarea() is different from Circle's version.

shapeDemo.cpp

```
class Square : public Rectangle
{
  public:
    Square(float size)
    {
      dim1 = size;
      dim2 = size;
    }
};
```

Square inherits publicly from Rectangle which inherits publicly from Shape
float dim1
float dim2

dim1 and dim2 are public members of Shape; therefore, Square can set them directly

Square inherited Rectangle's version of getarea(). No need to write a new version of getarea() since the area of a square is calculated using the same formula as a rectangle.

shapeDemo.cpp

```
int main (void)
{
   vector<Shape*>MyShapes;

   Circle C1(15.0);
   Rectangle R1(34.0,2.0);
   Circle C2(2.3);
   Rectangle R2(5.61,7.92);
   Square S1(3.33);

   MyShapes.push_back(&C1);
   MyShapes.push_back(&R1);
   MyShapes.push_back(&C2);
   MyShapes.push_back(&R2);
   MyShapes.push_back(&S1);

   for (auto it : MyShapes)
   {
      cout << "Area is " << it->getarea() << endl;
   }

   return 0;
}
```

`MyShapes` is a vector of pointers of class `Shape`

Instantiating `Circle`, `Rectangle` and `Square` objects.

Filling the vector with pointers to the objects

Use a range based `for` loop to display the vector contents

The elements of `MyShapes` are pointers; therefore, use pointer notation to call the member function `getarea()`

shapeDemo.cpp

```
Circle C1(15.0);
Rectangle R1(34.0,2.0);
Circle C2(2.3);
Rectangle R2(5.61,7.92);
Square S1(3.33);

cout << "Area is " << it->getarea() << endl;
```

```
Area is 706.858
Area is 68
Area is 16.619
Area is 44.4312
Area is 11.0889
```

shapeDemo.cpp

```
Circle C1(15.0);               MyShapes.push_back(&C1);
Rectangle R1(34.0,2.0);        MyShapes.push_back(&R1);
Circle C2(2.3);                MyShapes.push_back(&C2);
Rectangle R2(5.61,7.92);       MyShapes.push_back(&R2);
Square S1(3.33);               MyShapes.push_back(&S1);
```

```
(gdb) p MyShapes
$4 = std::vector of length 5, capacity 8 = {0x7fffffffe090,
0x7fffffffe0a0, 0x7fffffffe0b0, 0x7fffffffe0c0, 0x7fffffffe0d0}

(gdb) p &C1
$7 = (Circle *) 0x7fffffffe090

(gdb) p C1
$8 = {
  <Shape> = {
    _vptr.Shape = 0x401eb8 <vtable for Circle+16>,
    dim1 = 15,
    dim2 = 15
  }, <No data fields>}
```

shapeDemo.cpp

```
76              for (auto it : MyShapes)

(gdb) p it
$10 = (Shape *) 0x7fffffffe090
(gdb) p *it
$11 = {
  _vptr.Shape = 0x401eb8 <vtable for Circle+16>,
  dim1 = 15,
  dim2 = 15
}
```

```
(gdb) p &C1
$7 = (Circle *)
0x7fffffffe090
(gdb) p C1
$8 = {
  <Shape> = {
    _vptr.Shape = 0x401eb8
<vtable for Circle+16>,
    dim1 = 15,
    dim2 = 15
  }, <No data fields>}
```

```
78                      cout << "Area is " << it->getarea() << endl;
(gdb) step
Circle::getarea (this=0x7fffffffe090) at Shape.cpp:28
28                              return dim1 * dim2 * M_PI;
(gdb)
29                      }
(gdb)
Area is 706.858
```

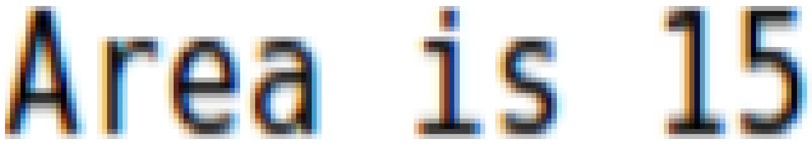
shapeDemo.cpp

What happens if we don't use pointers to the derived class instantiations?

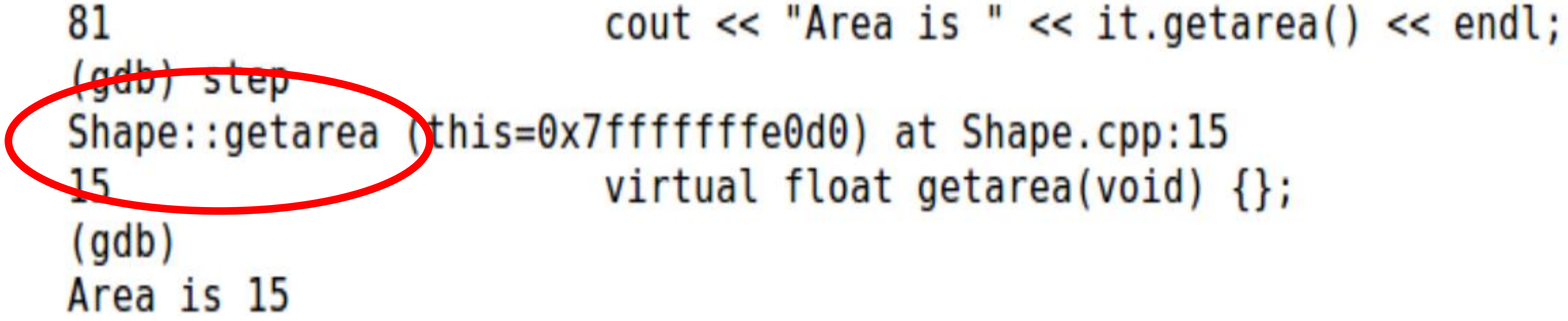
```
vector<Shape>MyShapes;

Circle C1(15.0);

MyShapes.push_back(C1);

for (auto it : MyShapes)
{
   cout << "Area is " << it.getarea() << endl;
}
```

No longer using pointer so use regular . notation to access member functions

`C1` is cast from `Circle` to `Shape` when it is pushed into a vector of type `Shape`. It then loses access to `Circle`'s `getarea()` and tries to use `Shape`'s `getarea()` which does not have a calculation in it which causes undefined behavior.



```
Area is 15
```



```
81                      cout << "Area is " << it.getarea() << endl;
(gdb) step
Shape::getarea (this=0x7fffffffe0d0) at Shape.cpp:15
15                      virtual float getarea(void) {};
(gdb)
Area is 15
```

shapeDemo.cpp

How do we add a private data member to Shape? dim1 and dim2 were both public.

Let's give each of our Shape objects a name by constructing each object with a name that matches the instantiation name. For example, object C1's name will be C1.

```
vector<Shape*>MyShapes;
```

No change

```
Circle C1("C1", 15.0);
Rectangle R1("R1", 34.0,2.0);
Circle C2("C2", 2.3);
Rectangle R2("R2", 5.61,7.92);
Square S1("S1", 3.33);
```

Passing a name in the constructor

```
MyShapes.push_back(&C1);
MyShapes.push_back(&R1);
MyShapes.push_back(&C2);
MyShapes.push_back(&R2);
MyShapes.push_back(&S1);
```

No change

```
class Shape
{
  public :
    float dim1;
    float dim2;
    virtual float getarea(void) {};

  private :
    string name;
};
```

shape1Demo.cpp

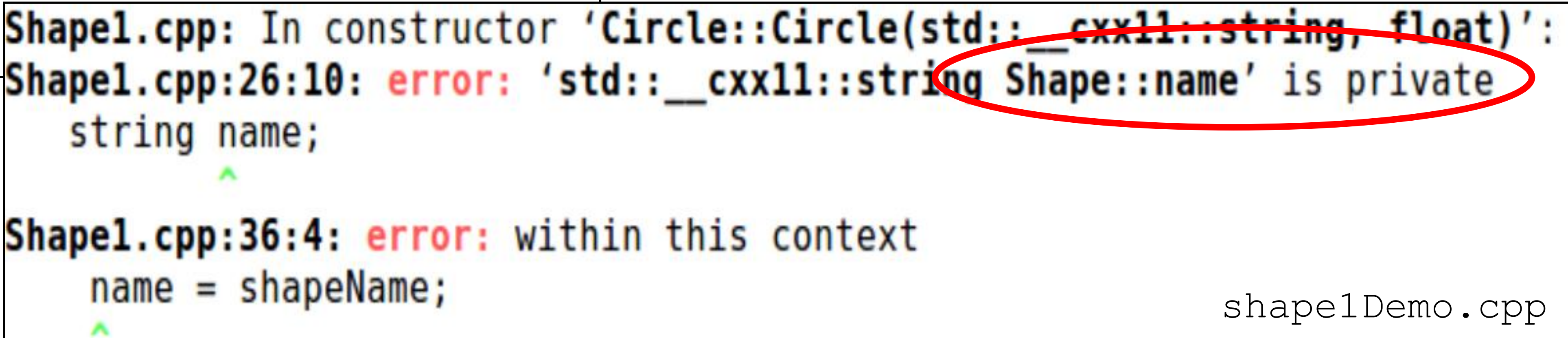
```
Circle C1(15.0);

class Circle : public Shape
{
   public:
      Circle(float radius=0)
      {
         dim1 = dim2 = radius;
      }

      float getarea()
      {
         return dim1 * dim2 * M_PI;
      }
};
```

```
Circle C1("C1", 15.0);

class Circle : public Shape
{
   public:
      Circle(string shapeName="",
             float radius=0)
      {
         dim1 = dim2 = radius
         name = shapeName;
      }

      float getarea()
      {
         return dim1 * dim2 * M_PI;
      }
};
```

shape1Demo.cpp

```
Circle C1("C1", 15.0);

class Circle : public Shape
{
   public:
      Circle(string shapeName="",
             float radius=0)
      {
         dim1 = dim2 = radius;
         name = shapeName;
      }

      float getarea()
      {
         return dim1 * dim2 * M_PI;
      }
};
```

C++ rigidly enforces restrictions on accessing private data members, so that *even a derived class* (which is intimately related to its base class) cannot access the base class's private data.



```
Shape1.cpp: In constructor 'Circle::Circle(std::__cxx11::string, float)':
Shape1.cpp:26:10: error: 'std::__cxx11::string Shape::name' is private
   string name;
          ^
Shape1.cpp:36:4: error: within this context
    name = shapeName;
    ^
```

shape1Demo.cpp

We could make our `private` data member `name` `protected` instead of `private`.

```
class Shape
{
     public :
          float dim1;
          float dim2;
          virtual float getarea(void) {};

     protected :
          string name;
};
```

This will compile just fine.

While this does allow direct access, using `protected` access is not always an option.

`shape1Demo.cpp`

```
class Shape
{
   public :
      float dim1;
      float dim2;
      virtual float getarea(void) {};
      string getName(void)
      {
         return name;
      }
      void setName(string shapeName)
      {
         name = shapeName;
      }

   private :
      string name;
};
```

member functions

shape1Demo.cpp

```
Circle C1("C1", 15.0);

class Circle : public Shape
{
   public:
      Circle(string shapeName="", float radius=0)
      {
         dim1 = dim2 = radius;
         setName(shapeName);
      }

      float getarea()
      {
         return dim1 * dim2 * M_PI;
      }
};
```

```
C1's area is 706.858
```

shape1Demo.cpp

```
class Rectangle : public Shape
{
public:
   Rectangle(string shapeName="", float height=0, float width=0)
   {
      dim1 = height;
      dim2 = width;
      setName(shapeName);
   }

   float getarea()
   {
      return dim1 * dim2;
   }
```

```
class Square : public Rectangle
{
   public:
      Square(string shapeName, float size)
      {
         dim1 = size;
         dim2 = size;
         setName(shapeName);
      }
};
```

shape1Demo.cpp

```
for (auto it : MyShapes)
{
   cout << it->getName() << "'s area is " << it->getarea() << endl;
}
```

```
C1's area is 706.858
R1's area is 68
C2's area is 16.619
R2's area is 44.4312
S1's area is 11.0889
```

shape1Demo.cpp

```
92          cout << it->getName() << "'s area is " << it->getarea() << endl;
(gdb) p it
$4 = (Shape *) 0x7fffffffe000
(gdb) p *it
$5 = {_vptr.Shape = 0x402940 <vtable for Circle+16>, dim1 = 15, dim2 = 15,
name = "C1"}
(gdb) step

Circle::getarea (this=0x7fffffffe000) at Shape1.cpp:40
40                      return dim1 * dim2 * M_PI;
(gdb)
41                  }
(gdb)
Shape::getName[abi:cxx11]() (this=0x7fffffffe000) at Shape1.cpp:18
18                      return name;
(gdb)
19                  }
(gdb)
```

C1's area is 706.858

shape1Demo.cpp

shape2Demo.cpp

```
vector<Shape*>MyShapes;

Circle C1("C1", 15.0);
Rectangle R1("R1", 34.0,2.0);
Circle C2("C2", 2.3);
Rectangle R2("R2", 5.61,7.92);
Square S1("S1", 3.33);

MyShapes.push_back(&C1);
MyShapes.push_back(&R1);
MyShapes.push_back(&C2);
MyShapes.push_back(&R2);
MyShapes.push_back(&S1);
```

automatic allocation
subject to scope lifetime rules

dynamic allocation
not subject to scope lifetime rules

```
vector<Shape*>MyShapes;

MyShapes.push_back(new Circle("C1", 15.0));
MyShapes.push_back(new Rectangle("R1", 34.0,2.0));
MyShapes.push_back(new Circle("C2", 2.3));
MyShapes.push_back(new Rectangle("R2", 5.61,7.92));
MyShapes.push_back(new Square("S1", 3.33));
```

```
class Shape
{
   public :
      Shape(string shapeName="")
      {
         name = shapeName;
      }
      float dim1;
      float dim2;
      virtual float getarea() {};
      string getName(void)
      {
         return name;
      }
      void setName(string shapeName)
      {
         name = shapeName;
      }
      void Hello(void)
      {
         cout << "Hi!  My class is Shape." << endl;
      }
   private :
      string name;
};
```

Added constructor with default parameters

Added member function

shape2Demo.cpp

```
class Square : public Rectangle
{
   public:
      Square(string shapeName, float size) : Rectangle()
      {
         dim1 = size;
         dim2 = size;
         setName(shapeName);
      }

      void Hello(void)
      {
         cout << "Hi!  My class is Square." << endl;
      }
};
```

```
void Hello(void)
{
   cout << "Hi!  My class is Rectangle." << endl;
}
```

```
void Hello(void)
{
   cout << "Hi!  My class is Circle." << endl;
}
```

shape2Demo.cpp

```
vector<Shape*>MyShapes;

MyShapes.push_back(new Shape("Shape1"));
MyShapes.push_back(new Circle("C1", 15.0));
MyShapes.push_back(new Rectangle("R1", 34.0,2.0));
MyShapes.push_back(new Circle("C2", 2.3));
MyShapes.push_back(new Rectangle("R2", 5.61,7.92));
MyShapes.push_back(new Square("S1", 3.33));



for (auto it : MyShapes)
{

   it->Hello();

   cout << it->getName() << "'s area is " << it->getarea() << endl;

}
```

Create a `Shape`

Call `Hello()` for each object in `MyShapes`

shape2Demo.cpp

```
Hi!  My class is Shape.
Shape1's area is 0
Hi!  My class is Shape.
C1's area is 706.858
Hi!  My class is Shape.
R1's area is 68
Hi!  My class is Shape.
C2's area is 16.619
Hi!  My class is Shape.
R2's area is 44.4312
Hi!  My class is Shape.
S1's area is 11.0889
Hi!  My class is Shape.
C3's area is 706.858
```

The version of `Hello()` in `Shape()` was encountered first; therefore, was the one that was executed.

The versions of `Hello()` in the derived classes were unreachable and were not executed.

`shape2Demo.cpp`

```
class Shape
{
   public :
      Shape(string shapeName="")
      {
         name = shapeName;
      }
      float dim1;
      float dim2;
      virtual float getarea() {};
      string getName(void)
      {
         return name;
      }
      void setName(string shapeName)
      {
         name = shapeName;
      }
      virtual void Hello(void)
      {
         cout << "Hi!  My class is Shape." << endl;
      }
   private :
      string name;
};
```

Added `virtual`

```
Hi!  My class is Shape.
Shape1's area is 0
Hi!  My class is Circle.
C1's area is 706.858
Hi!  My class is Rectangle.
R1's area is 68
Hi!  My class is Circle.
C2's area is 16.619
Hi!  My class is Rectangle.
R2's area is 44.4312
Hi!  My class is Square.
S1's area is 11.0889
```

shape2Demo.cpp

```
Hi!  My class is Shape.
Shape1's area is 0
Hi!  My class is Circle.
C1's area is 706.858
Hi!  My class is Rectangle.
R1's area is 68
Hi!  My class is Circle.
C2's area is 16.619
Hi!  My class is Rectangle.
R2's area is 44.4312
Hi!  My class is Square.
S1's area is 11.0889
```

No changes were made to the derived class's `Hello()`.

Now that the base class version of `Hello()` is `virtual`, the derived classes use their own versions.

`Rectangle`'s `Hello`() is `virtual` because the base class's version is `virtual`; therefore, `Square` uses its version.

Once declared `virtual`, any derived class version will be `virtual`.

Base class can still use the `virtual` function.

`shape2Demo.cpp`

Derived classes do not inherit destructors and do not have default destructor.

So what happens when we use `delete`?

```
for (auto it : MyShapes)
{
   delete it;
}

class Shape
{
   public :
      Shape(string shapeName="")
      {
         name = shapeName;
      }
      ~Shape()
      {
         cout << "Destroying Shape" << endl;
      }
```

Added message to destructor so when we can see when it runs

```
Hi!  My class is Shape.
Shape1's area is 0
Hi!  My class is Circle.
C1's area is 706.858
Hi!  My class is Rectangle.
R1's area is 68
Hi!  My class is Circle.
C2's area is 16.619
Hi!  My class is Rectangle.
R2's area is 44.4312
Hi!  My class is Square.
S1's area is 11.0889
Destroying Shape
Destroying Shape
Destroying Shape
Destroying Shape
Destroying Shape
Destroying Shape
```

Destructor in `Shape` is called.

Resources specific to the derived class would not be released

So we need to make the destructor virtual

```
class Shape
{
   public :
      Shape(string shapeName="")
      {
         name = shapeName;
      }
      virtual ~Shape()
      {
         cout << "Destroying Shape" << endl;
      }
```

```
Hi!  My class is Shape.
Shape1's area is 0
Hi!  My class is Circle.
C1's area is 706.858
Hi!  My class is Rectangle.
R1's area is 68
Hi!  My class is Circle.
C2's area is 16.619
Hi!  My class is Rectangle.
R2's area is 44.4312
Hi!  My class is Square.
S1's area is 11.0889
Destroying Shape
Destroying Shape
Destroying Shape
Destroying Shape
Destroying Shape
Destroying Shape
```

Destructor in `Shape` is `virtual` but still called because we did not create override versions in our derived classes.

```
~Circle()
{
    cout << "Destroying Circle" << endl;
}

~Rectangle()
{
    cout << "Destroying Rectangle" << endl;
}

~Square()
{
    cout << "Destroying Square" << endl;
}
```

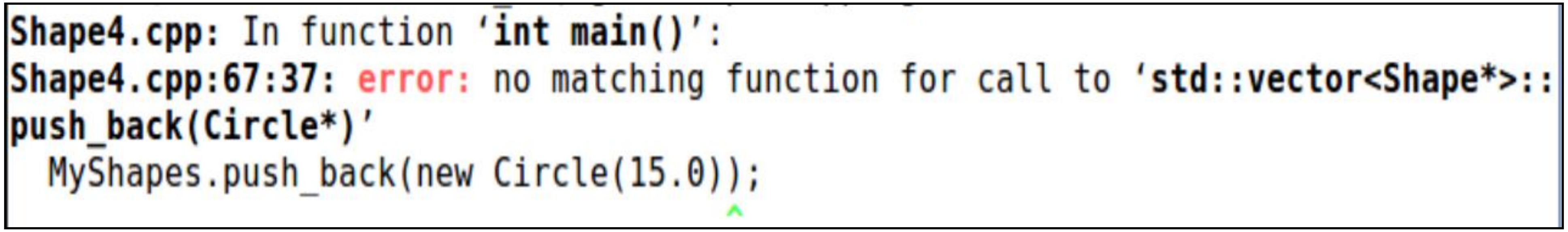
```
Hi!  My class is Shape.
Shape1's area is 0
Hi!  My class is Circle.
C1's area is 706.858
Hi!  My class is Rectangle.
R1's area is 68
Hi!  My class is Circle.
C2's area is 16.619
Hi!  My class is Rectangle.
R2's area is 44.4312
Hi!  My class is Square.
S1's area is 11.0889
Destroying Shape
Destroying Circle
Destroying Shape
Destroying Rectangle
Destroying Shape
Destroying Circle
Destroying Shape
Destroying Rectangle
Destroying Shape
Destroying Square
Destroying Rectangle
Destroying Shape
```

When a derived class object is destroyed, the base class part of the derived class is also destroyed

What happens if `Circle` is not a derived class of `Shape`?

```
class Circle
{
  public
     Circle(float xradius=0)
     {
        radius = xradius;
     }

     float getarea()
     {
        return radius * radius * M_PI;
     }
};
```

```
vector<Shape*>MyShapes;

MyShapes.push_back(new Circle("C1", 15.0));
```



```
Shape4.cpp: In function ‘int main()’:
Shape4.cpp:67:37: error: no matching function for call to ‘std::vector<Shape*>::
push_back(Circle*)’
  MyShapes.push_back(new Circle(15.0));
                                    ^
```

How to add a new class/shape

```
class Triangle : public Shape
{
   public:
       Triangle(string shapeName="", float base=0, float height=0)
       {
          dim1 = base;
          dim2 = height;
          setName(shapeName);
       }
       ~Triangle()
       {
          cout << "Destroying Triangle" << endl;
       }
       float getarea()
       {
          return dim1 * dim2 * 0.5;
       }
       void Hello(void)
       {
          cout << "Hi!  My class is Triangle." << endl;
       }
};
```

```
vector<Shape*>MyShapes;

MyShapes.push_back(new Shape("Shape1"));
MyShapes.push_back(new Circle("C1", 15.0));
MyShapes.push_back(new Rectangle("R1", 34.0,2.0));
MyShapes.push_back(new Circle("C2", 2.3));
MyShapes.push_back(new Rectangle("R2", 5.61,7.92));
MyShapes.push_back(new Square("S1", 3.33));
MyShapes.push_back(new Triangle("T1", 3.0, 6.0));

for (auto it : MyShapes)
{
   it->Hello();
   cout << it->getName() << "'s area is " << it->getarea() << endl;
}

for (auto it : MyShapes)
{
   delete it;
}
```

```
Hi!  My class is Shape.           Destroying Shape
Shape1's area is 0                Destroying Circle
Hi!  My class is Circle.          Destroying Shape
C1's area is 706.858              Destroying Rectangle
Hi!  My class is Rectangle.       Destroying Shape
R1's area is 68                   Destroying Circle
Hi!  My class is Circle.          Destroying Shape
C2's area is 16.619               Destroying Rectangle
Hi!  My class is Rectangle.       Destroying Shape
R2's area is 44.4312              Destroying Square
Hi!  My class is Square.          Destroying Rectangle
S1's area is 11.0889              Destroying Shape
Hi!  My class is Triangle.        Destroying Triangle
T1's area is 9                    Destroying Shape
```

Abstract Classes

An abstract class is a class from which you never intend to instantiate any objects.

Because these classes normally are used as base classes in inheritance hierarchies, they are referred to as abstract base classes.

These classes cannot be used to instantiate object.

Abstract classes are typically incomplete.

Abstract Classes

An abstract class is a base class from which other classes can inherit.

Classes that can be used to instantiate objects are called concrete classes.

Concrete classes define or inherit implementations for every member function they declare.

Abstract Classes

`Shape`

abstract class

`Circle, Rectangle, Square, Triangle`

concrete class

Abstract base classes are too generic to define real objects.

If I asked you to calculate the area of a `Shape`, how would you do that?

Abstract Classes

A class is made abstract by declaring one or more of its `virtual` functions to be "pure".

A pure `virtual` function is specified by placing "`= 0`" in its declaration.

```
virtual float getarea() = 0;
```

The "`= 0`" is a pure specifier.

Forces derived classes to override the `virtual` function. Non pure `virtual` functions do not force derived classes to implement overrides.

Abstract Classes

`virtual` function

has an implementation and gives the derived class the option of overriding the function

pure `virtual` function

does not have an implementation and requires the derived class to override the function

Abstract Classes

When should a function be set as pure `virtual?`

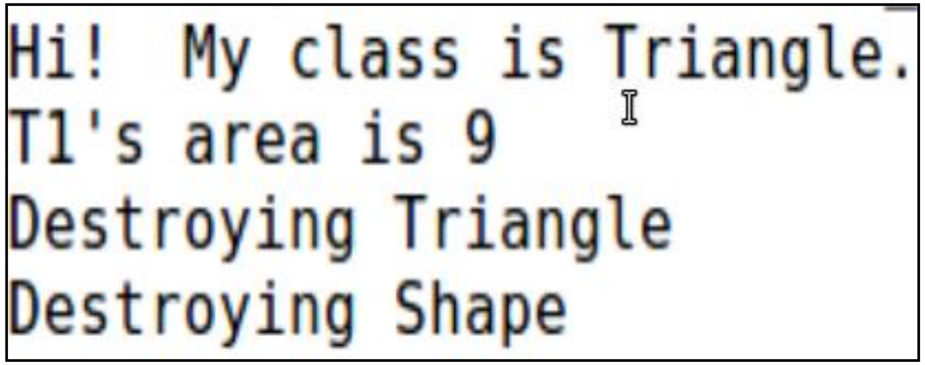
When the function implementation does not make sense for the base class and you want to force all concrete derived classes to implement the function.

If a derived class does not override the function, then the derived class remains abstract and cannot be concrete. Objects cannot be instantiated from that derived class (compiler errors).

```
class Triangle : public Shape
{
   public:
      Triangle(string shapeName="", float base=0, float height=0)
      {
         dim1 = base;
         dim2 = height;
         setName(shapeName);
      }
      ~Triangle()
      {
         cout << "Destroying Triangle" << endl;
      }
      float getarea()
      {
         return dim1 * dim2 * 0.5;
      }
      void Hello(void)
      {
         cout << "Hi!  My class is Triangle." << endl;
      }
};
```

```
class Shape
{
   public :
      virtual float getarea() {};
};
```

`getarea()` in `Shape` is `virtual` and `Triangle` is overriding it



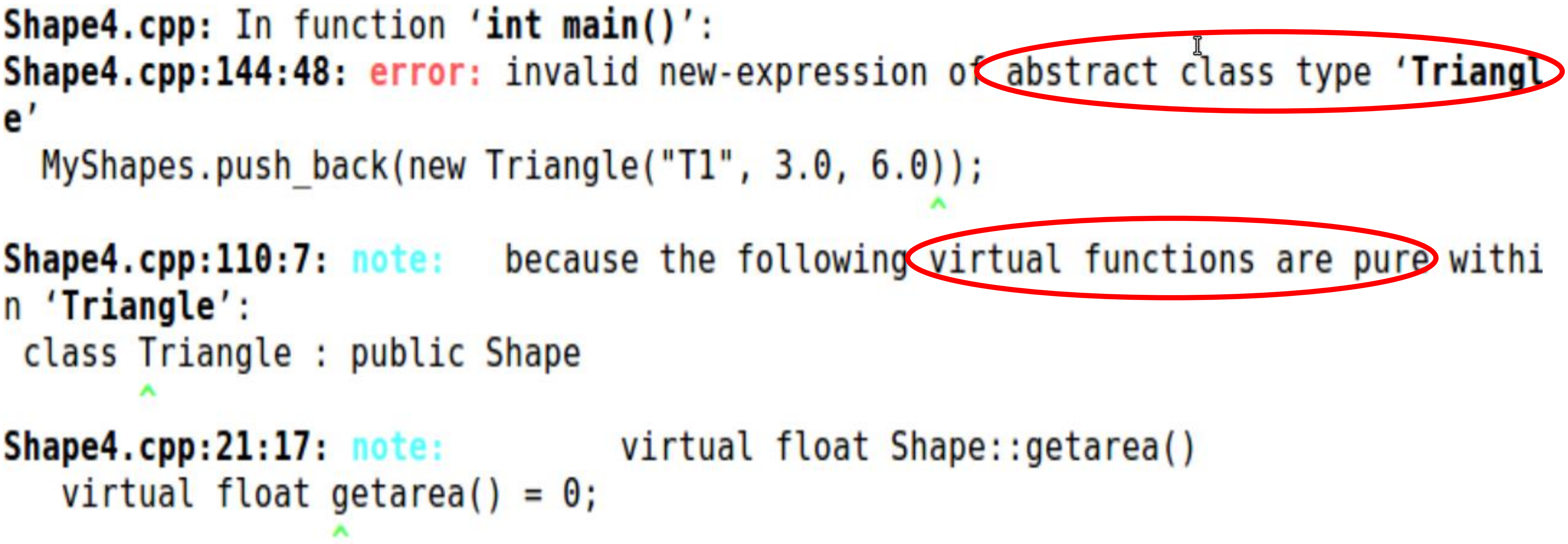
```
Hi!  My class is Triangle.
T1's area is 9
Destroying Triangle
Destroying Shape
```

```
class Triangle : public Shape
{
   public:
       Triangle(string shapeName="", float base=0, float height=0)
       {
          dim1 = base;
          dim2 = height;
          setName(shapeName);
       }
       ~Triangle()
       {
          cout << "Destroying Triangle" << endl;
       }
       void Hello(void)
       {
          cout << "Hi!  My class is Triangle." << endl;
       }
};
```

removed the overridden version of `getarea()`

```
class Shape
{
   public :
       virtual float getarea() = 0;
};
```

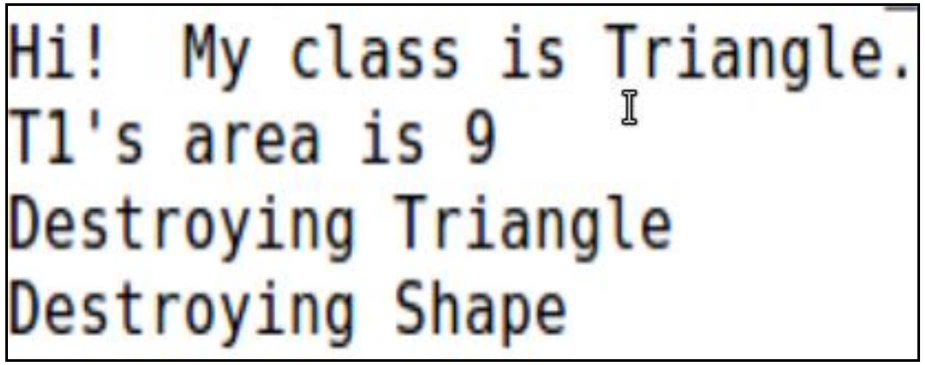
pure is not a keyword use "`= 0`"



```
Shape4.cpp: In function 'int main()':
Shape4.cpp:144:48: error: invalid new-expression of abstract class type 'Triangl
e'
  MyShapes.push_back(new Triangle("T1", 3.0, 6.0));
                                                ^

Shape4.cpp:110:7: note:   because the following virtual functions are pure withi
n 'Triangle':
 class Triangle : public Shape
       ^

Shape4.cpp:21:17: note:         virtual float Shape::getarea()
   virtual float getarea() = 0;
                 ^
```

```
class Triangle : public Shape
{
   public:
      Triangle(string shapeName="", float base=0, float height=0)
      {
         dim1 = base;
         dim2 = height;
         setName(shapeName);
      }
      ~Triangle()
      {
         cout << "Destroying Triangle" << endl;
      }
      float getarea()
      {
         return dim1 * dim2 * 0.5;
      }
      void Hello(void)
      {
         cout << "Hi!  My class is Triangle." << endl;
      }
};
```

```
class Shape
{
   public :
      virtual float getarea() = 0;
};
```



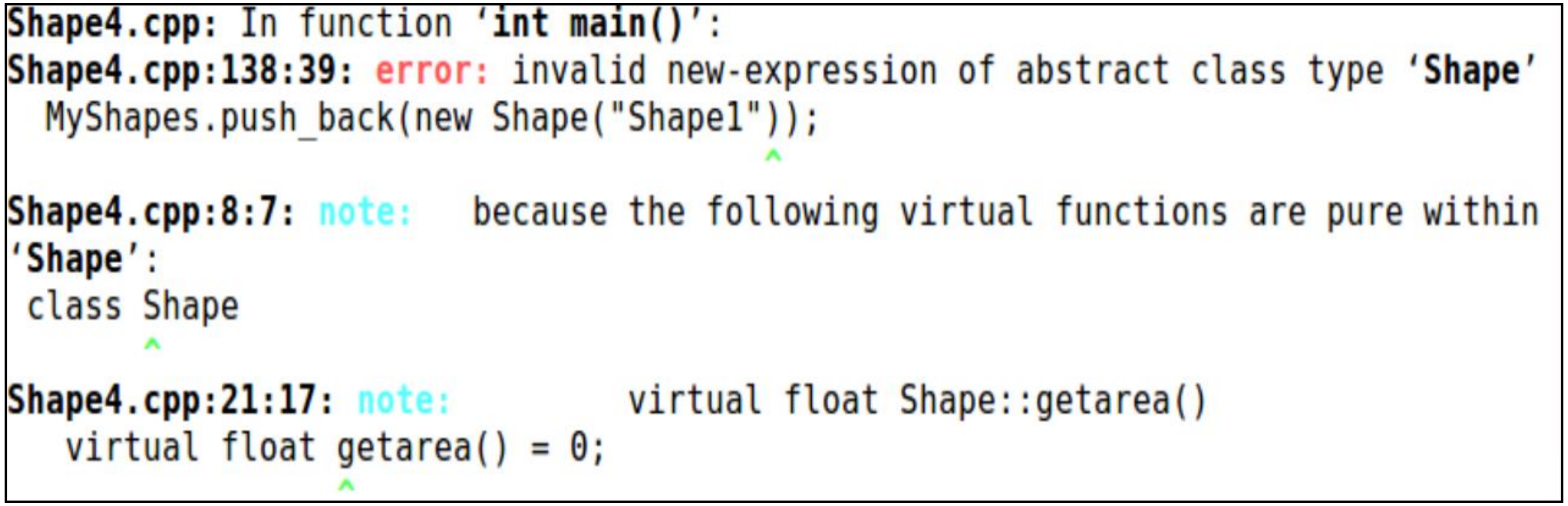
```
Hi!  My class is Triangle.
T1's area is 9
Destroying Triangle
Destroying Shape
```

With `getarea()` now set as a pure `virtual` function

```
class Shape
{
   public :
      virtual float getarea() = 0;
```

We can no longer instantiate objects from `Shape`. `Shape` is now an abstract class and cannot be instantiated.

```
MyShapes.push_back(new Shape("Shape1"));
```



```
Shape4.cpp: In function ‘int main()’:
Shape4.cpp:138:39: error: invalid new-expression of abstract class type ‘Shape’
  MyShapes.push_back(new Shape("Shape1"));
                                       ^
Shape4.cpp:8:7: note:   because the following virtual functions are pure within
‘Shape’:
 class Shape
       ^
Shape4.cpp:21:17: note:         virtual float Shape::getarea()
   virtual float getarea() = 0;
                 ^
```